USENIX Association

Proceedings of the 16th USENIX Conference on File and Storage Technologies

February 12–15, 2018
Oakland, CA, USA

Conference Organizers

Program Chairs

Nitin Agrawal, *Samsung Research*
Raju Rangaswami, *Florida International University*

Program Committee

Andrea Arpaci-Dusseau, *University of Wisconsin—Madison*
Anirudh Badam, *Microsoft Research*
Mahesh Balakrishnan, *Yale*
Pramod Bhatotia, *University of Edinburgh*
Andre Brinkmann, *Johannes Gutenberg-University Mainz*
Vijay Chidambaram, *University of Texas at Austin*
Angela Demke Brown, *University of Toronto*
Kevin Greenan, *Box Inc.*
Jorge Guerra, *VMWare*
Haryadi Gunawi, *University of Chicago*
Dean Hildebrand, *Google*
Cheng Huang, *Microsoft Azure*
Hong Jiang, *University of Texas at Arlington*
Umesh Maheshwari, *Nimble Storage (an HPE company)*
Arif Merchant, *Google*
Ethan L. Miller, *University of California, Santa Cruz, and Pure Storage*
Dushyanth Narayanan, *Microsoft Research*
Ed Nightingale, *Microsoft Research*
Sam H. Noh, *UNIST (Ulsan National Institute of Science and Technology)*
Vijayan Prabhakaran, *Amazon*
Bianca Schroeder, *University of Toronto*
Philip Shilane, *Dell EMC*
Keith A. Smith, *NetApp*
Vasily Tarasov, *IBM Research*
Cristian Ungureanu, *Google*
Ashish Vulimiri, *Samsung Research*
Andrew Warfield, *University of British Columbia*
Sage Weil, *Red Hat*
Youjip Won, *Hanyang University*
Gala Yadgar, *Technion - Israel Institute of Technology*
Ming Zhao, *Arizona State University*

Work-in-Progress/Posters Co-Chairs

Anirudh Badam, *Microsoft Research*
Gala Yadgar, *Technion - Israel Institute of Technology*

Test of Time Awards Committee

Bianca Schroeder, *University of Toronto*
Eno Thereska, *Amazon*

Tutorial Coordinators

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

Steering Committee

Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
Angela Demke Brown, *University of Toronto*
Greg Ganger, *Carnegie Mellon University*
Casey Henderson, *USENIX Association*
Kimberly Keeton, *HP Labs*
Geoff Kuenning, *Harvey Mudd College*
Florentina Popovici, *Google*
Erik Riedel, *Dell Technologies*
Jiri Schindler, *SimpliVity*
Bianca Schroeder, *University of Toronto*
Margo Seltzer, *Harvard University and Oracle*
Keith A. Smith, *NetApp*
Eno Thereska, *Amazon*
Carl Waldspurger, *Carl Waldspurger Consulting*
Ric Wheeler, *Red Hat*
Erez Zadok, *Stony Brook University*

External Reviewers

Ram Alagappan
Peter Alvaro
Leo Prasath Arulraj
Surendar Chandra
Aishwarya Ganesan
Jun He
Sudarsun Kannan
Eunji Lee
Jing Liu
Peter Macko
Vishal Misra
Yuvraj Patel
Radu Sion
Zev Weiss
Kan Wu
Suli Yang
Dennis Zhou

Message from the FAST '18 Program Co-Chairs

Welcome to the 16th 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 reliability, flash and persistent memory, cloud and distributed storage, coding and hashing, and traditional file systems. Our authors hail from ten countries on three continents and represent academia, industry, and the open-source community.

FAST '18 received 139 submissions. Of these we selected 23, for an acceptance rate of 17%. The Program Committee used a two-round online review process and then met in person to select the final program. In the first round, each paper received at least three reviews. For the second round, 56 papers received at least two more reviews. The Program Committee discussed 40 papers in an all-day meeting on December 8, 2017, at Samsung Research Laboratory in Mountain View, California. We used Eddie Kohler's superb HotCRP software to manage all stages of the review process, from submission to author notification.

As in the previous years, we included a category of short papers. Short papers provide a vehicle for presenting completed research that does not require a full-length paper to describe and evaluate. We received 26 short paper submissions of which 2 were accepted. In what we hope will become a new FAST tradition, we again included a category of deployed-systems papers, which address experience with the practical design, implementation, analysis or deployment of large-scale, operational systems. We received 10 deployed-systems submissions, of which we accepted 3.

We wish to thank the many people who contributed to this conference. First and foremost, we are grateful to all the authors who submitted their work to FAST '18. We would also like to thank the attendees of FAST '18 and the future readers of these papers. Together with the authors, you form the FAST community and make storage research vibrant and exciting. We extend our thanks to the USENIX staff, especially Casey Henderson, Hilary Hartman, and Michele Nelson, who have provided outstanding support throughout the planning and organizing of this conference with the highest degree of professionalism and friendliness. Most importantly, their behind-the-scenes work makes this conference actually happen. We would like to thank the Poster and Work-in-Progress session Chairs, Anirudh Badam and Gala Yadgar, and the Test of Time Awards Chairs, Bianca Schroeder and Eno Thereska. Our thanks also go also to the members of the FAST Steering Committee, who provided invaluable advice and feedback, and to our Steering Committee Liaison, Keith Smith, for his guidance and encouragement on many issues, large and small, over the past year.

Finally, we wish to thank our Program Committee for their many hours of hard work reviewing and discussing the submissions and for taking time from their busy schedules to travel from around the globe to attend our PC meeting in person. Together with a few external reviewers, they wrote 535 thoughtful and meticulous reviews. HotCRP recorded over 186,000 words in reviews and comments (excluding HotCRP boilerplate; 377K when included). The reviewers' evaluations, and their thorough and conscientious deliberations at the PC meeting, contributed significantly to the quality of our decisions. Finally, we also thank Mohammad Ataur Rahman Chowdhury, the scribe for the PC meeting. We look forward to an interesting and enjoyable conference!

Nitin Agrawal, *Samsung Research*
Raju Rangaswami, *Florida International University*
FAST '18 Program Co-Chairs

FAST '18: 16th USENIX Conference on File and Storage Technologies
February 12–15, 2018
Oakland, CA, USA

Failing and Recovering

Revealing Flashy Secrets

Understanding the Meta(data) Story

(continued on next page)

Coding, Hashing, Hiding

New Media and Old

Long Live the File System!

Distribute and Conquer

(continued on next page)

Dedup: Last but Not Least

Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems

Haryadi S. Gunawi[1], Riza O. Suminto[1], Russell Sears[2], Casey Golliher[2], Swaminathan Sundararaman[3], Xing Lin[4], Tim Emami[4], Weiguang Sheng[5], Nematollah Bidokhti[5], Caitie McCaffrey[6], Gary Grider[7], Parks M. Fields[7], Kevin Harms[8], Robert B. Ross[8], Andree Jacobson[9], Robert Ricci[10], Kirk Webb[10], Peter Alvaro[11], H. Birali Runesha[12], Mingzhe Hao[1], and Huaicheng Li[1]

[1]University of Chicago, [2]Pure Storage, [3]Parallel Machines, [4]NetApp, [5]Huawei, [6]Twitter, [7]Los Alamos National Laboratory, [8]Argonne National Laboratory, [9]New Mexico Consortium, [10]University of Utah, [11]University of California, Santa Cruz, and [12]UChicago Research Computing Center

Abstract

Fail-slow hardware is an under-studied failure mode. We present a study of 101 reports of fail-slow hardware incidents, collected from large-scale cluster deployments in 12 institutions. We show that all hardware types such as disk, SSD, CPU, memory and network components can exhibit performance faults. We made several important observations such as faults convert from one form to another, the cascading root causes and impacts can be long, and fail-slow faults can have varying symptoms. From this study, we make suggestions to vendors, operators, and systems designers.

1 Introduction

Understanding fault models is an important criteria of building robust systems. Decades of research has developed mature failure models such as fail-stop [3, 22, 30, 32, 35], fail-partial [6, 33, 34], fail-transient [26], faults as well as corruption [7, 18, 20, 36] and byzantine failures [14].

This paper highlights an under-studied "new" failure type: *fail-slow hardware*, hardware that is still running and functional but in a degraded mode, slower than its expected performance. We found that all major hardware components can exhibit fail-slow faults. For example, disk throughput can drop by three orders of magnitude to 100 KB/s due to vibration, SSD operations can stall for seconds due to firmware bugs, memory cards can degrade to 25% of normal speed due to loose NVDIMM connection, CPUs can unexpectedly run in 50% speed due to lack of power, and finally network card performance can collapse to Kbps level due to buffer corruption and retransmission.

While fail-slow hardware arguably did not surface frequently in the past, today, as systems are deployed at scale, along with many intricacies of large-scale operational conditions, the probability that a fail-slow hardware incident can occur increases. Furthermore, as hardware technology continues to scale (smaller and more complex), today's hardware development and manufacturing will only exacerbate the problem.

Unfortunately, fail-slow hardware is under-studied. A handful of prior papers already hinted the urgency of this problem; many different terms have been used such as "fail-stutter" [4], "gray failure" [25], and "limp mode" [17, 21, 27]. However, the discussion was not solely focused on hardware but mixed with software performance faults as well. We counted roughly only 8 stories per paper of fail-slow hardware mentioned in these prior papers, which is probably not sufficient enough to convince the systems community of this urgent problem.

To fill the void of strong evidence of hardware performance faults in the field, we, a group of researchers, engineers, and operators of large-scale datacenter systems across 12 institutions decided to write this "community paper." More specifically, we have collected 101 detailed reports of fail-slow hardware behaviors including the hardware types, root causes, symptoms, and impacts to high-level software. To the best of our knowledge, this is the most complete account of fail-slow hardware in production systems reported publicly.

Due to space constraints, we summarize our unique and important findings in Table 1 and do not repeat them here. The table also depicts the organization of the paper. Specifically, we first provide our high-level observations (§3), then detail the fail-slow incidents with internal root causes (§4) as well as external factors (§5), and finally provide suggestions to vendors, operators, and systems designers (§6). We hope that our paper will spur more studies and solutions to this problem.

2 Methodology

101 reports of fail-slow hardware were collected from large-scale cluster deployments in 12 institutions (Table

Important Findings and Observations
§3.1 **Varying root causes:** Fail-slow hardware can be induced by internal causes such as firmware bugs or device errors/wear-outs as well as external factors such as configuration, environment, temperature, and power issues.
§3.2 **Faults convert from one form to another:** Fail-stop, -partial, and -transient faults can convert to fail-slow faults (*e.g.*, the overhead of frequent error masking of corrupt data can lead to performance degradation).
§3.3 **Varying symptoms:** Fail-slow behavior can exhibit a permanent slowdown, transient slowdown (up-and-down performance), partial slowdown (degradation of sub-components), and transient stop (*e.g.*, occasional reboots).
§3.4 **A long chain of root causes:** Fail-slow hardware can be induced by a long chain of causes (*e.g.*, a fan stopped working, making other fans run at maximal speeds, causing heavy vibration that degraded the disk performance).
§3.4 **Cascading impacts:** A fail-slow hardware can collapse the entire cluster performance; for example, a degraded NIC made many jobs lock task slots/containers in healthy machines, hence new jobs cannot find enough free slots.
§3.5 **Rare but deadly (long time to detect):** It can take hours to months to pinpoint and isolate a fail-slow hardware due to many reasons (*e.g.*, no full-stack visibility, environment conditions, cascading root causes and impacts).
Suggestions
§6.1 **To vendors:** When error masking becomes more frequent (*e.g.*, due to increasing internal faults), more explicit signals should be thrown, rather than running with a high overhead. Device-level performance statistics should be collected and reported (*e.g.*, via S.M.A.R.T) to facilitate further studies.
§6.2 **To operators:** 39% root causes are external factors, thus troubleshooting fail-slow hardware must be done online. Due to the cascading root causes and impacts, full-stack monitoring is needed. Fail-slow root causes and impacts exhibit some correlation, thus statistical correlation techniques may be useful (with full-stack monitoring).
§6.3 **To systems designers:** While software systems are effective in handling fail-stop (binary) model, more research is needed to tolerate fail-slow (non-binary) behavior. System architects, designers and developers can fault-inject their systems with all the root causes reported in this paper to evaluate the robustness of their systems.

Table 1: **Summary of our findings and suggestions.**

Institution	#Nodes
Company 1	>10,000
Company 2	150
Company 3	100
Company 4	>1,000
Company 5	>10,000

Institution	#Nodes
Univ. A	300
Univ. B	>100
Univ. C	>1,000
Univ. D	500
Nat'l Labs X	>1,000
Nat'l Labs Y	>10,000
Nat'l Labs Z	>10,000

Table 2: **Operational scale.**

2). At such scales, it is more likely to witness fail-slow hardware occurrences. The reports were all unformatted text, written by the engineers and operators who still vividly remember the incidents due to the severity of the impacts. The incidents were reported between 2000 and 2017, with only 30 reports predating 2010. Each institution reports a unique set of root causes. For example, although an institution may have seen a corrupt buffer being the root cause that slows down networking hardware (packet loss and retransmission) many times, it is only collected as one report. Thus, a single report can represent multiple instances of the incident. If multiple different institutions report the same root cause, it is counted multiple times. However, the majority of root causes (66%) are unique and only 22% are duplicates (12% reports did not pinpoint a root causes). More specifically, a duplicated incident is reported on average by 2.4 institutions; for example, firmware bugs are reported from 5 institutions, driver bugs from 3 institutions, and the remaining issues from 2 institutions. The raw (partial) dataset can be downloaded on our group website [2].

We note that there is no analyzable hardware-level performance logs (more in §6.1), which prevents large-scale log studies. We strongly believe that there were many more cases that were slipped and unnoticed. Some stories are also not passed around as operators change jobs. We do not include known slowdowns (*e.g.*, random IOs causing slow disks, or GC activities occasionally slowing down SSDs). We only include reports of *unexpected* degradation. For example, unexpected hardware faults that make GC activities work harder is reported.

3 Observations (Take-Away Points)

From this study, we made five important high-level findings as summarized in Table 1.

3.1 Varying Root Causes

Pinpointing the root cause of a fail-slow hardware is a daunting task as it can be induced by a variety of root causes, as shown in Table 3. Hardware performance fault can be caused by *internal* root causes from within the device such as *firmware issues* (`FW`) or *device errors/wear-outs* (`ERR`), which will be discussed in Section 4. However, a perfectly working device can also be degraded by many *external* root causes such as *configuration* (`CONF`), *environment* (`ENV`), *temperature* (`TEMP`), and *power* (`PWR`) related issues, which will be presented in Section §5.

	Hardware types					
Root	**SSD**	**Disk**	**Mem**	**Net**	**CPU**	**Total**
ERR	10	8	9	10	3	40
FW	6	3	0	9	2	20
TEMP	1	3	0	2	5	11
PWR	1	0	1	0	6	8
ENV	3	5	2	4	4	18
CONF	1	1	0	2	3	7
UNK	0	3	1	2	2	8
Total	22	23	13	29	25	112

Table 3: **Root causes across hardware types.** *The table shows the occurrences of the root causes across hardware types. The table is referenced in Section 3.1. The hardware types are SSD, disk, memory ("Mem"), network ("Net"), and processors ("CPU"). The internal root causes are device errors (ERR) and firmware issues (FW) and the external root causes are temperature (TEMP), power (PWR), environment (ENV), and configuration (CONF). Issues that are marked unknown (UNK) implies that the operators cannot pinpoint the root cause, but simply replaced the hardware. Note that a report can have multiple root causes (environment and power/temperature issues), thus the total (112) is larger than the 101 reports.*

Note that a report can have multiple root causes (environment and power/temperature issues), thus the total in Table 3 (112) is larger than the 101 reports.

3.2 Fault Conversions to Fail-Slow

Different types of faults such as fail-stop, -partial, and -transient can convert to fail-slow faults.

• **Fail-stop to fail-slow:** As many hardware pieces are connected together, a fail-stop component can make other components exhibit a fail-slow behavior. For example, a dead power supply throttled the CPUs by 50% as the backup supply did not deliver enough power; a single bad disk exhausted the entire RAID card's performance; and a vendor's buggy firmware made a batch of SSDs stop for seconds, disabling the flash cache layer and making the entire storage stack slow. These examples suggest that fail-slow occurrences can be correlated to other fail-stop faults in the system. Furthermore, a robust fail-stop tolerant system should ensure that fail-stop fault does not convert to fail-slow.

• **Fail-transient to fail-slow:** Besides fail-stop, many kinds of hardware can exhibit fail-transient errors, for example, disks occasionally return IO errors, processors sometimes produce a wrong result, and from time to time memory bits get corrupted. Due to their transient and "rare" nature, firmware/software typically masks these errors from users. A simple mechanism is to *retry* the operation or *repair* the error (*e.g.*, with ECC or parity). However, when the transient failures are recurring much more frequently, *error masking* can be a "*double-edged sword*." That is, because error masking is not a free operation (*e.g.*, retry delays, repair costs), when the errors are not rare, the masking overhead becomes the common case performance.

We observed many cases of fail-transient to fail-slow conversion. For example, a disk firmware triggered frequent "read-after-write" checks in a degraded disk; a machine was deemed nonfunctional due to heavy ECC correction of many DRAM bit-flips; a loose PCIe connection made the driver retry IOs multiple times; and many cases of loss/corrupt network packets (between 1-50% rate in our reports) triggered heavy retries that collapsed the network throughput by orders of magnitude.

From the stories above, it is clear that there must be a distinction between rare and frequent fail-transient faults. While it is acceptable to mask the former, the latter should be exposed to and not hidden from high-level software stack and monitoring tools.

• **Fail-partial to fail-slow:** Some hardware can also exhibit fail-partial fault where only some part of the device is unusable (*i.e.*, a partial fail-stop). This kind of failure is typically masked by the firmware/software layer (*e.g.*, with remapping). However, when the scale of partial failure grows, the fault masking brings a negative impact to performance. For example, in one deployment, the available memory size decreased over time increasing the cache miss rate, but did not cause the system to crash; bad chips in SSDs reduce the size of over-provisioned space, triggering more frequent garbage collection; and a more known problem, remapping of a large number of bad sectors can induce more disk seeks. Similar to the fail-transient case above, there must be a distinction of small- vs. large-scale partial faults.

3.3 Varying Fail-Slow Symptoms

We observed the "many faces" of fail-slow symptoms: permanent, transient, and partial fail-slow and transient fail-stop, as illustrated in Figure 1. Table 4 shows the breakdown of these failure modes across different hardware types. Table 5 shows the breakdown of these failure modes across different root causes.

• **Permanent slowdown:** The first symptom (Figure 1a) is a permanent slowdown, wherein the device initially worked normally but then its performance drops and does not return to the normal condition (until the problem is manually fixed). This mode is the simplest among the four models because operators can consistently see the issue. As shown in Table 4, this symptom (fortunately) is the most common one.

• **Transient slowdown:** The second one (Figure 1b) is a transient slowdown, wherein the device performance fluctuates between normal condition and signifi-

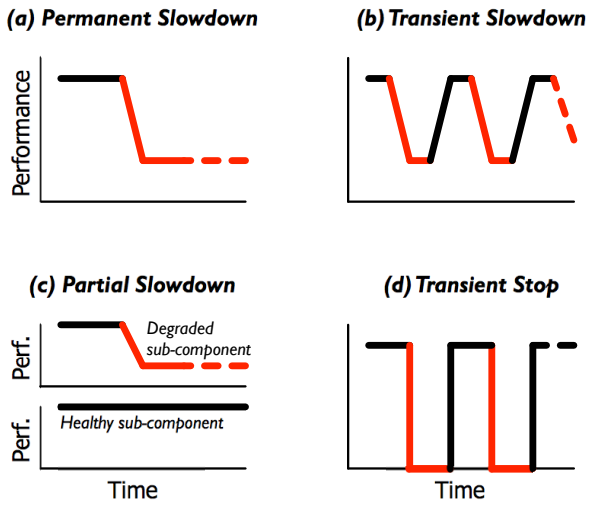


Figure 1: **Fail-slow symptoms.** *The figure shows four types of fail-slow symptom, as discussed in Section 3.3.*

cant degradation, which is more difficult to troubleshoot. For example, disk and network performance can degrade when the environment is too cold/hot, but will recover when the temperature is back to normal; occasional vibration when many disks were busy at the same time can reduce disk speed by orders of magnitude; and applications that create a massive load can cause the rack power control to deliver insufficient power to other machines (degrading their performance), but only until the power-hungry applications finish.

• **Partial slowdown:** The third model (Figure 1c) is partial slowdown, where only some parts of the device will exhibit slowdown. In other words, this is the case of partial fail-stop converting to partial slowdown (§3.2). For example, some parts of memory that are faulty require more ECC checks to be performed; some parts of network router's buffer that are corrupted will only cause the affected packets to be resent; and in one incident, 40% of big packets were lost, while none of small packets were lost. Partial fail-slow model also complicates debugging as some operations experience the slowdown but others (on the same device) are not affected.

• **Transient stop:** The last one (Figure 1d) is the case of transient stop, where the device occasionally reboots itself, thus there are times where the performance degrades to zero. For example, a buggy firmware made the SSDs sometimes "disappears" from RAID controller and later reappears; occasional bit flips in SAS/SCSI commands caused an host bus adapter to reboot repeatedly; and nodes automatically rebooted on thermal throttle (*e.g.*, when the fan firmware did not react quickly).

In one (hilarious) story, in the datacenter, there is a convenient table for staging, and one operator put an office chair adjacent to a storage cluster. The operator liked to rock in the chair, repeatedly popping hotplug drives out of the chassis (a hard correlation to diagnose).

	Symptoms			
HW Type	**Perm.**	**Trans.**	**Partial**	**Tr. Stop**
SSD	6	7	3	3
Disk	9	4	3	5
Mem	7	1	0	4
Net	21	0	5	2
CPU	10	6	1	3

Table 4: **Fail-slow symptoms across hardware types.** *The table depicts the occurrences of fail-slow symptoms across hardware types. The table is referenced in Section 3.3. The four symptoms "Perm.", "Trans.", "Partial", and "Tr. Stop" represent the four symptoms in Figure 1.*

	Symptoms			
Root	**Perm.**	**Trans.**	**Partial**	**Tr. Stop**
ERR	19	8	7	6
FW	11	3	1	4
TEMP	6	2	1	2
PWR	3	2	1	2
ENV	11	3	3	1
CONF	6	1	0	0
UNK	5	1	0	2

Table 5: **Fail-slow symptoms across root causes.** *The table is referenced in Section 3.3. The root-cause abbreviations can be found in the caption of Table 3. The four symptoms "Perm.", "Trans.", "Partial", and "Tr. Stop" represent the four symptoms in Figure 1.*

3.4 Cascading Causes and Impacts

Another intricacy of fail-slow hardware is the chain of cascading events: First, between the actual root cause and the hardware's fail-slow symptom, there is a chain of *cascading root causes*. Second, the fail-slow symptom then creates *cascading impacts* to the high-level software stack, and potentially to the entire cluster.

Below are some of the examples of long cascading root causes that lead to fail-slow hardware. A fan in a compute node stopped working, making other fans compensate the dead fan by operating at maximal speeds, which then caused a lot of noise and vibration that subsequently degraded the disk performance. A faulty sensor in a motherboard reported a false value to the OS making the CPUs run slower in energy saving mode. A lack of power from a broken power supply can cause many types of hardware, disks, processors, and network components to run sub-optimally. Power failure itself can also be caused by a long cascading causes, for example, the vendor omitted a 120V fuse that shipped with faulty

capacitors that have a high probability of shorting when power is cycled, which then caused minor electrical fires that cascade into rack-level power failures.

Next, when a hardware becomes fail-slow, not only it affects the host machine, but it can cause cascading impacts across the cluster. For example, a degraded NIC, from 1 Gbps to 1 Kbps, in one machine caused a chained reaction that slowed down the entire cluster of 100 machines (as the affected connecting tasks held up containers/slots for a long time, and new jobs cannot run due to slot shortage). In an HDFS HA (High Availability) deployment, a quorum of namenodes hang when one of the disks was extremely slow. In an HBase deployment, a memory card at 25% of normal speed caused backlogs, out-of-memory errors, and crashes. Similarly, a degraded disk created a backlog all the way to the client VMs, popping up the "blue screen of death" to users;

3.5 Rare but Deadly: Long TTD

The fail-slow hardware incidents in our report took *hours* or even *months* to detect (pinpoint). More specifically, 1% of the cases are detected in minutes, 13% in hours, 13% in days, 11% in weeks, and 17% in months (and unknown time in 45%). Some engineers called this a "costly debugging tail." In one story, an entire team of engineers were pulled to debug the problem, costing the institution tens of thousands of dollar. There are several reasons why the time-to-detect (TTD) is long.

First, the fact that the incidence of fail-slow hardware is not as frequent as fail-stop cases implies that today's software systems do not completely anticipate (*i.e.*, undermine) such scenarios. Thus, while more-frequent failures can be solved quickly, less-frequent but more complex failures (that cannot be mitigated by the system) can significantly cost the engineers time.

Second, as explained before, the root cause might not originate from the fail-slow hardware (*e.g.*, the case of transient slowdown caused by power-hungry applications in §3.3 took months to figure out as the problem was not rooted in the slow machines nor the power supply).

Third, external environment conditions beyond the control of the operators can prolong diagnosis (*e.g.*, for months, a vendor failed to reproduce the fail-slow symptoms in its sea-level testing facility as the hardware only slows down at a high mountain altitude).

Finally, operators do not always have full visibility of the entire hardware stack (*e.g.*, an incident took days to solve because the operators had no visibility into the power supply health).

4 Internal Root Causes

We now discuss internal root causes, primarily firmware bugs and device errors/wear-outs. We organize the discussion based on the hardware types (SSD, disk, memory, network, and processor).

4.1 SSD

Fail-slow SSDs can be triggered by firmware bugs and NAND flash management complexities.

Firmware bugs: We received three reports of SSD firmware bugs, admitted by the vendors. First, many individual IOs that should only take tens of µs were throttled by exactly multiples of 250µs, as high as 2-3ms. Even worse, in another report, a bad batch of SSDs stopped responding for seconds and then recovered. As mentioned before, an operator found some SSDs "disappeared" from the system and later reappeared. Upon vendor's inspection, the SSDs were performing some internal metadata writes that triggered hardware assertion failure and rebooted the device. In all these cases, the reasons why the firmware behaves as such were not explained (proprietary reasons). However, other incidents below might shed more light on the underlying problems.

Read retries with different voltages: In order to read a flash page, SSD controller must set a certain voltage threshold. As flash chips wear out, the charge in the oxide gates weakens, making the read operation with the default voltage threshold fail, forcing the controller to keep retrying the read with different voltage thresholds [10, 11]. We observed as high as 4 retries in the field.

RAIN/parity-based read reconstruction: Furthermore, if the data cannot be read (*i.e.*, is completely corrupted and fails the ECC checks), the SSD must reconstruct the page with RAIN (NAND-level RAID) [1, 41]. Three factors can make this situation worse. First, if the RAIN stripe width is N, $N-1$ additional reads must be generated to reconstruct the corrupt page. Second, the $N-1$ reads might also experience read retries as described above. Third, newer TLC-based SSDs use LDPC codes [40], which takes longer time to reconstruct the faulty pages. We observed that this reconstruction problem occurs frequently in devices nearing end of life. Moreover, SSD engineers found that the number of bit flips is a complex function of the time since the last write, the number of reads since the last write, the temperature of the flash, and the amount of wear on the flash.

Heavy GC in partially-failing SSD: Garbage collection (GC) of NAND flash pages is known to be a main culprit of user SLA violations [23, 28, 41]. However, in modern datacenter SSDs, the more advanced firmware successfully reduces GC impacts to users. In reality, there are SSDs shipped with "bad" chips. We witnessed that as more chips die, the size of the over-provisioned area gets reduced, which then triggers GC more frequently with impacts that cannot be hidden.

Broken parallelism by suboptimal wear-leveling: Ideally, large IOs are mapped to parallel channels/chips,

increasing IO parallelism. However, wear-leveling (the migration of hot/cold pages to hot/cold blocks) causes the mapping of LPN to PPN changes all the time. It has been observed that some rare workload behaviors can make wear-leveling algorithms suboptimal, making sequential LPNs mapped behind the same channels/chips (less parallelism). Furthermore, the problem of bad page/chip above also forces wear-leveling algorithms to make sub-optimal, less-parallel page/block mapping.

Hot temperature to wear-outs, repeated erases, and reduced space: Hot temperature can be attributed to external causes (§5.1), but can cause a chain reaction to SSD internals [31]. We also observed that SSD pages wear out faster with increasing temperature and there were instances of voltage threshold modeling that are not effective when SSDs operate at a higher temperature regime. As a result, after a block erase, the bits were not getting reset properly (not all bits become "1"). Consequently, some blocks had to be erased multiple times. Note that erase time is already long (*e.g.*, up to 6 ms), thus repeated erases resulted in observable fail-slow behavior. Worse, as some blocks cannot be reset properly after several tries, the firmware marked those blocks unusable, leading to reduced over-provisioned space, and subsequently more frequent GCs as discussed above.

Write amplification: Faster wear-outs and more frequent GCs can induce higher write amplification. It is worthy to report that we observed wildly different levels of amplification (*e.g.*, 5× for model "A", 600× for model "B", and "infinite" for certain workloads due to premature wear-outs).

Not all chips are created equal: In summary, most of the issues above originated with the fact that not all chips are created equal. Bad chips still pass vendor's testing, wherein each chip is given a quality value and high quality chips are mixed with lesser quality chips as long as the aggregate quality passes the quality-control standard. Thus, given an SSD, there are unequal qualities [10, 36]. Some workloads may cause more apparent wear-outs on the low quality chips, causing all the issues above.

4.2 Disk

Similar to SSDs, fail-slow disks can also be caused by firmware bugs and device errors/wear-outs.

Firmware bugs: We collected three reports related to disk firmware bugs causing slowdowns. There was a case where a disk controller delayed I/O requests for tens of seconds. In another problem, the disk "jitters" every few seconds, creating a problem that is hard to debug. In a large testbed, a RAID controller on the master node stalled, but then after restarted, the controller worked but with occasional timeouts and retries. Finally, there was an incident where a single bad disk exhausted the RAID card resources causing many IO timeouts (a failed case of bad-disk masking).

Device errors: Triggered by extensive disk rots, a RAID controller initiated frequent RAID rebuilding during run time; the fix reformatted the file systems so that bad sectors are collected and not used within the storage stack. Disk errors can be recurrent; in one case, disks with "bad" status were removed automatically from the storage pool but then added back when their status changed to "good," but the good-bad continuous transitions caused issues that affected user VMs. Some operators also observed media failures that forced the disks to retry every read operation multiple times before returning to the OS. A recent proposal advocates disks to automatically disable bad platters and continue working partially (with reduced bandwidth) [9].

Weak heads: This issue of disk "weak" heads is common in troubleshooting forums [17, 38], but the root cause is unclear. A report in our study stated that gunk that spills from actuator assembly and accumulates between the disk head and the platter can cause slow movement of the disk head. As disks are becoming "slimmer," the probability of trapped gunk increases. This problem can be fixed by performing random IOs to make the disk head "sweep the floor."

Other causes: Fail-slow disks can also be caused by environment conditions (*e.g.*, noises and vibrations from fans operating at the maximum speed) or temperature (*e.g.*, disks entering read-after-write mode in a colder environment [19]), which will be discussed later (§5).

4.3 Memory

Memory systems are considered quite robust, but we managed to collect a few evidence showing that memory hardware can also exhibit fail-slow faults.

Device errors: In cases of partial memory errors, there were reports of custom chips masking the errors and not exposing bad addresses. Here, as more errors increase over time, the available memory size decreases, causing higher cache misses. Unlike disk/SSD usage where out-of-space error is thrown when space runs out, memory usage is different; as long as the minimum memory space requirement is met, applications can still run albeit with slower performance due to more frequent page swapping from the reduced cache size.

External causes: There were two cases of memory cards slowing down due to the environment condition (specifically a high altitude deployment that introduces more cosmic events that cause frequent multi-bit upsets) and human mistakes (an operator plugged in a new NVDIMM card in a rush and the loose connection made the card still functional, but with slower performance).

Unknown causes: There were other fail-slow memory incidents with unknown causes. In an HBase deploy-

ment, a memory card ran only 25% of normal speed. In another non-deterministic case, low memory bandwidth was observed under a certain benchmark, but not under different benchmarks.

SRAM errors: Much attention is paid to DRAM errors [37] and arguably DRAM reliability is largely a solved problem – most errors can be masked by ECC (by sacrificing predictable latency) or lead to fail-stop behavior of the impacted program. Besides DRAM, SRAM usage is pervasive in device controllers (*e.g.*, FPGAs, network cards, and storage adapters). Unlike DRAM, SRAM works by constantly holding the voltage of each memory cell at the desired level; it does not incorporate refresh cycles that can cause read/write to stall. It is most commonly used by circuits that cannot afford to incur stalls or buffer data between RAM and the combinatorial logic that consumes the data.

SRAM errors on data paths are typically transparently masked; they ultimately lead to a CRC validation error, and the network packet or disk I/O is simply retried. However, SRAM is also incorporated in *control* paths. We observed SRAM errors that caused *occasional reboots* of the device from broken control path (among many other problems), inducing a transient-stop symptom (as discussed in §3.3). SRAM per-bit error rates unfortunately have not improved [8]. Therefore in practice, SRAM errors are a regular occurrence in large-scale infrastructure, a major culprit of service disruptions.

4.4 Network

Network performance variability is a well-known problem, typically caused by load fluctuations. This paper highlights that fail-slow networking hardware can be a major cause of network performance degradation.

Firmware bugs: We collected three reports of "bad" routing algorithms in switch firmware. In one case, the network performance decreased to half of the maximum performance due to a dynamic routing algorithm on stock driver/firmware that did not work "as promised [by the vendor]." Due to lack of visibility to what is happening in the firmware, the operators must hack the kernel to perform ping between the switches, which consumed a long time. In another story, MAC learning was not being responsive and special types of traffic such as multicast were not working well, creating traffic floods. The third story is similar to the first one.

NIC driver bugs: Four instances of NIC driver bugs were reported, dropping many packets and collapsing TCP performance. In one story, 5% package loss caused many VMs to go into "blue screen of death." Another NIC driver bug caused a "very poor" throughput and the operators had to disable TCP offload to work around the problem. In another case, the developers found a non-deterministic network driver bug in Linux that only surfaced on one machine, making the 1 Gbps NIC card transmit only at 1 Kbps. Finally, a bug caused an unexpected auto-negotiation between a NIC and a TOR switch that capped the bandwidth between them, underutilizing the available bandwidth.

Device errors: In one interesting story, the physical implementation of the network cards did not match the design specification – there is a distant corner of the chip that is starving from electrons and not performing at full speed; the vendor re-manufactured all the network cards, a very costly ramification. Similarly, a bad VSCEL laser degraded switch to switch performance; this bad design affected hundreds of cables. In one deployment, a router's internal buffer memory was introducing occasional bit errors into packets, causing failed end-to-end checksums and subsequently TCP retries.

External causes: Some fail-slow networking components were also caused by environment conditions (*e.g.*, loose network cables, pinched fiber optics), configuration issues (*e.g.*, a switch environment not supporting jumbo frames such that MTU size must be configured to 1500 bytes), and temperature (*e.g.*, clogged air filter, bad motherboard design that puts NIC behind CPU).

Unknown causes: There are other reports of throughput degradation at the hardware level or severe loss rates without known root causes. For example, a 7 Gbps fibre channel collapsed to 2 Kbps, a 1 Gbps throughput degraded to 150 Mbps with just 1% loss rate, 40% of big packets were lost (but no small-package loss), and some observed error/loss rates as high as 50%. TCP performance is highly sensitive to loss rate.

4.5 Processor

We find processors are quite reliable and do not self-inflict fail-slow mode. Most of the fail-slow CPUs are caused by external factors, which we briefly discuss below, but will be detailed in the next section (§5).

External causes: We observed fail-slow processors caused by configuration mistakes (*e.g.*, a buggy BIOS firmware incorrectly down-clocked the CPUs), environment conditions (*e.g.*, a high-altitude deployment made the CPUs enter thermal throttle), temperature issues (*e.g.*, CPU heat-sinks were not in physical contact with the CPUs, a fan firmware did not react quickly to cool down the CPUs), and power shortage (*e.g.*, insufficient capacitors in the motherboard's power control logic did not deliver enough power when the load is high).

5 External Root Causes

We now describe external root causes of fail-slow hardware such as temperature variance, power shortage, environment condition, and configuration mistakes. These external causes complicate troubleshooting because the

symptoms can be non-deterministic and only reproducible in the same online scenario, but not observable in offline (in-office) testing.

5.1 Temperature

To keep temperature in normal operating condition, fans or heat-sinks must work correctly. Below are root causes of temperature variance that went undetected by the monitoring tools.

Clogged air filter: In one report, a clogged air filter caused optics in the switch to start failing due to a high temperature, generating a high 10% packet loss rate. After the air filter was cleaned, the switch returned to normal speed but only temporarily. It is likely that the high temperature had broken the switch's internal parts.

Cold environment: Cold temperature can induce fail-slow faults as well [19]. In one deployment, some of the disks went into read-after-write mode. Upon inspection, the machine room had a "cold-air-under-the-floor" system, which was more common in the past. The disks at the bottom of the racks had a higher incidence of slow performance. This suggests that temperature variance can originate from deployment environment as well.

Broken fans: Cooling systems such as fans sometimes work as a cluster, rather than individually. There was a case where a fan in a compute node stopped working, and to compensate this failing fan, fans in other compute nodes started to operate at their maximal speed, which then generated heavy noise and vibration that degraded the disk performance. Again, this is an example of cascading root causes (§3.4).

Buggy fan firmware: Fans can be fully functional, but their speeds are controlled by the fan firmware. In one condition, a fan firmware would not react quickly enough when CPU-intensive jobs were running, and as a result the CPUs entered thermal throttle (reduced speed) before the fans had the chance to cool down the CPUs.

Improper design/assembly/operation: A custom motherboard was "badly" designed in such a way that the NIC was soldered on the motherboard behind the CPU and memory. The heat from the CPU affected the NIC causing many packet errors and retries. In a related story, due to bad assembly, CPU heat-sinks were not in physical contact with the CPUs, causing many nodes to overheat. In another case, new disks were plugged into machines with "very old" fans. The fans did not give enough cooling for the newer disks, causing the disks to run slowly.

5.2 Power

Reduced power can easily trigger fail-slow hardware. Below are some of the root causes of power shortage.

Insufficient capacitors: In one custom motherboard design, the capacitor on the motherboard's power control logic did not provide adequate voltage to the CPUs under certain load. This put the processors out of specification, causing corruptions and recomputations. The diagnosis time was months due to the fact that the problem could not be reliably reproduced. To fix the problem, a small capacitor was added to each motherboard on site for thousands of nodes. In a similar story, an inadequate capacitor caused voltage drop, but only when multiple cores transition from parked to turbo-boost simultaneously (a corner-case situation). Thus, independent testing of the updated BIOS and software did not reproduce the issue.

PCU firmware bugs: In one scenario, the firmware of the power control units (PCUs) entered a "weird" state and did not deliver enough power, and the whole rack failed off the power control. This was a transient fault that sometimes can be fixed by resetting the controller, sometimes by re-flashing the firmware, and in rare instances, by replacing the PCUs.

Fail-partial power supply: In one deployment, every four machines share two power supplies. However, when one power supply failed, there was not enough power to run all the four machines at normal capacity, thus throttling the CPUs on each machine by 50%. The problem cascaded as the machines were used for indexing service and could not keep up with the number of requests. The problem took days to solve because the operators had no visibility into the power supply health. This problem is also interesting as two power supplies do not imply that one of them is a full-working backup, but rather a reduced power, enough to keep the machines alive.

Power-hungry neighbors: Some nodes were running slow because other nodes in the same rack were drawing more power, causing the rack power supply to go unstable, and dropping power to various parts of the rack. It took months to diagnose the problem as it was not rooted in the slow machines and only happened when power-hungry applications were running in neighboring nodes.

Faulty motherboard sensors: After a long period of debugging a slow machine, the operator discovered that the motherboard had a faulty sensor that reported faulty value to the OS, making the OS configure the CPUs to run in slower speed in energy saving mode.

5.3 Environment

Fail-slow hardware can be induced by a variety of environment conditions, as listed below.

Altitude and cosmic events: One of the most interesting reports we collected is from a deployment at altitude of 7500 feet. At this height, some CPUs would become hot and enter thermal throttle (reduced performance). Apparently, the fault was not in the CPUs, but

rather in the vendor's cooling design that was not providing enough cooling at such a high altitude. In another report, still at the same altitude, some memory systems experienced more frequent multi-bit upsets than usual (increased ECC checks and repairs), which then were shipped back to the vendor and re-assembled with more memory protection.

Loose interconnects: Loose network cables and pinched fiber optics caused network delays up to hundreds of milliseconds, making the storage cluster behave abnormally. It took several days to diagnose the problem, as the symptom was not deterministic. The reason behind loose/pinched cables can be vibration or human factor. In some other cases, loose PCIe connections between the SSDs and the PCIe slots made the device driver layer retry the operations multiple times. In another story, an NVDIMM was not plugged in properly when the operator was rushed in fixing the machine. The machine was still functional albeit with a much lower speed.

Vibrations: The performance of some disk drives collapsed to 100 KB/s when deployed in the racks, but performed maximally 100 MB/s when tested in office. Apparently, faulty chassis fans surrounding the nodes caused such a strong vibration, making the drives go into recovery mode. The solution was to add vibration dampers to each of the eight hard drive screws and replace roughly 10% system fans in all nodes.

Environment and operating condition mismatch: In one institution, a system was configured correctly at the advertised clock rate, temperature range, and voltage range. However, due to an unknown environment condition, it was not working optimally, and the solution was turning down the clock slightly, putting a software monitor on processor temperature and voltage, and killing the node if voltage/temperature got close to the edge of the binned values (*i.e.*, a dead node is better than a slow node). Time to diagnose was months due to not reliably able to reproduce. In another case, a switch environment did not support "jumbo frames" and caused the 10 Gbps throughput network to have a poor throughput. The fix was to reconfigure the MTU size to be 1500 bytes.

Unknown causes: In one interesting report, billions of SAS errors simultaneously reported by all the independent drives in the cluster, lasting for five minutes. The report stated that this happened when a technician was performing maintenance on another machine.

5.4 Configuration

While hardware typically runs in default configuration, today's hardware has "knobs" that allow configurable parameters. Such configurations can be modified by human operators or software/firmware layers (*e.g.*, BIOS). In our findings, fail-slow hardware can be induced by the following misconfiguration mistakes.

Buggy BIOS firmware: In one institution, one of the systems typically ingested 2.8 billion metrics per minute, however at one time the metric write time increased, taking more than a minute to process all the metrics from previous minutes. The operators added more nodes (thinking that it will load balance the request spikes). Counter-intuitively, adding more nodes resulted in increased write time. The diagnosis spanned a month. The root cause was the BIOS was incorrectly downclocking the CPUs of the new machines being added to the database cluster. These machines were "limping" along but were assigned the same number of load (as if a correctly clocked machine). Similarly, as reported elsewhere [16, §3.6], a buggy initialization configuration can also disable the processor caches.

Human mistakes: Regarding SSD connections, not all PCIe slots have the same number of lanes. Mistakes in mapping PCIe cards to PCIe slots with different number of lanes had occasionally been made by human operators, which results in under-utilization of full connection bandwidth. In a different case, an incorrect parameter set in `xtnird.ini`, a network configuration that manages High Speed Networking (HSN) over InfiniBand, was not set up properly and the network was throttling. There is plethora of related work on configuration mistakes [5, 42]. We believe there are many more instances of configuration mistakes that trigger fail-slow hardware, not recorded in production logs.

6 Suggestions

In addition to cataloguing instances of fail-slow hardware, a goal of this paper is to offer vendors, operators and systems designers insights about how to address this poorly-studied failure mode.

6.1 To Vendors

• **Making implicit error masking explicit:** Fail-slow hardware can be categorized as an "implicit" fault, meaning they do not always return any explicit hard errors, for example due to error masking (§3.2). However, there were many cases of slowly increasing error rates that would eventually cause cascading performance failures. Although statistics of error rates are obtainable from the device (*e.g.*, number of ECC repairs, corrupt packets), they are rarely monitored by the overall system. Vendors might consider throwing explicit error signals when the error rates far exceed the expected rate.

We understand that this could be a far-from-reach reality because vendors often hide internal statistics (*e.g.*, most recent SSDs no longer expose the number of internal writes, as some users were upset to learn about the write amplification). In fact, the trend of moving to

white-box storage makes the situation worse. That is, black-box storage such as commodity disks and SSDs conform to some standards (*e.g.*, S.M.A.R.T data), however as more institutions now compose the entire hardware/software storage stack (*e.g.*, fully host-managed flash), the hardware designers might not conform to existing standards, making software-level error management more difficult.

• **Exposing device-level performance statistics:** Two decades ago, statistical data of hard errors was hard to obtain, but due to user demands, modern hardware now exposes such information (*e.g.*, via S.M.A.R.T), which then spurred many statistical studies of hardware failures [6, 7, 30, 34, 35, 36] However, the situation for hardware-level performance studies is bleak. Our conversations with operators suggest that the information from S.M.A.R.T is "insufficient to act on." In some institutions, hardware-level performance logs are only collected hourly, and we could not pinpoint whether a slow performance was due to the workload or the device degradation. With these limitations, many important statistical questions are left unanswered (*e.g.*, how often fail-slow hardware occurs, how much performance was degraded, what correlations fail-slow faults exhibit with other metrics such as device age, model, size, and vendor). We hope vendors will expose device-level performance data to support future statistical studies.

6.2 To Operators

• **Online diagnosis:** In our study, 39% of the cases were caused by external root causes, which suggests that blames cannot be directed towards the main hardware components. Some reports suggest that operators took days or even months to diagnose, as the problems cannot be reproduced in offline ("in-office") testing. Thus, online diagnosis is important, but also not straightforward because not all hardware components are typically monitored, which we discuss next.

• **Monitoring of all hardware components:** Today, in addition to main hardware components (*e.g.*, disks, NICs, switches, CPUs), other hardware components and environment conditions such as fan speeds and temperature are also monitored. Unfortunately, not all hardware is monitored in practice. For example, multiple organizations failed to monitor network cables, and instead used the flow of traffic as a proxy for cable health. The diagnosis took much longer time because performance blames are usually directed towards the main hardware components such as NICs or switches. The challenge is then to prevent too much data being logged.

Another operational challenge is that different teams are responsible for different parts of the data center (*e.g.*, software behavior, machine performance, cooling, power). Thus, with limited views, operators cannot fully diagnose the problem. In one incident, the operators, who did not have access to power supply health, took days to diagnose the reason behind the CPUs running only at 50% speed. In another example, power supply health information was available, but basic precautions, such as adding fuses to the input line, were overlooked.

Another challenge to come is related to proprietary full-packaged solution like hyper-converged or rack-scale design. Such design usually comes with the vendor's monitoring tools, which might not monitor and expose all information to the operators. Instead, vendors of such systems often monitor hardware health remotely, which can lead to fragmentation of monitoring infrastructure as the number of vendors increases.

• **Correlating full-stack information:** With full-stack performance data, operators can use statistical approaches to pinpoint and isolate the root cause [15].

Although most of the cases in our study were hard-to-diagnose problems, fortunately, the revealed root causes were relatively "simple." For example, when a power-hungry application was running, it drained the rack power and degraded other nodes. Such a correlation can be easily made, but requires process- to power-level information. As another example, when a fan stopped, and to compensate, the other fans ran in maximum speed to compensate, the resulting vibration degraded disk performance. This 3-level correlation between fan status, vibration level, and disk performance can also be correlated. Future research can be done to evaluate whether existing statistical monitoring approaches can detect such correlations.

While the metrics above are easy to monitor, there are other fine-grained metrics that are hard to correlate. For example, in one configuration issue, only multicast network traffic was affected, and in another similar one, only big packets (>1500 bytes) experienced long latencies. In these examples, the contrast between multicast and unicast traffics and small and big packets is clear. However, to make the correlation, detailed packet characteristics must be logged as well.

Finally, monitoring algorithms should also detect "counter-intuitive" correlations. For example, when users performance degrades, operators tend to react by adding more nodes. However, there were cases where adding more nodes did not translate to better performance, as the underlying root cause was not isolated.

6.3 To Systems Designers

While the previous section focuses on post-mortem remedies, this section provides some suggestions on how to anticipate fail-slow hardware better in future systems.

• **Making implicit error-masking explicit:** Similar to the error masking problem at the hardware level, error masking (as well as "tail" masking) in higher software stack can make the problem worse. We have observed fail-slow hardware that caused many jobs to timeout and be retried again repeatedly, consuming many other resources and converting the single hardware problem into larger cluster-wide failures. Software systems should not just silently work around fail-slow hardware, but need to expose enough information to help troubleshooting.

• **Fail-slow to fail-stop:** Earlier, we discussed about many fault conversions to fail-slow faults (§3.2). The reverse can be asked: can fail-slow faults be converted into fail-stop mode? Such a concept is appealing because modern systems are well equipped to handle fail-stop failures [12]. Below we discuss opportunities and challenges of this concept.

Skip non-primary fail-slow components: Some resources such as (*e.g.*, caching layers) can be considered non-primary components. For example, in many deployments, SSDs are treated as a caching layer for the back-end disks. The assumption that SSD is always fast and never stalls does not always hold (§4.1). Thus, when fail-slow SSDs (acting as a caching layer) introduce more latencies than the back-end disks, they can be skipped temporarily until the problem subsides. However, consistency issues must be taken into account. In one story, the operators had to disable the flash cache layer for one month until the firmware was fixed. Another suggestion is to run in "partial" mode rather than in full mode but with slow performance. For example, if many disks cause heavy vibration that degrades the disk throughput significantly, it is better to run fewer disks to eliminate the throughput-degrading vibration [13].

Detect fail-slow recurrences: Another method to make slow-to-stop conversion is to monitor the recurrence of fail-slow faults. For example, disks/SSDs that continue to "flip-flop" in online/offline mode (§4.1), triggering RAID rebalancing all the time, is better to be put offline. As another example, if I/O communication to a hardware requires many retries, the device perhaps can be removed. We observed several cases of transient fail-slow hardware that was taken offline but after passing the in-office diagnosis, the device was put online again, only to cause the same problem.

Challenges: While the concept of slow-to-stop conversion looks simple, there are many challenges that impedes its practicality in the field, which we hope can trigger more research in the community. First, an automated shutdown algorithm should be robust (no bugs or false positives) such that healthy devices are not incorrectly shut down. Second, some storage devices cannot be abruptly taken offline as it can cause excessive replication load. Third, similarly, removing slow nodes can risk availability; in one deployment, some machines exhibited 10-20% performance degradation but if they were taken out, availability would be reduced, and data loss could ensue. Fourth, a node is an expensive resource (*e.g.*, with multiple NICs, CPUs, memory cards, SSDs, disks), thus there is a need for capability to shut off devices at fine-grained level. Fifth, and more importantly, due to the cascading nature (§3.4), fail-slow hardware can be induced by external factors; here, the solution is to isolate the external factors, not to shutdown the slow device.

• **Fail-Slow fault injections:** System architects can inject fail-slow root causes reported in this paper to their systems and analyze the impacts.

For example, one can argue that asynchronous distributed systems (*e.g.*, eventual consistency) should naturally tolerate fail-slow behaviors. While this is true, there are many stateful systems that cannot work in fully asynchronous mode; for example, in widely-used open-sourced distributed systems, fail-slow hardware can cause cascading failures such as thread pool exhaustion, message backlogs, and out-of-memory errors [17].

Another type of systems is tail-tolerant distributed systems [16]. However, another recent work shows that the "tail" concept only targets performance degradation from resource contention, which is different than fail-slow hardware model such as slow NICs, and as a result not all tail-tolerant systems (*e.g.*, Hadoop, Spark) can cut tail latencies induced by degraded NICs [39].

Beyond networking components, the assumption that storage latency is stable is also fatal. It has been reported that disk delays causes race condition or deadlock in distributed consistency protocols [29]. The problem is that some consistency protocols, while tolerating network delays, do not incorporate the possibility of disk delays, for the sake of simplicity.

With fail-slow injections, operators can also evaluate whether their systems or monitoring tools signal the right warnings or errors. There were a few cases in our reports, where wrong signals were sent, causing the operators to debug only the healthy part of the system.

Overall, we strongly believe that injecting root causes reported in this paper will reveal many flaws in existing systems. Furthermore, all forms of fail-slow hardware such as slow NICs, switches, disks, SSD, NVDIMM, and CPUs need to be exercised as they lead to different symptoms. The challenge is then to build future systems that enable various fail-slow behaviors to be injected easily.

7 Discussions

• **Limitations (and Failed Attempts):** We acknowledge the major limitation of our methodology: the lack of

quantitative analysis. Given the reports in the form of anecdotes, we were not able to answer statistical questions such as how often fail-slow hardware occurs, how much performance was degraded, what correlations fail-slow faults exhibit with other metrics such as device age, model, size, and vendor, etc.

We initially had attempted to perform a quantitative study. However, many institutions do not maintain a database of hardware-level performance data. Many institutions that we asked to join in this community paper responded with either "we do not have clearance" or "we do not collect such data (but have unformatted reports)." In the former category (no clearance), it is inconclusive whether they have such data available or the nature of this public study was not allowed in the first place.

An institution told us that they collect large performance data at the software level, but direct inference to fail-slow hardware is challenging to perform. In our prior work, we only collected hourly aggregate of disk/SSD-level performance data [24], but the coarse hourly granularity has limitations and the findings cannot be directly tied to "hard proof" of the existence of fail-slow hardware.

We also managed to obtain ticket logs (in unformatted text) from a large institution, but searching for fail-slow hardware instances in tens of thousands of tickets is extremely challenging as the operators did not log the full information and there is no standard term for "fail-slow/limping/jittery" hardware. For example, searching for the word "slow" produces hundreds of results that do not directly involve hardware issues.

Indeed, we believe that the lack of easily accessible and analyzable data is a reason that the study in this paper is valuable. Regardless of the limitation of our study, we believe we have successfully presented the most complete account of fail-slow hardware in production systems that can benefit the community.

• **"Honorable Mentions":** While this paper focuses on fail-slow faults, our operators shared to us many other interesting anecdotes related to data loss, which we believe are "honorable" to mention as the details were rarely mentioned in literature.

Triple replication is (sometimes) not enough: In one large Hadoop cluster, many machines were failing regularly such that data loss was unavoidable even with triple replication. Apparently, this was caused by a large batch of malfunctioning SSDs. The controller on this brand of SSDs was "bad" and would stop responding. About 3-5% of the drives would be failing each week. Worse, the servers would not shut down properly because the shutdown required a successful write to the SSD to do so. Thus, there were lower success rates because broken machines with failed SSDs would try to serve traffic and could not shut themselves down.

Single point of failure (in unseen parts): While at a high level, datacenter operators ensures that there is no single hardware failure (redundant machines, power, cooling, etc.), there was a case of redundant EEPROMS that rely on *single* capacitor (a part that was unobservable by the operators and only known by the vendor). Unfortunately, the capacitor failed and triggered correlated failures on both SAS paths, causing a complete 24-hour outage in production.

In a related story, a healthy-looking system was actually miscabled, without apparent performance issues, but the miscabling led to multiple single points of failure. There was no cable topology monitoring, thus the technicians had to devise recabling strategies that maintain the expected redundancy level.

Failed NVRAM dump under power fault: To handle write idiosyncrasies of NAND flash, writes are "persisted" to NVRAM (capacitor-backed RAM) with the promise that under a power fault the content of the RAM should be flushed ("dumped") to the non-volatile NAND flash. However, there was a non-deterministic case where in 1 out of 10,000 power losses, the firmware did not trigger the NVRAM dump. Apparently, the FPGA design assumed a pin was grounded, but the pin was attached to a test pad instead, and the RFI led to propagation of "nonsense" from the pin into the NVRAM dump logic. More studies of SSD robustness under power fault are needed.

8 Conclusion

Today's software systems are arguably robust at logging and recovering from fail-stop hardware – there is a clear, binary signal that is fairly easy to recognize a and interpret. We believe fail-slow hardware is a fundamentally harder problem to solve. It is very hard to distinguish such cases from ones that are caused by software performance issues. It is also evident that many modern, advanced deployed systems do not anticipate this failure mode. We hope that our study can influence vendors, operators, and systems designers to treat fail-slow hardware as a separate class of failures and start addressing them more robustly in future systems.

9 Acknowledgments

We thank Dean Hildebrand, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We also would like to thank Tracy Carver for contributing anecdotes and Jeffrey Heller for his support of this work. This material was supported by funding from NSF (grant Nos. CCF-1336580, CNS-1350499, CNS-1526304, and CNS-1563956).

References

[1] NAND Flash Media Management Through RAIN. Micron, 2011.

[2] http://ucare.cs.uchicago.edu/projects/failslow/, 2018.

[3] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, 2001.

[5] Mona Attariyan and Jason Flinn. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[6] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.

[7] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[8] Robert C. Baumann. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. *IEEE Transactions on Device and Materials Reliability (TDMR)*, 5(3), September 2005.

[9] Eric Brewer. Spinning Disks and Their Cloudy Future (Keynote). In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[10] Yu Cai, Yixin Luo, Saugata Ghose, and Onur Mutlu. Read Disturb Errors in MLC NAND Flash Memory: Characterization and Mitigation. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2015.

[11] Yu Cai, Yixin Luo, Erich F. Haratsch, Ken Mai, and Onur Mutlu. Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA-21)*, 2015.

[12] George Candea and Armando Fox. Crash-Only Software. In *The Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[13] Christine S. Chan, Boxiang Pan, Kenny Gross, Kenny Gross, and Tajana Simunic Rosing. Correcting vibration-induced performance degradation in enterprise servers. In *The Greenmetrics workshop (Greenmetrics)*, 2013.

[14] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[15] Daniel J. Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[16] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[17] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[18] 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 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[19] Nosayba El-Sayed, Ioan A. Stefanovici, George Amvrosiadis, Andy A. Hwang, and Bianca Schroeder. Temperature Management in Data Centers: Why Some (Might) Like It Hot. In *Proceedings of the 2012 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

[20] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[21] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[22] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages.

In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[23] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

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

[25] Peng Huang, Chuanxiong Guo, Lindong Znhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randonph Yao. Gray Failure: The Achilles' Heel of Cloud Scale Systems. In *The 16th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, 2017.

[26] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating Hardware Device Failures in Software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[27] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-Box Problem Diagnosis in Parallel File Systems. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

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

[29] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[30] Ao Ma, Fred Douglis, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[31] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2015.

[32] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[33] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[34] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[35] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

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

[37] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *Proceedings of the 2009 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2009.

[38] Brian D. Strom, SungChang Lee, George W. Tyndall, and Andrei Khurshudov. Hard Disk Drive Reliability Modeling and Failure Prediction. *IEEE Transactions on Magnetics (TMAG)*, 43(9), September 2007.

[39] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Uma Maheswara Rao G., and Haryadi S. Gunawi. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[40] Eitan Yaakobi, Laura Grupp, Paul H. Siegel, Steven Swanson, and Jack K. Wolf. Characterization and Error-Correcting Codes for TLC Flash Memories. In *International Conference on Computing, Networking and Communications (ICNC)*, 2012.

[41] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[42] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

Protocol-Aware Recovery for Consensus-Based Storage

Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee[†], Aws Albarghouthi,
Vijay Chidambaram[†], Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

University of Wisconsin – Madison [†] *University of Texas at Austin*

Abstract

We introduce *protocol-aware recovery* (PAR), a new approach that exploits protocol-specific knowledge to correctly recover from storage faults in distributed systems. We demonstrate the efficacy of PAR through the design and implementation of corruption-tolerant replication (CTRL), a PAR mechanism specific to replicated state machine (RSM) systems. We experimentally show that the CTRL versions of two systems, LogCabin and ZooKeeper, safely recover from storage faults and provide high availability, while the unmodified versions can lose data or become unavailable. We also show that the CTRL versions have little performance overhead.

1 Introduction

Failure recovery using redundancy is central to improved reliability of distributed systems [14, 22, 31, 35, 61, 67]. Distributed systems recover from node crashes and network failures using copies of data and functionality on several nodes [6,47,55]. Similarly, bad or corrupted data on one node should be recovered from redundant copies.

In a *static* setting where all nodes always remain reachable and where clients do not actively update data, recovering corrupted data from replicas is straightforward; in such a setting, a node could repair its state by simply fetching the data from any other node.

In reality, however, a distributed system is a *dynamic* environment, constantly in a state of flux. In such settings, orchestrating recovery correctly is surprisingly hard. As a simple example, consider a quorum-based system, in which a piece of data is corrupted on one node. When the node tries to recover its data, some nodes may fail and be unreachable, some nodes may have recently recovered from a failure and so lack the required data or hold a stale version. If enough care is not exercised, the node could "fix" its data from a stale node, overwriting the new data, potentially leading to a data loss.

To correctly recover corrupted data from redundant copies in a distributed system, we propose that a recovery approach should be *protocol-aware*. A *protocol-aware recovery* (PAR) approach is carefully designed based on how the distributed system performs updates to its replicated data, elects the leader, etc. For instance, in the previous example, a PAR mechanism would realize that a faulty node has to query at least R (read quorum) other nodes to safely and quickly recover its data.

In this paper, we apply PAR to replicated state machine (RSM) systems. We focus on RSM systems for two reasons. First, correctly implementing recovery is most challenging for RSM systems because of the strong consistency and durability guarantees they provide [58]; a small misstep in recovery could violate the guarantees. Second, the reliability of RSM systems is crucial: many systems entrust RSM systems with their critical data [45]. For example, Bigtable, GFS, and other systems [7,26] store their metadata on RSM systems such as Chubby [16] or ZooKeeper [4]. Hence, protecting RSM systems from storage faults such as data corruption will improve the reliability of many dependent systems.

We first characterize the different approaches to handling storage faults by developing the *RSM recovery taxonomy*, through experimental and qualitative analysis of practical systems and methods proposed by prior research (§2). Our analyses show that most approaches employed by currently deployed systems do not use any protocol-level knowledge to perform recovery, leading to disastrous outcomes such as data loss and unavailability.

Thus, to improve the resiliency of RSM systems to storage faults, we design a new protocol-aware recovery approach that we call corruption-tolerant replication or CTRL (§3). CTRL constitutes two components: a *local storage layer* and a *distributed recovery protocol*; while the storage layer reliably detects faults, the distributed protocol recovers faulty data from redundant copies. Both the components carefully exploit RSM-specific knowledge to ensure safety (e.g., no data loss) and high availability.

CTRL applies several novel techniques to achieve safety and high availability. For example, a *crash-corruption disentanglement* technique in the storage layer distinguishes corruptions caused by crashes from disk faults; without this technique, safety violations or unavailability could result. Next, a *global-commitment determination* protocol in the distributed recovery separates committed items from uncommitted ones; this separation is critical: while recovering faulty committed items is necessary for safety, discarding uncommitted items quickly is crucial for availability. Finally, a novel *leader-initiated snapshotting* mechanism enables identical snapshots across nodes to greatly simplify recovery.

We implement CTRL in two storage systems that are based on different consensus algorithms: LogCabin [43]

(based on Raft [50]) and ZooKeeper [4] (based on ZAB [39]) (§4). Through experiments, we show that CTRL versions provide safety and high availability in the presence of storage faults, while the original systems remain unsafe or unavailable in many cases; we also show that CTRL induces minimal performance overhead (§5).

2 Background and Motivation

We first provide background on storage faults and RSM systems. We then present the taxonomy of different approaches to handling storage faults in RSM systems.

2.1 Storage Faults in Distributed Systems

Disks and flash devices exhibit a subtle and complex failure model: a few blocks of data could become inaccessible or be silently corrupted [8, 9, 32, 59]. Although such storage faults are rare compared to whole-machine failures, in large-scale distributed systems, even rare failures become prevalent [60, 62]. Thus, it is critical to reliably detect and recover from storage faults.

Storage faults occur due to several reasons: media errors [10], program/read disturbance [60], and bugs in firmware [9], device drivers [66], and file systems [27, 28]. Storage faults manifest in two ways: block *errors* and *corruption*. Block errors (or latent sector errors) arise when the device internally detects a problem with a block and throws an error upon access. Studies of both flash [33, 60] and hard drives [10, 59] show that block errors are common. Corruption could occur due to lost and misdirected writes that may not be detected by the device. Studies [9, 51] and anecdotal evidence [36, 37, 57] show the prevalence of data corruption in the real world.

Many local file systems, on encountering a storage fault, simply propagate the fault to applications [11, 54, 64]. For example, ext4 silently returns corrupted data if the underlying device block is corrupted. In contrast, a few file systems transform an underlying fault into a different one; for example, btrfs returns an error to applications if the accessed block is corrupted on the device. In either case, storage systems built atop local file systems should handle corrupted data and storage errors to preserve end-to-end data integrity.

One way to tackle storage faults is to use RAID-like storage to maintain multiple copies of data on each node. However, many distributed deployments would like to use inexpensive disks [22, 31]. Given that the data in a distributed system is inherently replicated, it is wasteful to store multiple copies on each node. Hence, it is important for distributed systems to use the inherent redundancy to recover from storage faults.

2.2 RSM-based Storage Systems

Our goal is to harden RSM systems to storage faults. In an RSM system, a set of nodes compute identical states by executing commands on a state machine (an in-memory data structure on each node) [58]. Typically, clients interact with a single node (the leader) to execute operations on the state machine. Upon receiving a command, the leader durably writes the command to an on-disk *log* and replicates it to the followers. When a majority of nodes have durably persisted the command in their logs, the leader applies the command to its state machine and returns the result to the client; at this point, the command is committed. The commands in the log have to be applied to the state machine *in-order*. Losing or overwriting committed commands violates the safety property of the state machine. The replicated log is kept consistent across nodes by a consensus protocol such as Paxos [41] or Raft [50].

Because the log can grow indefinitely and exhaust disk space, periodically, a *snapshot* of the in-memory state machine is written to disk and the log is garbage collected. When a node restarts after a crash, it restores the system state by reading the latest on-disk snapshot and the log. The node also recovers its critical metadata (e.g., log start index) from a structure called *metainfo*. Thus, each node maintains three critical persistent data structures: the *log*, the *snapshots*, and the *metainfo*.

These persistent data structures could be corrupted due to storage faults. Practical systems try to safely recover the data and remain available under such failures [15, 17]. However, as we will show, none of the current approaches correctly recover from storage faults, motivating the need for a new approach.

2.3 RSM Recovery Taxonomy

To understand the different possible ways to handling storage faults in RSM systems, we analyze a broad range of approaches. We perform this analysis by two means: first, we analyze practical systems including ZooKeeper, LogCabin, etcd [25], and a Paxos-based system [24] using a fault-injection framework we developed (§5); second, we analyze techniques proposed by prior research or used in proprietary systems [15, 17].

Through our analysis, we classify the approaches into two categories: *protocol-oblivious* and *protocol-aware*. The oblivious approaches do not use any protocol-level knowledge to perform recovery. Upon detecting a fault, these approaches take a recovery action locally on the faulty node; such actions interact with the distributed protocols in unsafe ways, leading to data loss. The protocol-aware approaches use some RSM-specific knowledge to recover; however, they do not use this knowledge correctly, leading to undesirable outcomes. Our taxonomy is *not* complete in that there may be other techniques; however, to the best of our knowledge, we have not observed other approaches apart from those in our taxonomy.

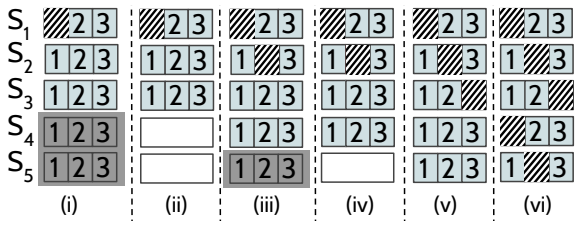


Figure 1: **Sample Scenarios.** *The figure shows sample scenarios in which current approaches fail. Faulty entries are striped. Crashed and lagging nodes are shown as gray and empty boxes, respectively.*

To illustrate the problems, we use Figure 1. In all cases, log entries[†] 1, 2, and 3 are committed; losing these items will violate safety. Table 1 shows how each approach behaves in Figure 1's scenarios. As shown in the table, all current approaches lead to safety violation (e.g., data loss), low availability, or both. A recovery mechanism that effectively uses redundancy should be safe and available in all cases. Table 1 also compares the approaches along other axes such as performance, maintenance overhead (intervention and extra nodes), recovery time, and complexity. Although Figure 1 shows only faults in the *log*, the taxonomy applies to other structures including the snapshots and the metainfo.

NoDetection. The simplest reaction to storage faults is none at all: to trust every layer in the storage stack to work reliably. For example, a few prototype Paxos-based systems [24] do not use checksums for their on-disk data; similarly, LogCabin does not protect its snapshots with checksums. *NoDetection* trivially violates safety; corrupted data can be obliviously served to clients. However, deployed systems do use checksums and other integrity strategies for most of their on-disk data.

Crash. A better strategy is to use checksums and handle I/O errors, and crash the node on detecting a fault. *Crash* may seem like a good strategy because it intends to prevent any damage that the faulty node may inflict on the system. Our experiments show that the *Crash* approach is common: LogCabin, ZooKeeper, and etcd crash sometimes when their logs are faulty. Also, ZooKeeper crashes when its snapshots are corrupted.

Although *Crash* preserves safety, it suffers from severe unavailability. Given that nodes could be unavailable due to other failures, even a single storage fault results in unavailability, as shown in Figure 1(i). Similarly, a single fault even in different portions of data on a majority (e.g., Figure 1(v)) renders the system unavailable. Note that simply restarting the node does not help; storage faults, unlike other faults, could be persistent: the node will encounter the same fault and crash again until manual intervention, which is error-prone and may cause a data loss. Thus, it is desirable to recover automatically.

Truncate. A more sophisticated action is to truncate (possibly faulty) portions of data and continue operating. The intuition behind *Truncate* is that if the faulty data is discarded, the node can continue to operate (unlike *Crash*), improving availability.

[†]A log entry contains a state-machine command and data.

Class	Approach	Safety	Availability	Performance	No Intervention	No extra nodes	Fast Recovery	Low Complexity	(i)	(ii)	(iii)	(iv)	(v)	(vi)
Protocol Oblivious	*NoDetection*	×	√	√	√	√	na	√	E	E	E	E	E	E
	Crash	√	×	√	×	√	na	√	U	C	U	C	U	U
	Truncate	×	√	√	√	√	×	√	C	L	C	L	L	L
	DeleteRebuild	×	√	√	×	√	×	√	C	L	C	L	L	L
Protocol Aware	*MarkNonVoting*	×	×	√	√	√	×	√	U	C	U	C	U	U
	Reconfigure	√	×	√	×	×	×	√	U	C	U	C	U	U
	Byzantine FT	√	×	×	√	×	na	×	U	C	U	U	U	U
	CTRL	√	√	√	√	√	√	√	C	C	C	C	C	C

E- Return Corrupted, L- Data Loss, U- Unavailable, C- Correct

Table 1: **Recovery Taxonomy.** *The table shows how different approaches behave in Figure 1 scenarios. While all approaches are unsafe or unavailable,* CTRL *ensures safety and high availability.*

However, we find that *Truncate* can cause a safety violation (data loss). Consider the scenario shown in Figure 2 in which entry 1 is corrupted on S_1; S_4, S_5 are lagging and do not have any entry. Assume S_2 is the leader. When S_1 reads its log, it detects the corruption; however, S_1 truncates its log, losing the corrupted entry and all subsequent entries (Figure 2(ii)). Meanwhile, S_2 (leader) and S_3 crash. S_1, S_4, and S_5 form a majority and elect S_1 the leader. Now the system does not have any knowledge of committed entries 1, 2, and 3, resulting in a *silent data loss*. The system also commits new entries x, y, and z in the place of 1, 2, and 3 (Figure 2(iii)). Finally, when S_2 and S_3 recover, they follow S_1's log (Figure 2(iv)), completely removing entries 1, 2, and 3.

In summary, although the faulty node detects the corruption, it truncates its log, losing the data locally. When this node forms a majority along with other nodes that are lagging, data is silently lost, violating safety. We find this safety violation in ZooKeeper and LogCabin.

Further, *Truncate* suffers from *inefficient recovery*. For instance, in Figure 1(i), S_1 truncates its log after a fault, losing entries 1, 2, and 3. Now to fix S_1's log, the leader needs to transfer *all* entries, increasing S_1's recovery time and wasting network bandwidth. ZooKeeper and LogCabin suffer from this slow recovery problem.

DeleteRebuild. Another commonly employed action is to manually delete all data on the faulty node and restart the node. Unfortunately, similar to *Truncate*, *DeleteRebuild* can violate safety; specifically, a node whose data is deleted could form a majority along with the lagging nodes, leading to a silent data loss. Surprisingly, administrators often use this approach hoping that the faulty

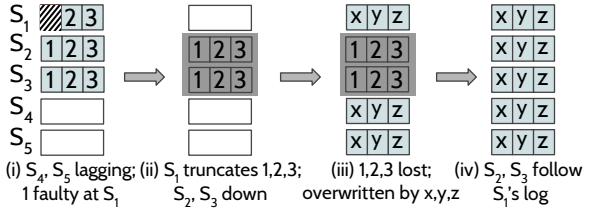


Figure 2: **Safety Violation Example.** *The figure shows the sequence of events which exposes a safety violation in Truncate.*

node will be "simply fixed" by fetching the data from other nodes [63, 65, 73]. *DeleteRebuild* also suffers from the slow recovery problem similar to *Truncate*.

MarkNonVoting. In this approach, used by a Paxos-based system at Google [17], a faulty node deletes all its data on a fault and marks itself as a non-voting member; the node does not participate in elections until it observes one round of consensus and rebuilds its data from other nodes. By marking a faulty node as non-voting, safety violations such as the one in Figure 2 are avoided. However, *MarkNonVoting* can sometimes violate safety as noted by prior work [70]. The underlying reason for unsafety is that a corrupted node deletes all its state including the promises[†] given to leaders. Once a faulty node has lost its promise given to a new leader, it could accept an entry from an old leader (after observing a round of consensus on an earlier entry). The new leader, however, still believes that it has the promise from the faulty node and so can overwrite the entry, previously committed by the old leader.

Further, this approach suffers from unavailability. For example, when only a majority of nodes are alive, a single fault can cause unavailability because the faulty node cannot vote; other nodes cannot now elect a leader.

Reconfigure. In this approach, a faulty node is removed and a new node is added. However, to change the configuration, a configuration entry needs to be committed by a majority. Hence, the system remains unavailable in many cases (for example, when a majority are alive but one node's data is corrupted). Although *Reconfigure* is not used in practical systems to tackle storage faults, it has been suggested by prior research [15, 44].

BFT. An extreme approach is to use a Byzantine-fault-tolerant algorithm which should theoretically tolerate storage faults. However, *BFT* is expensive to be used in practical storage systems; specifically, *BFT* can achieve only half the throughput of what a crash-tolerant protocol can achieve [21]. Moreover, *BFT* requires $3f + 1$ nodes to tolerate f faults [2], thus remaining unavailable in most scenarios in Figure 1.

Taxonomy Summary. None of the current approaches effectively use redundancy to recover from storage faults. Most approaches do not use any protocol-level knowledge to recover; for example, *Truncate* and *DeleteRebuild* take actions locally on the faulty node and so interact with the distributed protocol in unsafe ways, causing a global data loss. Although some approaches (e.g., *MarkNonVoting*) use some RSM-specific knowledge, they do not do so correctly, causing data loss or unavailability. Thus, to ensure safety and high availability, a recovery approach should effectively use redundancy by exploiting protocol-specific knowledge. Further, it is beneficial to avoid other problems such as manual intervention and slow recovery. Our protocol-aware approach, CTRL, aims to achieve these goals.

[†] In Paxos, a promise for a proposal numbered p is a guarantee given by a follower (acceptor) to the leader (proposer) that it will not accept a proposal numbered less than p in the future [41].

3 Corruption-Tolerant Replication

Designing a correct recovery mechanism needs a careful understanding of the underlying protocols of the system. For example, the recovery mechanism should be cognizant of how updates are performed on the replicated data and how the leader is elected. We base CTRL's design on the following important protocol-level observations common to most RSM systems.

Leader-based. A single node acts as the leader; all data updates flow only through the leader.

Epochs. RSM systems partition time into logical units called *epochs*. For any given epoch, only one leader is guaranteed to exist. Every data item is associated with the epoch in which it was appended and its *index* in the log. Since the entries could only be proposed by the leader and only one leader could exist for an epoch, an $\langle epoch, index \rangle$ pair uniquely identifies a log entry.

Leader Completeness. A node will not vote for a candidate if it has more up-to-date data than the candidate. Since committed data is present at least in a majority of nodes and a majority vote is required to win the election, the leader is guaranteed to have all the committed data.

CTRL exploits these protocol-level attributes common to RSM systems to correctly recover from storage faults. CTRL divides the recovery responsibility between two components: the *local storage layer* and the *distributed recovery protocol*; while the storage layer reliably detects faulty data on a node, the distributed protocol recovers the data from redundant copies. Both the components use RSM-specific knowledge to perform their functions.

In this section, we first describe CTRL's fault model (§3.1) and safety and availability guarantees (§3.2). We then describe the local storage layer (§3.3). Finally, we describe CTRL's distributed recovery in two parts: first, we show how faulty *logs* are recovered (§3.4) and then we explain how faulty *snapshots* are recovered (§3.5).

3.1 Fault Model

Our fault model includes the standard failure assumptions made by crash-tolerant RSM systems: nodes could

	Fault Outcome	Possible Causes
Data	corrupted data	misdirected and lost writes in ext
	inaccessible data	LSE, corruptions in ZFS and btrfs
FS Metadata	missing files/directories	directory entry corrupted, fsck may remove a faulty inode
	unopenable files/directories	sanity check fails after inode corruption, permission bits corrupted
	files with more or fewer bytes	*i_size* field in the inode corrupted
	file system read-only	journal corrupted; fsck not run
	file system unmountable	superblock corrupted; fsck not run

Table 2: **Storage Fault Model.** *The table shows storage faults included in our model and possible causes that lead to a fault outcome.*

crash at any time and recover later, and nodes could be unreachable due to network failures [21, 42, 50]. Our model adds another realistic failure scenario where persistent data on the individual nodes could be corrupted or inaccessible. Table 2 shows a summary of our storage fault model. Our model includes faults in both user data and the file-system metadata blocks.

User data blocks in the files that implement the system's persistent structures could be affected by errors or corruption. A number of (possibly contiguous) data blocks could be faulty as shown by studies [12,59]. Also, a few bits/bytes of a block could be corrupted. Depending on the local file system in use, corrupted data may be returned obliviously or transformed into errors.

File-system metadata blocks can also be affected by faults; for example, the inode of a log file could be corrupted. Our fault model considers the following outcomes that can be caused by file-system metadata faults: files/directories may go missing, files/directories may be unopenable, a file may appear with fewer or more bytes, the file system may be mounted read-only, and in the worst case, the file system may be unmountable. Some file systems such as ZFS may mask most of the above outcomes from applications [72]; however, our model includes these faulty outcomes because they could realistically occur on other file systems that provide weak protection against corruption (e.g., ext2/3/4). Through fault-injection tests, we have verified that the metadata fault outcomes shown in Table 2 do occur on ext4.

3.2 Safety and Availability Guarantees

CTRL guarantees that if there exists at least one correct copy of a *committed* data item, it will be recovered or the system will wait for that item to be fixed; committed data will never be lost. In unlikely cases where all copies of a committed item are faulty, the system will correctly remain unavailable. CTRL also guarantees that the system will make a decision about an *uncommitted* faulty item as early as possible, ensuring high availability.

3.3 CTRL Local Storage Layer

To reliably recover, the storage layer (CLSTORE) needs to satisfy three key requirements. First, CLSTORE must be able to reliably detect a storage fault. Second,

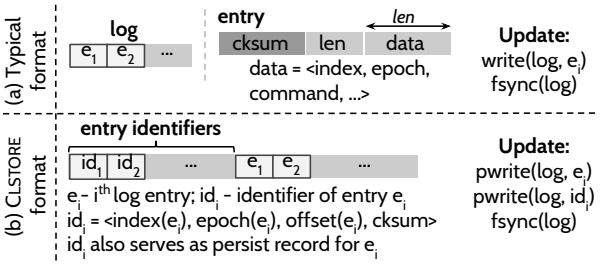


Figure 3: **Log Format.** *(a) shows the format and update protocol of a typical RSM log; (b) shows the same for* CLSTORE.

CLSTORE must correctly distinguish crashes from corruptions; safety can be violated otherwise. Third, CLSTORE must identify which pieces of data are faulty; only if CLSTORE identifies which pieces have been affected, can the distributed protocol recover those pieces.

3.3.1 Persistent Structures Overview

As we discussed, RSM systems maintain three persistent structures: the log, the snapshots, and the metainfo. CLSTORE uses RSM-specific knowledge of how these structures are used and updated, to perform its functions. For example, CLSTORE detects faults at a different granularity depending on the RSM data structure: faults in the log are detected at the granularity of individual entries, while faults in the snapshot are detected at the granularity of chunks. Similarly, CLSTORE uses the RSM-specific knowledge that a log entry is uniquely qualified by its $\langle epoch, index \rangle$ pair to identify faulty log entries.

Log. The log is a set of files containing a sequence of entries. The format of a typical RSM log is shown in Figure 3(a). The log is updated synchronously in the critical path; hence, changes made to the log format should not affect its update performance. CLSTORE uses a modified format as shown in Figure 3(b) which achieves this goal. A corrupted log is recovered at the granularity of individual entries.

Snapshots. The in-memory state machine is periodically written to a snapshot. Since snapshots can be huge, CLSTORE splits them into chunks; a faulty snapshot is recovered at the granularity of individual chunks.

Metainfo. The metainfo is special in that faulty metainfo *cannot* be recovered from other nodes. This is because the metainfo contains information unique to a node (e.g., its current epoch); recovering metainfo obliviously from other nodes could violate safety. CLSTORE uses this knowledge correctly and so maintains two copies of the metainfo locally; if one copy is faulty, the other copy is used. Fortunately, the metainfo is only a few tens of bytes in size and is updated infrequently; therefore, maintaining two copies does not incur significant overheads.

3.3.2 Detecting Faulty Data

CLSTORE uses well-known techniques for detection: *inaccessible* data is detected by catching return codes (e.g.,

EIO) and *corrupted* data is detected by a checksum mismatch. CLSTORE assumes that if an item and its checksum agree, then the item is not faulty. In the log, each entry is protected by a checksum; similarly, each chunk in a snapshot and the entire metainfo are checksummed.

CLSTORE also handles file-system metadata faults. Missing and unopenable files/directories are detected by handling error codes upon `open`. Log and metainfo files with fewer or more bytes are detected easily because these files are preallocated and are of a fixed size; snapshot sizes are stored separately, and CLSTORE cross-checks the stored size with the file-system reported size to detect discrepancies. A read-only/unmountable file system is equivalent to a missing data directory. In most cases of file-system metadata faults, CLSTORE crashes the nodes. Crashing reliably on a metadata fault preserves safety but compromises on availability. However, we believe this is an acceptable behavior because there are far more data blocks than metadata blocks; therefore, the probability of faults is significantly less for metadata than data blocks.

3.3.3 Disentangling Crashes and Corruption in Log

An interesting challenge arises when detecting corruptions in the log. A checksum mismatch for a log entry could occur due to two different situations. First, the system could have *crashed* in the middle of an update; in this case, the entry would be partially written and hence cause a mismatch. Second, the entry could be safely persisted but *corrupted* at a later point. Most log-based systems conflate these two cases: they treat a mismatch as a crash [30]. On a mismatch, they discard the corrupted entry and all subsequent entries, losing the data. Discarding entries due to such conflation introduces the possibility of a global data loss (as shown earlier in Figure 2).

Note that if the mismatch were really due to a crash, it is safe to discard the partially written entry. It is safe because the node would not have acknowledged to any external entity that it has written the entry. However, if an entry is *corrupted*, the entry cannot be simply discarded since it could be globally committed. Further, if a mismatch can be correctly attributed to a crash, the faulty entry can be quickly discarded locally, avoiding the distributed recovery. Hence, it is important for the local storage layer to distinguish the two cases.

To denote the completion of an operation, many systems write a commit record [13, 18]. Similarly, CLSTORE writes a persist record, p_i, after writing an entry e_i. For now, assume that e_i is ordered before p_i, i.e., the sequence of steps to append an entry e_i is $write(e_i), fsync(), write(p_i), fsync()$. On a checksum mismatch for e_i, if p_i is not present, we can conclude that the system crashed during the update. Conversely, if p_i is present, we can conclude that the mismatch was caused due to a corruption and *not* due to a crash. p_i is checksummed and is very small; it can be atomically written and thus cannot be "corrupted" due to a crash. If p_i is corrupted in addition to e_i, we can conclude that it is a corruption and not a crash.

The above logic works when e_i is ordered before p_i. However, such ordering requires an (additional) expensive *fsync* in the critical path, affecting log-update performance. For this reason, CLSTORE does not order e_i before p_i; thus, the append protocol is $\mathbf{t_1}$:$write(e_i)$, $\mathbf{t_2}$:$write(p_i)$, $\mathbf{t_3}$:$fsync()$.[†] Given this update sequence, assume a checksum mismatch occurs for e_i. If p_i is *not present*, CLSTORE can conclude that it is a crash (before $\mathbf{t_2}$) and discard e_i. Contrarily, if p_i is *present*, there are two possibilities: either e_i could be affected by a corruption after $\mathbf{t_3}$ or a crash could have occurred between $\mathbf{t_2}$ and $\mathbf{t_3}$ in which p_i hit the disk while e_i was only partially written. The second case is possible because file systems can reorder writes between two *fsync* operations and e_i could span multiple sectors [3, 19, 52, 53]. CLSTORE can still conclude that it is a corruption if e_{i+1} or p_{i+1} is present. However, if e_i is the *last entry*, then we cannot determine whether it was a crash or a corruption.[*]

The inability to disentangle the last entry when its persist record is present is not specific to CLSTORE, but rather a fundamental limitation in log-based systems. For instance, in ext4's journal_async_commit mode (where a transaction is not ordered before its commit record), a corrupted last transaction is assumed to be caused due to a crash, possibly losing data [38, 69]. Even if crashes and corruptions can be disentangled, there is little a single-machine system can do to recover the corrupted data. However, in a distributed system, redundant copies can be used to recover. Thus, when the last entry cannot be disentangled, CLSTORE safely marks the entry as *corrupted* and leaves it to the distributed recovery to fix or discard the entry based on the global commitment.

The entanglement problem does not arise for snapshots or metainfo. These files are first written to a temporary file and then atomically renamed. If a crash happens before the rename, the partially written temporary file is discarded. Thus, the system will never see a corrupted snapshot or metainfo due to a crash; if these structures are corrupted, it is because of a storage corruption.

3.3.4 Identifying Faulty Data

Once a faulty item is detected, it has to be *identified*; only if CLSTORE can identify a faulty item, the distributed layer can recover the item. For this purpose, CLSTORE redundantly stores an *identifier* of an item apart from the item itself; duplicating only the identifier instead of the whole item obviates the (2×) storage and performance

[†] The final *fsync* is required for durability.

[*] The proof of this claim is available [1].

overhead. However, storing the identifier near the item is less useful; a misdirected write can corrupt both the item and its identifier [9, 10]. Hence, identifiers are physically separated from the items they identify.

The ⟨*epoch*, *index*⟩ pair serves as the identifier for a log entry and is stored separately at the head of the log, as shown in Figure 3(b). The offset of an entry is also stored as part of the identifier to enable traversal of subsequent entries on a fault. The identifier of a log entry also conveniently serves as its persist record. Similarly, for a snapshot chunk, the ⟨*snap-index*, *chunk#*⟩ pair serves as the identifier; the *snap-index* and the snapshot size are stored in a separate file than the snapshot file. The identifiers have a nominal storage overhead (32 bytes for log entries and 12 bytes for snapshots), can be atomically written, and are also protected by a checksum.

It is highly unlikely an item and its identifier will both be faulty since they are physically separated [9, 10, 12, 59]. In such unlikely and unfortunate cases, CLSTORE crashes the node to preserve safety. Table 3 (second column) summarizes CLSTORE's key techniques.

3.4 CTRL Distributed Log Recovery

The local storage layer detects faulty data items and passes on their identifiers to the distributed recovery layer. We now describe how the distributed layer recovers the identified faulty items from redundant copies using RSM-specific knowledge. We first describe how *log entries* are recovered and subsequently describe *snapshot* recovery. As we discussed, metainfo files are recovered locally and so we do not discuss them any further. We use Figure 4 to illustrate how log recovery works.

Naive Approach: Leader Restriction. RSM systems do not allow a node with an incomplete log to become the leader. A naive approach to recovering from storage faults could be to impose an additional constraint on the election: *a node cannot be elected the leader if its log contains a faulty entry*. The intuition behind the naive approach is as follows: since the leader is guaranteed to have all committed data and our new restriction ensures that the leader is not faulty, faulty log entries on other nodes could be fixed using the corresponding entries on the leader. Cases (a)(i) and (a)(ii) in Figure 4 show scenarios where the naive approach could elect a leader. In (a)(i), only S_1 can become the leader because other nodes are either lagging or have at least one faulty entry. Assume S_1 is the leader also in case (a)(ii).

Fixing Followers' Logs. When the leader has no faulty entries, fixing the followers is straightforward. For example, in case (a)(i), the followers inform S_1 of their faulty entries; S_1 then supplies the correct entries. However, sometimes the leader might not have any knowledge of an entry that a follower is querying for. For instance, in case (a)(ii), S_5 has a faulty entry at index 3 but with a *different epoch*. This situation is possible because S_5 could have been the leader for epoch 2 and crashed immediately after appending an entry. As discussed earlier, an entry is uniquely identified by its ⟨*epoch*, *index*⟩; thus, when querying for faulty entries, a node needs to specify the epoch of the entry in addition to its index. Thus, S_5 informs the leader that its entry ⟨*epoch*:2, *index*:3⟩ is faulty. However, S_1 does not have such an entry in its log. If the leader does not have an entry that the follower has, then the entry *must be an uncommitted entry* because the leader is guaranteed to have all committed data; thus, the leader instructs S_5 to truncate the faulty entry and also replicates the correct entry.

Although the naive approach guarantees safety, it has availability problems. The system will be unavailable in cases such as the ones shown in (b): a leader cannot be elected because the logs of the alive nodes are either faulty or lagging. Note that even a single storage fault can cause an unavailability as shown in (b)(i). It is possible for a carefully designed recovery protocol to provide better availability in these cases. Specifically, since at least one intact copy of all committed entries exists, it is possible to collectively reconstruct the log.

3.4.1 Removing the Restriction Safely

To recover from scenarios such as those in Figure 4(b), we remove the additional constraint on the election. Specifically, any node that has a more up-to-date log can now be elected the leader even if it has faulty entries. This relaxation improves availability; however, two key questions arise: first, when can the faulty leader proceed to accept new commands? second, and more importantly, is it safe to elect a faulty node as the leader?

To accept a new command, the leader has to append the command to its log, replicate it, and apply it to the state machine. However, before applying the new command, *all* previous commands must be applied. Specifically, faulty commands cannot be skipped and later applied when they are fixed; such out-of-order application would violate safety. Hence, it is required for the leader to fix its faulty entries before it can accept new commands. Thus, for improved availability, the leader needs to fix its faulty entries as early as possible.

The crucial part of the recovery to ensure safety is to fix the leader's log using the redundant copies on the followers. In simple cases such as (b)(i) and (b)(ii), the leader S_1 could fix its faulty entry ⟨*epoch*:1, *index*:1⟩ using the correct entries from the followers and proceed to normal operation. However, in several scenarios, the leader cannot immediately recover its faulty entries; for example, none of the reachable followers might have any knowledge of the entry to be recovered or the entry to be recovered could also be faulty on the followers.

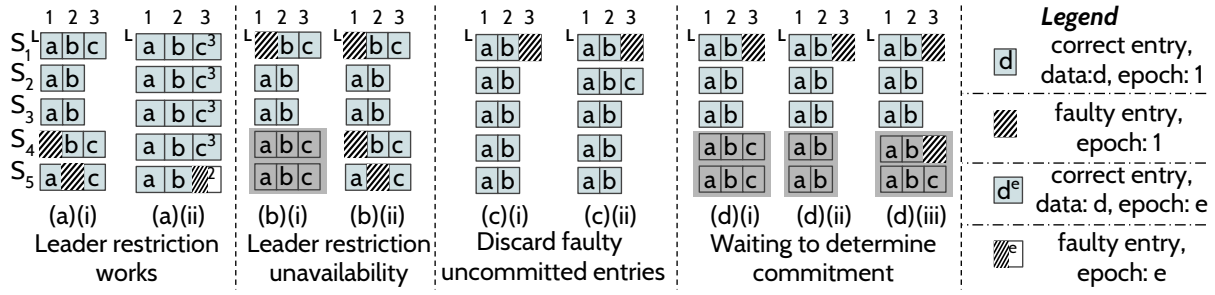


Figure 4: **Distributed Log Recovery.** *The figure shows how* CTRL*'s log recovery operates. All entries are appended in epoch 1 unless explicitly mentioned. For entries appended in other epochs, the epoch number is shown in the superscript. Entries shown as striped boxes are faulty. A gray box around a node denotes that it is down or extremely slow. The leader is marked with L on the left. Log indexes are shown at the top.*

3.4.2 Determining Commitment

The main insight to fix the leader's faulty log safely and quickly is to distinguish *uncommitted* entries from *possibly committed* ones; while recovering the committed entries is necessary for safety, uncommitted entries can be safely discarded. Further, discarding uncommitted faulty entries immediately is crucial for availability. For instance, in case (c)(i), the faulty entry on S_1 cannot be fixed since there are no copies of it; waiting to fix that entry results in indefinite unavailability. Sometimes, an entry could be partially replicated but remain uncommitted; for example, in case (c)(ii), the faulty entry on S_1 is partially replicated but is not committed. Although there is a possibility of recovering this entry from the other node (S_2), this is *not* necessary for safety; it is completely safe for the leader to discard this uncommitted entry.

To determine the commitment of a faulty entry, the leader queries the followers. If a majority of the followers respond that they do *not* have the entry (negative acknowledgment), then the leader concludes that the entry is uncommitted. In this case, the leader safely discards that and all subsequent entries; it is safe to discard the subsequent entries because entries are committed in order. Conversely, if the entry were committed, at least one node in this majority would have that entry and inform the leader of it; in this case, the leader can fix its faulty entry using that response.

Waiting to Determine Commitment. Sometimes, it may be impossible for the leader to quickly determine commitment. For instance, consider the cases in Figure 4(d) in which S_4 and S_5 are down or slow. S_1 queries the followers to recover its entry $\langle epoch{:}1, index{:}3\rangle$. S_2 and S_3 respond that they do not have such an entry (negative acknowledgment). S_4 and S_5 do not respond because they are down or slow. The leader, in this case, has to wait for either S_4 or S_5 to respond; discarding the entry without waiting for S_4 or S_5 could violate safety. However, once S_4 or S_5 responds, the leader will make a decision immediately. In (d)(i), S_4 or S_5 would respond with the correct entry, fixing the leader. In (d)(ii), S_4 or S_5 would respond that it does not have the entry, accumulating three (a majority out of five) negative acknowledgments; hence, the leader can conclude that the entry is uncommitted, discard it, and continue to normal operation. In (d)(iii), S_4 would respond that it has the entry but is faulty in its log too. In this case, the leader has to wait for the response from S_5 to determine commitment. In the unfortunate and unlikely case where all copies of an entry are faulty, the system will remain unavailable.

3.4.3 The Complete Log Recovery Protocol

We now assemble the pieces of the log recovery protocol. First, fixing faulty followers is straightforward; the committed faulty entries on the followers can be eventually fixed by the leader because the leader is guaranteed to have all committed data. Faulty entries on followers that the leader does not know about are uncommitted; hence, the leader instructs the followers to discard such entries.

The main challenge is thus fixing the leader's log. The leader queries the followers to recover its entry $\langle epoch{:}\text{e}, index{:}\text{i}\rangle$. Three types of responses are possible:

Response 1: **have** – a follower could respond that it has the entry $\langle epoch{:}\text{e}, index{:}\text{i}\rangle$ and is not faulty in its log.

Response 2: **dontHave** – a follower could respond that it does not have the entry $\langle epoch{:}\text{e}, index{:}\text{i}\rangle$.

Response 3: **haveFaulty** – a follower could respond that it has $\langle epoch{:}\text{e}, index{:}\text{i}\rangle$ but is faulty in its log too.

Once the leader collects these responses from the followers, it takes the following possible actions:

Case 1: if it gets a *have* response from at least one follower, it fixes the entry in its log.

Case 2: if it gets a *dontHave* response from a majority of followers, it confirms that the entry is uncommitted, discards that entry and all subsequent entries.

Case 3: if it gets a *haveFaulty* response from a follower, it waits for either Case 1 or Case 2 to happen.

Case 1 and Case 2 can happen in any order; both orderings are safe. Specifically, if the leader decides to discard the faulty entry (after collecting a majority *dontHave* responses), it is safe since the entry was uncommitted anyways. Conversely, there is no harm in accepting a correct entry (at least one *have* response) and replicating it. The

first to happen out of these two cases will take precedence over the other.

The leader proceeds to normal operation only after its faulty data is discarded or recovered. However, CTRL discards uncommitted data as early as possible and minimizes the recovery latency by recovering faulty data at a fine granularity (as we show in §5.2), ensuring that the leader proceeds to normal operation quickly.

The leader could crash or be partitioned while recovering its log. On a leader failure, the followers will elect a new leader and make progress. The partial repair done by the failed leader is harmless: it could have either fixed committed faulty entries or discarded uncommitted ones, both of which are safe.

3.5 CTRL Distributed Snapshot Recovery

Because the logs can grow indefinitely, periodically, the in-memory state machine is written to disk and the logs are garbage collected. Current systems including ZooKeeper and LogCabin do not handle faulty snapshots correctly (§2.3): they either crash or load corrupted snapshots obliviously. CTRL aims to recover faulty snapshots from redundant copies. Snapshot recovery is different from log recovery in that all data in a snapshot is committed and already applied to the state machine; hence, faulty snapshots cannot be discarded in any case (unlike uncommitted log entries which can be discarded safely).

3.5.1 Leader-Initiated Identical Snapshots

Current systems [43] have two properties with respect to snapshots. First, they allow new commands to be applied to the state machine while a snapshot is in progress. Second, they take index-consistent snapshots: a snapshot S_i represents the state machine in which log entries exactly up to i have been applied. One of the mechanisms used in current systems to realize the above two properties is to take snapshots in a `fork`-ed child process; while the child can write an index-consistent image to the disk, the parent can keep applying new commands to its copy of the state machine. CTRL should enable snapshot recovery while preserving the above two properties.

In current systems, every node runs the snapshot procedure independently, taking snapshots at different log indexes. Because the snapshots are taken at different indexes, snapshot recovery can be complex: a faulty snapshot on one node cannot be simply fetched from other nodes. Further, snapshots cannot be recovered at the granularity of chunks because they will be byte-wise non-identical; entire snapshots have to be transferred across nodes, slowing down recovery.

This complexity can be significantly alleviated if the nodes take the snapshot at the same index; identical snapshots also enable chunk-based recovery.

However, coordinating a snapshot operation across nodes can, in general, affect the common-case performance. For example, one naive way to realize identical snapshots is for the leader to produce the snapshot, insert it into the log as yet another entry, and replicate it. However, such an approach will affect update performance since the snapshot could be huge and all client commands must wait while the snapshot commits [49]. Moreover, transferring the snapshot to the followers wastes network bandwidth.

	Local Storage	Distributed Recovery
Log	granularity: entry; identifier:⟨*epoch*, *index*⟩; crash-corruption disentanglement	global-commitment determination to fix leader, leader fixes followers
Snapshot	granularity: chunk; identifier:⟨*snap-index*, *chunk#*⟩; no entanglement	leader-initiated identical snapshots, chunk-based recovery
Metainfo	granularity: file; identifier: n/a; no entanglement	none (only internal redundancy)

Table 3: **Techniques Summary.** *The table shows a summary of techniques employed by* CTRL*'s storage layer and distributed recovery.*

CTRL takes a different approach to identical snapshots that preserves common-case performance. The leader initiates the snapshot procedure by first deciding the index at which a snapshot will be taken and informing the followers of the index. Once a majority agree on the index, all nodes independently take a snapshot at the index. When the leader learns that a majority (including itself) have taken a snapshot at an index i, it garbage collects its log up to i and instructs the followers to do the same.

CTRL implements the above procedure using the log. When the leader decides to take a snapshot, it inserts a special marker called `snap` into the log. When the `snap` marker commits, and thus when a node applies the marker to the state machine, it takes a snapshot (i.e., the snapshot corresponds to the state where commands exactly up to the marker have been applied). Within each node, we reuse the same mechanism used by the original system (e.g., a `fork`-ed child) to allow new commands to be applied while a snapshot is in progress. Notice that the snapshot operation happens independently on all nodes but the operation will produce identical snapshots because the marker will be seen at the same log index by all nodes when it is committed. When the leader learns that a majority of nodes (including itself) have taken a snapshot at an index i, it appends another marker called `gc` for i; when the `gc` marker is committed and applied, the nodes garbage collect their log entries up to i.

3.5.2 Recovering Snapshot Chunks

With the identical-snapshot mechanism, snapshot recovery becomes easier. Once a faulty snapshot is detected, the local storage layer provides the distributed protocol the snapshot index and the chunk that is faulty. The distributed protocol recovers the faulty chunk from other

System	Recovery Scenario	Total Test Cases	Original Approach	Original Outcomes: Unavailable	Original Outcomes: Unsafe	Original Outcomes: Correct	CTRL Outcomes: Unavailable	CTRL Outcomes: Unsafe	CTRL Outcomes: Correct
LogCabin	Possible	2401	truncate	0	2355	46	0	0	2401
			crash	2355	0	46	0	0	2401
	Not possible	1695	truncate	0	1695	0	1695	0	0
			crash	1695	0	0	1695	0	0
ZooKeeper	Possible	2401	truncate	0	2355	46	0	0	2401
			crash	2355	0	46	0	0	2401
	Not possible	1695	truncate	0	1695	0	1695	0	0
			crash	1695	0	0	1695	0	0

(a) Targeted Corruptions

System	Experiment	Total Test Cases	Outcomes: Original: Unavailable	Original: Unsafe	Original: Correct	CTRL: Unavailable	CTRL: Unsafe	CTRL: Correct
LogCabin	Corruptions	5000	738	793	3469	0	0	5000
	Errors	5000	2497	0	2503	0	0	5000
ZooKeeper	Corruptions	5000	807	656	3537	0	0	5000
	Errors	5000	2469	0	2531	0	0	5000

(b) Random Block Corruptions and Errors

System	Total Test Cases	Outcomes: Original: Unavailable	Original: Unsafe	Original: Correct	CTRL: Unavailable	CTRL: Unsafe	CTRL: Correct
LogCabin	5000	4194	141	665	0	0	5000
ZooKeeper	5000	1306	1806	1888	0	0	5000

(c) Corruptions with Lagging Nodes

Table 4: **Log Recovery.** *(a) shows results for targeted corruptions; we trigger two policies (truncate and crash) in the original systems. (b) shows results for random block corruptions and errors. (c) shows results for random corruptions with crashed and lagging nodes.*

nodes. First, the leader recovers its faulty chunks from the followers and then fixes the faulty snapshots on followers. Three cases arise during snapshot recovery.

First, the log entries for a faulty snapshot may not be garbage collected yet; in this case, the snapshot is recovered locally from the log (after fixing the log if needed).

Second, if the log is garbage collected, then a faulty snapshot has to be recovered from other nodes. However, if the log entries for a snapshot are garbage collected, then at least a majority of the nodes must have taken the same snapshot. This is true because the `gc` marker is inserted only after a majority of nodes have taken the snapshot. Thus, faulty garbage-collected snapshots are recovered from those redundant copies.

Third, sometimes, the leader may not know a snapshot that a follower is querying for (for example, if a follower took a snapshot and went offline for a long time and the leader replaced that snapshot with an advanced one); in this case, the leader supplies the full advanced snapshot.

3.6 CTRL Summary

The storage layer detects and identifies faulty data. Atop the storage layer, the distributed protocol recovers the faulty items from redundant copies. Both the layers exploit RSM-specific knowledge to correctly perform their functions. A summary of CTRL's local storage and distributed recovery techniques is shown in Table 3.

4 Implementation

We implement CTRL in two different RSM systems, LogCabin (v1.0) and ZooKeeper (v3.4.8); while LogCabin is based on Raft, ZooKeeper is based on ZAB. Implementing CTRL's storage layer and distributed recovery took only a moderate developer effort; CTRL adds about 1500 lines of code to each of the base systems.

4.1 Local Storage Layer

We implemented CLSTORE by modifying the storage engines of LogCabin and ZooKeeper. In both systems, the log is a set of files, each of a fixed size and preallocated. The header of each file is reserved for the log-entry identifiers. The size of the reserved header is proportional to the file size. CLSTORE ensures that a log entry and its identifier are at least a few megabytes physically apart. Both systems batch many log entries to improve update performance. With batching, CLSTORE performs crash-corruption disentanglement as follows: the first faulty entry without an identifier and its subsequent entries are discarded; faulty entries preceding that point are marked as corrupted and passed on to the distributed layer.

In both systems, the state machine is a data tree. We modified both the systems to take index-consistent identical snapshots: when a `snap` marker is applied, the state machine (i.e., the tree) is serialized to the disk. The *snap-index* and snapshot size are stored separately. CLSTORE uses a chunk size of 4K, enabling fine-grained recovery.

In LogCabin, the metainfo contains the `currentTerm` and `votedFor` structures. Similarly, in ZooKeeper, structures such as `acceptedEpoch` and `currentEpoch` constitute the metainfo. CLSTORE stores redundant copies of metainfo and protects them using checksums.

Log entries, snapshot chunks, and metainfo are protected by a CRC32 checksum. CLSTORE detects inaccessible data items by catching errors (`EIO`); it then populates the item's in-memory buffer with zeros, causing a checksum mismatch. Thus, CLSTORE deals with both corruptions and errors as checksum mismatches.

4.2 Distributed Recovery

LogCabin. In Raft, *terms* are equivalent to epochs. Thus, a log entry is uniquely identified by its $\langle$*term*, *index*$\rangle$ pair. To fix the followers, we modified the AppendEntries RPC used by the leader to replicate entries [50]. The followers inform the leader of their faulty log entries and snapshot chunks in the responses of this RPC; the leader sends the correct entries and chunks in a subsequent RPC. A follower starts applying commands to its state machine once its faulty data is fixed. To fix the

leader, we added a new RPC which the leader issues to the followers. The leader does not proceed to normal operation until its faulty data is fixed. After a configurable recovery timeout, the leader steps down if it is unable to recover its faulty data (for example, due to a partition), allowing other nodes to become the leader. Several entries and chunks are batched in a single request/response, avoiding multiple round trips.

ZooKeeper. In ZAB, the epoch and index are packed into the *zxid* which uniquely identifies a log entry [5]. Followers discover and connect to the leader in Phase 1. We modified Phase 1 to send information about the followers' faulty data. The followers are synchronized with the leader in Phase 2. We modified Phase 2 so that the leader sends the correct data to the followers. The leader waits to hear from a majority during Phase 1 after which it sends a `newEpoch` message; we modified this message to send information about the leader's faulty data. The leader does not proceed to Phase 2 until its data is fixed.

5 Evaluation

We evaluate the correctness and performance of CTRL versions of LogCabin and ZooKeeper. We conducted our performance experiments on a three-node cluster on a 1-Gb network; each node is a 40-core Intel Xeon CPU E5-2660 machine with 128 GB memory running Linux 3.13, with a 500-GB SSD and a 1-TB HDD managed by ext4.

5.1 Correctness

To verify CTRL's safety and availability guarantees, we built a fault-injection framework that can inject storage faults (targeted corruptions and random block corruptions and errors). The framework can also inject crashes. By injecting crashes at different points in time, the framework simulates lagging nodes. After injecting faults, we issue reads from clients to determine whether the target system remains available and preserves safety.

We first exercise different log-recovery scenarios. Then, we test snapshot recovery, and finally file-system metadata fault recovery.

5.1.1 Log Recovery

Targeted Corruptions. We initialize the cluster with four log entries, replicated to all three nodes. We exercise all combinations of entry corruptions across the three nodes ($(2^4)^3 = 4096$ combinations). Out of the 4096 cases, a correct recovery is possible in 2401 cases (at least one non-faulty copy of each entry exists). In the remaining 1695 cases, recovery is not possible because one or more entries are corrupted on *all* the nodes. We inject targeted corruptions into two different sets of on-disk structures. In the first set, on a corruption, the original systems invoke the *truncate* action (i.e., they truncate faulty data and continue). In the second set, the original systems invoke the *crash* action (i.e., node crashes on detection). For example, while ZooKeeper *truncates* when the tail of a transaction is corrupted, it *crashes* the node if the transaction header is corrupted. CTRL always recovers the corrupted data from other replicas.

System	Total Test Cases	Outcomes: Original: Unavailable	Original: Unsafe	Original: Correct	CTRL: Unavailable	CTRL: Unsafe	CTRL: Correct
LogCabin	1000	297	257	446	0	0	1000
ZooKeeper	1000	417	200	383	0	0	1000

(a) Snapshot Recovery

Total Test Cases	Outcomes: Original: Unavailable	Original: Unsafe	Original: Correct	CTRL: Unavailable	CTRL: Unsafe	CTRL: Correct
1000	405	36	559	434	0	566
1000	329	192	479	502	0	498

(b) FS Metadata Faults

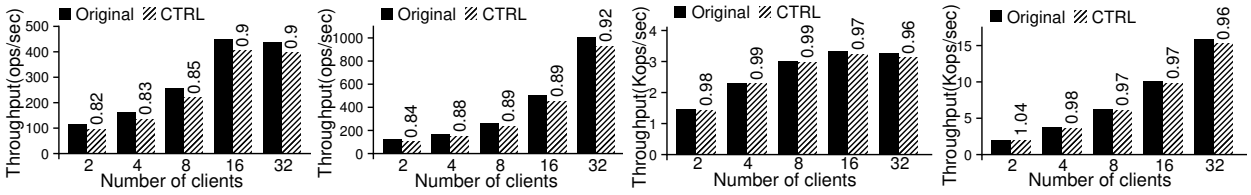
Table 5: **Snapshot and FS Metadata Faults.** *(a) and (b) show how* CTRL *recovers from snapshot and FS metadata faults, respectively.*

Table 4(a) shows the results. When recovery is possible, the original systems recover only in 46/2401 cases. In those 46 cases, no node or only one node is corrupted. In the remaining 2355 cases, the original systems are either unsafe (for *truncate*) or unavailable (for *crash*). In contrast, CTRL correctly recovers in all 2401 cases. When a recovery is not possible (all copies corrupted), the original systems are either unsafe or unavailable in all cases. CTRL, by design, correctly remains unavailable since continuing would violate safety.

Random Block Corruptions and Errors. We initialize the cluster by replicating a few entries to all nodes. We first choose a random set of nodes. In each such node, we then corrupt a randomly selected file-system block (from the files implementing the log). We repeat this process, producing 5000 test cases. We similarly inject block errors. Since we inject a fault into a block, several entries and their checksums within the block will be faulty.

Table 4(b) shows the results. For *block corruptions*, original LogCabin is unsafe or unavailable in about 30% $((738+793)/5000)$ of cases. Similarly, original ZooKeeper is incorrect in about 30% of cases. On a *block error*, original LogCabin and ZooKeeper simply crash the node, leading to unavailability in about 50% of cases. In contrast, CTRL correctly recovers in all cases.

Faults with Crashed and Lagging Nodes. In the previous experiments, all entries were committed and present on all nodes. In this experiment, we inject crashes at different points on a random set of nodes while inserting entries. Thus, in the resultant log states, nodes could be lagging, entries could be uncommitted, and have different epochs on different nodes for the same log index. $\langle S_1 : [a^1,\_,\_], S_2 : [b^2,c^3,\_], S_3 : [b^2,\_,\_]\rangle$ is an example state where S_1 appends a at index 1 in epoch 1 (shown in superscript) and crashes, S_2 appends b at index 1 in epoch 2, replicates to S_3, then S_2, S_3 crash and recover,



(a) LogCabin: Write (HDD) (b) ZooKeeper: Write (HDD) (c) LogCabin: Write (SSD) (d) ZooKeeper: Write (SSD)

Figure 5: **Common-Case Write Performance.** *(a) and (b) show the write throughput in original and* CTRL *versions of LogCabin and ZooKeeper on an HDD. (c) and (d) show the same for SSD. The number on top of each bar shows the performance of* CTRL *normalized to that of original.*

S_2 appends c in epoch 3 and crashes. From each such state, we corrupt different entries, generating 5000 test cases. For example, from the above state, we corrupt a on $S1$ and b, c on S_2. If S_2 is elected the leader, S_2 needs to fix b from S_3 (since b is committed), discard c (c is uncommitted and cannot be recovered), and also instruct $S1$ to discard a (a is uncommitted) and replicate correct entry b. As shown in Table 4(c), CTRL correctly recovers from all such cases, while the original versions are unsafe or unavailable in many cases.

Model Checking. We also model checked CTRL's log recovery since it involves many corner cases, using a python-based model that we developed. We explored over 2.5M log states all of in which CTRL correctly recovered. Also, when key decisions are tweaked, the checker finds a violation immediately: for example, the leader concludes that a faulty entry is uncommitted only after gathering $\lfloor N/2 \rfloor + 1$ *dontHave* responses; if this number is reduced, then the checker finds a safety violation. We have also added the specification of CTRL's log recovery to the TLA+ specification of Raft [23] and confirmed that it correctly recovers from corruptions, while the original specification violates safety.

5.1.2 Snapshot Recovery

We trigger the nodes to take a snapshot, crashing them at different points, producing three possible states for each node: l, t, and g, where l is a state where the node has only the log (it has not taken a snapshot), t is a snapshot for which garbage collection has not been performed yet, and g is a snapshot which has been garbage collected. We produce all possible combinations of states across three nodes. On each such state, we randomly pick a set of nodes to inject faults, and corrupt a random combination of snapshots and log entries, generating 1000 test cases. For example, $\langle S_1 : t, S_2 : g, S_3 : l \rangle$ is a base state on which we corrupt snapshot t and a few preceding log entries on S_1 and g on S_2. In such a state, if S_1 becomes the leader, it has to fix its log from S_3, then has to locally recover its t snapshot, after which it has to fix g on S_2. S_1 also needs to install the snapshot on S_3. As shown in Table 5(a), CTRL correctly recovers from all such cases. Original LogCabin is incorrect in about half of the cases because it obliviously loads faulty snapshots sometimes and crashes sometimes. Original ZooKeeper crashes the node if it is unable to locally construct the data from the snapshot and the log, leading to unavailability; unsafety results because a faulty log is truncated in some cases.

5.1.3 File-system Metadata Faults

To test how CTRL recovers from file-system metadata faults, we corrupt file-system metadata structures (such as inodes and directory blocks) resulting in unopenable files, missing files, and files with fewer or more bytes. We inject such faults in a randomly chosen file on one or two nodes at a time, creating 1000 test cases. Table 5(b) shows the results. In some cases, the faulty nodes in original versions crash because of a failed deserialization or assertion. However, sometimes original LogCabin and ZooKeeper do not detect the fault and continue operating, violating safety in 36 and 192 cases, respectively. In contrast, CTRL reliably crashes the node on a file-system metadata fault, preserving safety always.

5.2 Performance

We now compare the common-case performance of the CTRL versions against the original versions. In both LogCabin and ZooKeeper, reads are served from memory and the read paths are not affected by CTRL. Hence, we show only performance of write workloads. The workload runs for 300 seconds, inserting entries each of size 1K. Both systems batch writes to improve throughput. Snapshots are taken periodically during the updates. Numbers reported are the average over five runs.

Figure 5(a) and (b) show the throughput on an HDD for varying number of clients in LogCabin and ZooKeeper, respectively. CLSTORE physically separates the identifier from the entry; this separation induces a seek on disks in the update path. However, the seek cost is amortized when more requests are batched; CTRL has an overhead of 8%-10% for 32 clients on disks. Figure 5(c) and (d) show throughput on an SSD; CTRL adds very minimal overhead on SSDs (4% in the worst case). Note that our workload performs only writes and therefore shows CTRL's overheads in the worst case; for more realistic workloads that predominantly perform reads, the overheads should be even lower.

Fast Log Recovery. To show the potential reduction in log-recovery time, we insert 30K log entries (each of size 1K) and corrupt the first entry on one node. In origi-

nal LogCabin, the faulty node detects the corruption but truncates *all* entries; hence, the leader transfers all entries to bring the node up-to-date. CTRL fixes only the faulty entry, reducing recovery time. The faulty node is fixed in 1.24 seconds (32MB transferred) in the original system, while CTRL takes only 1.2 ms (7KB transferred). We see a similar reduction in log-recovery time in ZooKeeper.

6 Related Work

Our analysis of how RSM-based systems react to storage faults (§2.3) builds upon several fault-injection studies. Our design of CTRL (§3) builds upon several efforts on tolerating practical faults in distributed systems.

Storage Faults. Several studies on storage faults [34, 46, 48, 59, 60] motivated our work. Our previous work [29, 30] discovered fundamental reasons why distributed systems are not resilient to storage faults. However, the study did not uncover any safety or availability violations reported in §2.3; this is because the fault model in our previous study considers injecting only storage faults (precisely, a single storage fault on a single node at a time). In contrast, our fault model in this work considers crashes and network failures in addition to storage faults, exposing previously unknown safety and availability violations in RSM systems.

Targeted Approaches. Prior research describes two approaches [15, 17] to tackle storage faults in RSM systems. However, these approaches suffer from unavailability. Furthermore, the *MarkNonVoting* approach [17] can violate safety because important metainfo such as promises can be lost on a storage fault [70]. CTRL avoids such safety violations by storing two copies of metainfo on each node. Approaches that improve the reliability of other specific systems have also been proposed [68, 71].

Generic Approaches. Many generic approaches to handling practical faults other than crashes have been proposed. PASC [21] hardens systems to tolerate corruptions by maintaining two copies of the entire state on each node and assumes that both the copies will not be faulty at the same time. This approach does not work well for storage faults; having two copies of on-disk state incurs $2\times$ space overhead. Furthermore, in most cases, PASC crashes the node on a fault, causing unavailability. XFT [42] is designed to tolerate non-crash faults. However, it can tolerate only a total of $\lfloor (N-1)/2 \rfloor$ crash and non-crash faults. Similarly, UpRight [20] has an upper bound on the total faults to remain safe and available.

CTRL differs from the generic approaches through its special focus on storage faults. This focus brings two main advantages. First, CTRL attributes faults at a fine granularity: while the generic approaches consider a node as faulty if any of its data is corrupted, CTRL considers faults at the granularity of individual data items. Second, because of such fine-granular fault treatment, CTRL can be available as long as a majority of nodes are up and at least one non-faulty copy of a data item exists even though portions of data on *all* nodes could be corrupted. CTRL cannot tolerate arbitrary non-crash faults [40] (e.g., memory errors). However, CTRL can augment the generic approaches: for example, a system can be hardened against memory faults using PASC while making it robust to storage faults using CTRL.

7 Conclusions

Recovering from storage faults in distributed systems is surprisingly hard. We introduce *protocol-aware recovery* (PAR), a new approach that exploits protocol-specific knowledge of the underlying distributed system to correctly recover from storage faults. We design CTRL, a protocol-aware recovery approach for RSM systems. We experimentally show that CTRL correctly recovers from a range of storage faults with little performance overhead.

Our work is only a first step in hardening distributed systems to storage faults: while we have successfully applied the PAR approach to RSM systems, other classes of systems (e.g., primary-backup, Dynamo-style quorums) still remain to be analyzed. We believe the PAR approach can be applied to such classes as well. We hope our work will lead to more work on building reliable distributed storage systems that are robust to storage faults.

Acknowledgments

We thank Mahesh Balakrishnan (our shepherd), the anonymous reviewers, and the members of ADSL for their excellent feedback. We also thank CloudLab [56] for providing a great environment to run our experiments. This material was supported by funding from NSF grants CNS-1421033 and CNS-1218405, DOE grant DE-SC0014935, and donations from EMC, Huawei, Microsoft, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Crash-Corruption Disentanglement Proof. `http://research.cs.wisc.edu/adsl/Publications/par/`.

[2] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. *Distributed Computing*, 18(5):387–408, 2006.

[3] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai,

Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[4] Apache. ZooKeeper. https://zookeeper.apache.org/.

[5] Apache. ZooKeeper Guarantees, Properties, and Definitions. https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_guaranteesPropertiesDefinitions.

[6] Apache Cassandra. Cassandra Replication. http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html.

[7] Apache ZooKeeper. Applications and Organizations using ZooKeeper. https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy.

[8] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[9] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[10] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, CA, June 2007.

[11] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[12] Lakshmi Narayanan Bairavasundaram. *Characteristics, Impact, and Tolerance of Partial Disk Failures*. PhD thesis, University of Wisconsin, Madison, 2008.

[13] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[14] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: An Exercise in Distributed Computing. *Commun. ACM*, 25(4):260–274, April 1982.

[15] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[16] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[17] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, August 2007.

[18] 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.

[19] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

[20] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright Cluster Services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[21] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical Hardening of Crash-Tolerant Systems. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, June 2012.

[22] Jeff Dean. Building Large-Scale Internet Services. http://static.googleusercontent.com/media/research.google.com/en//people/jeff/SOCC2010-keynote-slides.pdf.

[23] Diego Ongaro. Raft TLA+ Specification. https://github.com/ongardie/raft.tla.

[24] epaxos. epaxos source code. https://github.com/efficient/epaxos.

[25] etcd. etcd. https://coreos.com/etcd.

[26] Etcd. Etcd: Production users. https://coreos.com/etcd/docs/latest/production-users.html.

[27] Daniel Fryer, Dai Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the Integrity of Transactional Mechanisms. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, CA, February 2014.

[28] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

[29] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to File-System Faults. *ACM Trans. Storage*, 13(3):20:1–20:33, September 2017.

[30] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 2017.

[31] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[32] Matthias Grawinkel, Thorsten Schafer, Andre Brinkmann, Jens Hagemeyer, and Mario Porrmann. Evaluation of Applied Intra-disk Redundancy Schemes to Improve Single Disk Reliability. In *Proceedings of the 19th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Washington, DC, July 2011.

[33] Kevin M Greenan, Darrell DE Long, Ethan L Miller, Thomas Schwarz, and Avani Wildani. Building Flexible, Fault-Tolerant Flash-Based Storage Systems. In *The 5th Workshop on Hot Topics in System Dependability (HotDep '09)*, Lisbon, Portugal, June 2009.

[34] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*, New York, New York, December 2009.

[35] James Hamilton. On Designing and Deploying Internet-Scale Services. In *Proceedings of the 21st Annual Large Installation System Administration Conference (LISA '07)*, Dallas, Texas, November 2007.

[36] James Myers. Data Integrity in Solid State Drives. http://intel.ly/2cF0dTT.

[37] John Goerzen. Silent Data Corruption Is Real. http://changelog.complete.org/archives/9769-silent-data-corruption-is-real.

[38] Jonathan Corbet. Responding to ext4 journal corruption. https://lwn.net/Articles/284037/.

[39] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-Performance Broadcast for Primary-Backup Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, China, June 2011.

[40] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. HAFT: Hardware-assisted Fault Tolerance. In *Proceedings of the EuroSys Conference (EuroSys '16)*, London, United Kingdom, April 2016.

[41] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

[42] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical Fault Tolerance Beyond Crashes. In *Proceedings*

of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16), Savannah, GA, November 2016.

[43] LogCabin. LogCabin. `https://github.com/logcabin/logcabin`.

[44] Jacob R Lorch, Atul Adya, William J Bolosky, Ronnie Chaiken, John R Douceur, and Jon Howell. The SMART Way to Migrate Replicated Stateful Services. In *Proceedings of the EuroSys Conference (EuroSys '06)*, Leuven, Belgium, April 2006.

[45] Parisa Jalili Marandi, Christos Gkantsidis, Flavio Junqueira, and Dushyanth Narayanan. Filo: Consolidated Consensus As a Cloud Service. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016.

[46] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*, Portland, Oregon, June 2015.

[47] MongoDB. MongoDB Replication. `https://docs.mongodb.org/manual/replication/`.

[48] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*, Haifa, Israel, June 2016.

[49] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014.

[50] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.

[51] Bernd Panzer-Steindel. Data Integrity. *CERN/IT*, 2007.

[52] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 2017.

[53] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[54] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, October 2005.

[55] Redis. Redis Replication. `http://redis.io/topics/replication`.

[56] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[57] Robert Harris. Data corruption is worse than you know. `http://www.zdnet.com/article/data-corruption-is-worse-than-you-know/`.

[58] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[59] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, CA, February 2010.

[60] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, February 2016.

[61] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: The Growth of a Distributed System. *ACM Trans. Comput. Syst.*, 2(1):3–23, February 1984.

[62] Thomas Schwarz, Ahmed Amer, Thomas Kroeger, Ethan L. Miller, Darrell D. E. Long, and Jehan-Franois Pris. RESAR: Reliable Storage at Exabyte

Scale. In *Proceedings of the 24th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, London, United Kingdom, September 2016.

[63] Romain Slootmaekers and Nicolas Trangez. Arakoon: A Distributed Consistent Key-Value Store. In *SIGPLAN OCaml Users and Developers Workshop*, volume 62, 2012.

[64] Stackoverflow. Can ext4 detect corrupted file contents? `http://stackoverflow.com/questions/31345097/can-ext4-detect-corrupted-file-contents`.

[65] Stackoverflow. ZooKeeper Clear State. `http://stackoverflow.com/questions/17038957/org-apache-hadoop-hbase-pleaseholdexception-master-is-initializing`.

[66] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[67] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, CO, December 1995.

[68] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. HARDFS: Hardening HDFS with Selective and Lightweight Versioning. In *Proceedings of the 11th Conference on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[69] Theodore Ts'o. What to do when the journal checksum is incorrect. `https://lwn.net/Articles/284038/`.

[70] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2015.

[71] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus Scalable Block Store. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, April 2013.

[72] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, CA, February 2010.

[73] ZooKeeper Jira Issues. Unable to load database on disk when restarting after node freeze. `https://issues.apache.org/jira/browse/ZOOKEEPER-1546`.

WAFL Iron: Repairing Live Enterprise File Systems

Ram Kesavan
NetApp, Inc.
ram.kesavan@gmail.com

Harendra Kumar*
Composewell Technologies
harendra.kumar@gmail.com

Sushrut Bhowmik
NetApp, Inc.
sushrut@netapp.com

Abstract

Consistent and timely access to an arbitrarily damaged file system is an important requirement of enterprise-class systems. Repairing file system inconsistencies is accomplished most simply when file system access is limited to the repair tool. Checking and repairing a file system while it is open for general access present unique challenges. In this paper, we explore these challenges, present our online repair tool for the NetApp® WAFL® file system, and show how it achieves the same results as offline repair even while client access is enabled. We present some implementation details and evaluate its performance. To the best of our knowledge, this publication is the first to describe a fully functional online repair tool.

1 Introduction

File system state can be corrupted by hardware failures [9, 38, 41, 45, 35, 53] including misdirected or lost writes and media errors, software bugs [4, 14, 51], and even human errors such as inserting a device in the wrong shelf or slot. Journaling [25, 49, 37, 44, 10], shadow paging [29, 13, 50], and soft updates [22] are all techniques that provide file system crash consistency. Recon [21], incremental checksum, and digest-based transaction audit [34] are well-understood mechanisms to prevent some hardware and software bugs from corrupting the file system. Despite such defenses, corruptions are still an unfortunate reality for file systems, and our customer deployment confirms that reality.

Corruption can affect both user data and metadata. A corrupted user data block affects client access only to it and not to the rest of the file system. However, a corrupted metadata block not only affects access to user data but can also compromise other metadata when client operations are processed. Repair of user data is limited to recovery from backup, whereas metadata can be repaired using consistency properties of the file system. Corruption in metadata is detected either when a metadata block fails some checks as it is read into memory or later when some operation detects a violation of a file system invariant. Some corruption can be fixed on-the-fly using techniques such as RAID or erasure coding. If not, the file system is placed in a restricted mode while it is repaired by an offline repair tool, such as *fsck* [26]. Restricting access serves two purposes: it greatly simplifies the task of repair, and it prevents the inconsistency from causing further damage.

Several approaches have been proposed to speed up or to improve *offline* repair [36, 12, 27, 24] but the time to completion remains proportional to the amount of metadata that must be checked, which in turn is a function of the size of the file system. Enterprises require continuous access to data; any disruption outside of scheduled maintenance windows is highly undesirable. The damaged file system must be made completely available to clients as soon as possible, and repair must therefore not preclude client access. Repair should also not invalidate any data that is accessed or modified by clients after repair is initiated. Additionally, the impact to client performance must be within acceptable limits.

The NetApp® WAFL® file system [29, 20] validates the consistency of its data structures during normal operation. In the rare event of a detected inconsistency that can be neither trivially recovered by RAID nor tolerated, the file system is marked as inconsistent and taken offline. *WAFLIron* [31] (henceforth called *Iron*) repairs the file system even while allowing full access to clients. This paper presents the challenges inherent to this mode of file system repair. It describes the high-level design and implementation of Iron, and evaluates its performance. Most importantly, it explains the theory behind Iron to show that it fixes file system inconsistencies with practically the same assurances as offline repair. Our field experience has shown that Iron is extremely reliable, and meets high performance goals. Although online repair is available for ReFS [30], to the best of our knowledge, there is no prior published work on this topic.

*Research performed while working at NetApp.

2 Motivation

A file system can be damaged in several ways [52], but a repair tool is required in only some cases.

2.1 When Is Repair Required?

NetApp is a storage and data management company that offers software, systems, and services to manage and store data, including the proprietary NetApp ONTAP® software, which is built on the WAFL file system. Although fsck [40] was originally designed to fix inconsistencies created by an unclean shutdown, WAFL and other file systems use well-understood techniques such as copy-on-write (COW) and journaling to guarantee file system consistency after a crash. WAFL logs recent client operations in a stand-alone nonvolatile journal, and those operations are replayed after a crash to recover them [29]. Because the replay of each operation re-creates all necessary file system state, a corruption of the journal cannot corrupt the persistent file system; at worst, it might result in the loss of logged operations. Furthermore, this loss is limited because WAFL's transaction mechanism ensures the journal typically has client operations only from the past few seconds.

Each block in WAFL is written out to storage media together with a checksum and with some file system specific context that helps further identify the block [8, 13, 48, 47]. If a write is misdirected or lost by the device or if a previously-persisted block is damaged, a subsequent read results in a context or checksum mismatch. The damaged block can be recomputed and fixed by using the underlying RAID [43, 15]. This fixup is done on-the-fly when servicing a read or through a periodic background *scrub* [3]. Other file systems such as ZFS and Btrfs also leverage RAID or data mirroring [1, 5, 46] to provide similar protection. In the rare case of multidevice failures, reconstruction of damaged blocks can become impossible. If such blocks contain metadata that are critical to the functioning of WAFL, the file system is marked as inconsistent and is brought offline, so that it can be repaired.

Despite rigorous testing and prevention mechanisms, rarely occurring software bugs [4, 14, 51] and hardware errors [9, 38, 41, 45, 35, 53] might corrupt a block before its checksum is computed. Such faults cannot be detected by using the persistent checksum or context, and they cannot be repaired by using underlying redundancy [52]. WAFL detects such corruptions when it reads or uses metadata, and if the code path is unable to navigate past it, the file system is marked as inconsistent and is brought offline, so that it can be repaired.

2.2 Traditional Offline Repair

Exclusive access to the file system greatly simplifies offline repair, which walks the metadata of the file system exhaustively, checks them for inconsistencies, lists out each inconsistency with a recommended fix, and provides the choice to commit each fix [40]. Such an audit requires accounting metadata to track progress. Repair tools were designed to avoid writing to the physical storage that hosts the file system under repair until the administrator chooses to commit the recommended changes. Thus, the tools keep all their accounting data structures in memory until that time. In general, the amount of metadata increases with file system size, which means an increase in the memory that is required by the tool. This increase is typically offset either by breaking the file system into disjointed chunks of storage [28] or by the tool making multiple passes of the file system, thereby lowering memory requirements. *WAFLCheck*, the first and now obsolete offline tool for repairing the WAFL file system, suffered from similar drawbacks.

2.3 Enterprise Needs

Enterprise file systems are usually hundreds of TiB in size, and depending on the features supported their metadata can be both large (several GiB) and complex. Thus, the repair of a 100 TiB file system can take hours or even weeks depending on the I/O capability of the underlying media. Businesses require uninterrupted data availability; an hour-long outage can cost millions in lost revenue. Furthermore, an enterprise storage system typically hosts and exports multiple file systems. Because CPU and memory on such a system are shared resources, the repair of one file system can affect the performance of the others. Therefore, NetApp invested in building online repair instead of making incremental improvements to WAFLCheck. NetApp support staff get involved when a WAFL file system is marked as inconsistent and is taken offline. Under their supervision, the file system is brought online with an option to enable Iron. Clients gain full access to the data even while the persistent file system is checked and repaired. Iron logs its progress and completion, at which point the file system is marked as consistent. The time required for completion depends on several factors, such as file system size and client load on the system. All client ops that were logged in the nonvolatile journal are replayed; as implied earlier, Iron does not repair any corruptions in the journal.

One version of "online" repair [39] argues that orphaned blocks and inodes are the primary outcomes of file system inconsistency. Hence, a snapshot of the file system is taken, the file system is made available, and background

fsck runs on the snapshot to reclaim orphaned blocks and inodes into the active version of the file system. In general, the WAFL file system easily survives orphaned blocks and inodes, and is taken offline only when it encounters an inconsistency that prevents continued operation. Therefore, this approach does not apply.

2.4 Considerations With Online Repair

[1] Unconditional commit: Iron fixes corruptions as it encounters them so that the file system can continue operations. This means, unlike fsck, the administrator is not given the option to accept or decline repairs. With fsck, if the damage is truly extensive it is likely that the administrator would choose not to commit and restore the entire file system from backup. Online repair does not preclude this option, but all intervening client mutations are lost when the entire file system is restored from backup. There is one scenario in which offline repair is preferable. A customer with poor practices might have no (recent) backup of the file system, and might want to conservatively use offline repair and carefully choose which of the recommended fixes are committed.

[2] Speed of repair: An ONTAP system hosts multiple file systems. An aggressive repair process can affect the performance of the clients of all those file systems. A customer might prioritize the completion of Iron because the full repair of the dataset is more important to their business than are the IOPS made available to the applications on that storage system—especially if the backup copies of the corrupted dataset are not sufficiently recent. The ability to control the speed of the repair process is therefore important.

3 Metadata and Inconsistencies

This section presents a simplified version of the WAFL file system (persistent) metadata, and the the inconsistencies that can affect them.

3.1 Persistent Metadata

WAFL is a UNIX-style file system that uses inodes to represent its files, which are the basic construct for storing both metadata and user data. An inode stores metadata (permissions, timestamps, block count) about the file and its data in a symmetric tree of fixed-size blocks, henceforth called the inode's *blocktree*. Only leaf nodes (L_0s) of the blocktree hold the file's data; interior nodes are called *indirect* blocks. An inode that stores file system metadata in its leaves is called a *metafile*. Inodes themselves are stored in the leaves of the *inodefile* metafile, and its blocktree is rooted in the superblock of the file system. All together, they constitute the WAFL file system tree [29].

Each directory is stored in a file that contains a list of entries, where each entry is the name of a file or subdirectory and its corresponding inode number; the root directory is stored in a well-known inode. The reference count (*refcnt*) metafile stores a list of integers where the i^{th} integer tracks the number of references to the i^{th} block of the file system[1]. Multiple references occur because of features such as deduplication that result in block-sharing. The file system stores several counters, some that reside in structures such as inodes, and others that are global.

From the viewpoint of repair, we classify WAFL file system metadata into two broad categories.

[1] Primary metadata constitute the blocks of the file system required to read user data. In WAFL, this comprises the superblock, the inodefile blocktree, directories, inodes (for user files) and their blocktrees. WAFL stores copies of some key data structures primarily to protect against storage media failures; corruption due to most software bugs will damage both copies. Importantly, corrupted primary metadata in WAFL cannot be reconstructed by using other metadata. A damaged indirect block in a blocktree cannot be repaired, and therefore, at best, its child sub-tree can be recovered in the *lost+found* folder [40] on completion of the repair process. Similarly, the corresponding inode of a damaged directory entry can be recovered only in lost+found. It should be noted that, to avoid single points of failure, a file system could build redundancy into its primary metadata—each block could encode its location in the file system tree; however, that comes with additional complexity and the run-time cost of maintaining it. To protect against storage media failures, ONTAP uses efficient redundancy techniques: dual-parity RAID [15], triple-parity RAID [2], and remote synchronous mirroring [6].

[2] Derived metadata track the usage of resources, such as blocks and inodes, by the file system and can be recomputed by walking the primary metadata. They are typically maintained by the file system software for its efficient functioning, or for enabling specific features, such as file system quotas. The block count of an inode, the refcnt file, and various global counters are all examples of such metadata in WAFL. Damage to derived metadata can usually be repaired based on primary or other derived metadata.

Note that although derived metadata can eventually be reconstituted, they are needed for basic file system op-

[1] In reality, a bitmap tracks the first and the refcnt tracks additional references to blocks [32, 33]. Without loss of generality, the bitmap is subsumed into the refcnt metafile for the purposes of this paper.

eration. For example, the file system must consult and update the refcnt metadata to process new mutations, but that metadata is not fully validated until repair completes. Therefore, the primary complexity of online repair centers around the repair of derived metadata even while the metadata are used by file system operations. The refcnt metafile is the largest and most complex derived metadata in WAFL, and is therefore deliberately used as a running example in this paper.

3.2 Inconsistencies

The enablement of Iron does not change how corruption in WAFL is detected; only the action precipitated by such detection. If Iron is not enabled and the software cannot navigate past the inconsistency, the file system is marked as inconsistent and is taken offline. Otherwise, it is repaired. Metadata can be corrupted in one of two ways.

[1] Manifest corruption: This form of corruption is detected either when the block is read into memory—checksum or context mismatch—or when some of its contents are used for the first time—well-known signatures appear wrong or some data structures are outside acceptable bounds. Such a block needs repair only if it cannot be recomputed by the underlying RAID, which can happen either because multiple hardware elements have failed or the block was corrupted before the associated parity was computed.

[2] Latent corruption: File system invariants typically define relationships across different metadata. A latent corruption violates a relationship even while each participant block is devoid of manifest corruption. The relationship might involve primary metadata only. For example, a directory L_0 and an inodefile L_0 might each be independently reliable, but the former maps a dir entry to an inode that is marked as free in the latter. Sec. 4.5 presents examples of latent corruption across primary and/or derived metadata. Latent corruption is detected only when a metadata consistency invariant in the code trips up. Before its detection, it can create further inconsistencies if used by the file system; Sec. 4.5 has more details.

Both forms of corruption can be caused by bugs in the file system logic or memory scribbles. Device failures and media errors typically result in manifest corruption only; the block will appear to be unreliable.

4 Basics of WAFL Iron

Much as offline repair would, Iron walks all primary metadata, checks consistency with other metadata, and makes repairs where necessary. However, full client access is enabled early on. After a file's blocktree has been completely checked, all derived metadata for the file (such as its block count) is verified. As mentioned earlier, WAFL stores all user data and metadata (both primary and derived) in files. Therefore, after all files in a file system have been checked, all derived metadata is verified and the file system is marked as consistent.

The first version of Iron (circa 2003) focused on mitigating the main drawbacks of WAFLCheck: (1) scaling of the metadata needed for checking the file system, and (2) allowing early file system access while still providing the same assurances as WAFLCheck, aside from the unconditional commit highlighted in Sec. 2.4. This paper focuses primarily on proving parity in functionality with offline repair, and therefore does not do justice to the details of implementation. This section presents the rules for addressing the complications from allowing client access, and presents the main design.

4.1 Rules for Iron

Rule #1, Interposition: *Every block is processed by Iron before the rest of the file system software can use it.* This rule lets Iron make repairs early, which prevents the rest of the file system software from making decisions based on inconsistencies. This rule necessitates a filter in the read-from-storage code path so that all blocks are examined by Iron first.

Rule #2, Irreversibility: *After Iron starts, any state that is exposed to the client cannot be revoked by any future repair done by Iron.* Practicality requires that the data served to a client, as well as the results of any client mutation, not be revoked by subsequent repair[2]. To satisfy this rule, a file system op (client or internal) waits when loading a block until Iron validates any metadata required to ensure that block's continued survival through the completion of Iron. This approach has two obvious implications: (1) The latency of an op can be significantly affected; later sections explore this impact. (2) Iron needs a definition of the metadata required to ensure a block's survival; we look at that implication next.

Let the relationship $b \rightarrow b_i$ define a WAFL consistency invariant where metadata block b_i must exist and contain the "right" information to ensure that block b belongs to the file system; b can be user data or metadata. In other words, Iron must either move b to lost+found, or create or modify b_i to preserve the relationship. This relationship is obviously transitive, i.e., if $b \rightarrow b_i$ and $b_i \rightarrow b_j$, then $b \rightarrow b_j$. We define the *essential set*, $\Psi(b)$, of all metadata blocks, such that $b_i \in \Psi(b) \implies b \rightarrow b_i$. When an op loads b, Iron uses the filter described in Rule #1

[2] In fact, this invariant is extended to include state exposed to all internal file system ops. It simplifies the interaction of Iron with several file system modules.

to load, check, and potentially repair all metadata blocks in $\Psi(b)$ before allowing the op to proceed. Thereafter (with help from Rule #3), Iron does not change anything in $\Psi(b)$ that revokes that state of b, thereby preserving Rule #2[3]. This is true even if b is modified by the op.

Let's look a little closer at the essential set. All ancestor blocks of b in the file system tree (Sec. 3.1) trivially belong to $\Psi(b)$. This includes any ancestor indirect blocks of b within its inode, the corresponding inodefile L_0, the ancestor indirect blocks of the inodefile, including the superblock. Iron is invoked through the *mount* command, and so no blocks of the file system are in memory at the start of Iron. Thus, Rule #2 is trivially satisfied for an ancestor block because it is always loaded before its child. *In fact, all primary metadata in $\Psi(b)$ can be exhaustively shown to satisfy this rule*; for brevity, we do not list them here. However, as an example, if b belongs to a user file, the directory block L_0 with the corresponding directory entry also belongs in $\Psi(b)$, and it is loaded and accessed before b. Derived metadata associated with b, such as the refcnt L_0 with its refcnt entry, also must be loaded and checked for consistency. Sec. 4.3 explores an important complication with the essential set.

Rule #3, Convergence: *As Iron incrementally checks and repairs metadata, it monotonically expands the portion of the file system metadata that is self-consistent.* Iron ensures that file system metadata is never checked more than once, and therefore the extra cost of checking the essential set when loading any block diminishes with the progression of Iron. For example, when a second child of block b is loaded, Iron does not repeat the checking of all primary metadata performed on the first load of a child of b. Rule #2 ensures that all metadata associated with new mutations to the file system have also been checked and are included in the portion of the file system metadata that is considered self-consistent. Thus, convergence is guaranteed. This rule implies that Iron maintains data structures to track its progress, which leads us to the next rule.

Rule #4, Scalability: *Data structures that Iron needs to track progress must scale with file system size without requiring additional system memory.* The previous two rules make it clear that Iron makes a single pass over the file system metadata. Iron scales its data structures with file system size by storing them in files that are paged in and out of the WAFL buffer cache [19] much like any other metafile; they are called *Iron status files*. Much like any other file in the file system, all previous rules apply to the creation, consultation, and mutation of the status files. In other words, the ever expanding portion of self-consistent metadata (of Rule #3) includes all status files.

[3] A new corruption introduced to a block after it has been checked may violate this statement; Sec. 5 discusses that topic in more detail.

4.2 Iron Status Files

Status files are created and used by Iron for each file system that it repairs. All status files are deleted upon completion of Iron. Status files can be broadly classified into *progress indicator metafiles* and *derived shadow metafiles*.

Progress indicator metafiles: This class of status files tracks the progress of Iron and avoids repeated work, both of which are necessary for ensuring Rule #3. One example is the *checked bitmap* status file, which is a vector of bits where the i^{th} bit indicates that the i^{th} block of the file system has been checked, and repaired if necessary. Because a metadata block can be scavenged from the buffer cache and subsequently re-read from storage, this bitmap ensures a given block is processed exactly once.

Derived shadow metafiles: Iron computes shadow versions of some derived metadata as it walks the file system. On completion, Iron compares the shadow version with the original version, repairs both manifest and latent corruption in that derived metadata, and discovers any orphaned resources that are tracked by that metadata. One example is the *claimed refcnt* status file, a list of shadow integers where the i^{th} integer tracks the references to block b_i that Iron encounters. On completion, Iron replaces each refcnt integer with its claimed refcnt counterpart; a count that changes to zero represents an orphaned block. Thus, Iron can ignore the corresponding refcnt block when it processes the essential set for a block; the refcnt L_0 in $\Psi(b_i)$ is replaced by the corresponding claimed refcnt L_0. Until Iron completes, the WAFL write allocator [18] consults both refcnt integers to decide if a block is free, and freeing a block requires decrementing both refcnt integers. The claimed refcnt integer can never underflow because Rule #2 guarantees that Iron claims a block before freeing it. Sec. 4.4.2 and Sec. 4.5 discuss the underflow and overflow of a damaged refcnt integer. Iron uses other shadow derived metadata in a similar fashion, but all of them are smaller in size and in complexity than the claimed refcnt file, and are therefore not discussed here.

4.3 Recursivity Within the Essential Set

Let's say that $b_i \in \Psi(b)$. When Iron loads b_i on behalf of b, Rule #2 forces a recursive load of all metadata blocks from $\Psi(b_i)$. Although this recursivity implies indefinitely long response times for client ops, we will show why that is not true in reality; let us look at each type of metadata in $\Psi(b)$.

[1] Primary metadata: For every primary metadata block $b_i \in \Psi(b)$, all primary metadata blocks in $\Psi(b_i)$

also belong in $\Psi(b)$ (and will therefore have already been loaded and checked earlier). For example, if b_i is an ancestor of b, then all ancestors of b_i are also ancestors of b. A similar argument can be made for directory blocks of $\Psi(b)$ when walking a pathname.

[2] Derived metadata: Depending on the specific derived metadata, let's say M, Iron breaks this recursion in one of two ways: (1) It loads and checks all of M during mount and before client access is allowed, and Rule #3 ensures M is not checked again. This approach is taken for smaller metadata. (2) Iron does not load or consult M when it processes the essential set. Instead, it maintains and updates a derived shadow metafile that corresponds to M; M is checked only when it is loaded for other file system activity. We illustrate this approach by using the refcnt file example.

Let b_j be the refcnt L_0 block that contains the i^{th} refcnt integer; clearly, $b_j \in \Psi(b_i)$. Then, the L_0 block of the refcnt file with the j^{th} integer, let's say b_k, belongs in $\Psi(b_j)$. This means that Iron would need to check b_j, b_k, and so on, possibly checking the entire refcnt file to load b_i, which would result in unpredictable operational latencies. As described in Sec. 4.2, instead of loading and checking b_j, Iron increments the i^{th} claimed refcnt integer[4]. This breaks the recursion, and b_j is checked later.

Thus, the number of yet-to-be-checked blocks in $\Psi(b)$ is quite small in practice. The analysis in this section can be used to prove freedom from deadlocks even when essential sets of concurrent client ops happen to overlap; we cannot present a formal proof due to lack of space.

4.4 Repairing Manifest Corruption

Depending on the type of metadata, Iron chooses from two techniques—tombstoning and quarantining—to handle manifest corruption.

4.4.1 Tombstoning Primary Metadata

Sec. 3.1 explains why damaged primary metadata in WAFL cannot be repaired, which is true with offline repair as well. Before making the file system available to clients, Iron checks the higher part of the file system tree hierarchy, such as the superblock, the inodefile blocktree, and the root directory. Iron aborts if corruption is detected there; the file system is not considered repairable. The customer can then choose between restoration from a recent snapshot (stored locally or remotely) or manual stitch-up of the file system by skilled technicians with direct access to the storage media.

Data structures in the lower part of the file system tree hierarchy with manifest corruptions are *tombstoned* by setting them to a corresponding zero value or to a special value that WAFL code paths recognize. If the whole block is unreliable, its entire content is tombstoned. A client read op that encounters it—say, a tombstoned child pointer in an indirect block of a user file—returns an appropriate error. A subsequent mutation can change the tombstone to a legal value. For example, a client op that writes to an offset in that file corresponding to the range covered by that child pointer replaces the tombstone. Tombstoning a child pointer results in an orphaned sub-tree, which is eventually recovered and placed into lost+found by Iron. Much as in traditional repair, the administrator can choose to stitch it back into the file system, but in concert with the application accessing it. If the administrator chooses otherwise, the data structures remain tombstoned until they are overwritten or deleted by new mutations. Given Rule #2, and that WAFL eschews redundancy within primary metadata (to avoid the associated performance overhead), we conclude:

Conclusion 1. *The repair of manifest corruption in primary metadata of WAFL by offline repair is no better than repair by Iron.*[5]

4.4.2 Quarantining Derived Metadata

If Iron encounters manifest corruption in a derived data structure, it *quarantines* the data structure by setting it to a corresponding well-known and conservative value that protects the resource that it tracks. The well-known value never overflows or underflows, which allows WAFL code paths to navigate past invariants that use it. If an entire block of derived metadata is deemed to be unreliable, then every data structure in it is quarantined. On completion, all quarantined structures are set to their corresponding values computed by Iron. Thus, given that all damage to derived metadata is quarantined before consulted by file system operations, and that the quarantined value conservatively protects the resource that it tracks, we conclude:

Conclusion 2. *Iron guarantees that mutations to the file system can never cause new or additional corruption due to existing manifest corruption in derived metadata.*

[4] Because WAFL is a COW file system, a claimed refcnt increment results in that claimed refcnt L_0 getting written to a new location, let's say block b_n, which in turn requires the n^{th} claimed refcnt integer to be incremented, and so on. This is a different type of recursion that impacts most allocation bitmaps in WAFL. The WAFL block allocator finds free blocks colocated in the block number space, and therefore this recursion converges very quickly. Previous publications [32, 33] detail how recursion during decrements (due to frees) converge.

[5] Offline repair in a file system with redundancy in primary metadata could stitch an orphaned subtree back into its correct location, repair the damaged child pointer, and avoid data loss. Online repair for such a file system would need to suspend client access to the tombstoned structure until the orphaned subtree is found.

The following example helps illustrate this conclusion. Let's say that Iron determines a refcnt file L_0 to be unreliable when it is first loaded, and let's say that L_0 stores refcnt integers for blocks b_i through $b_{(i+n-1)}$ of the file system. Iron then sets each of those refcnt integers in the L_0 to the quarantined value, thereby ensuring all potential references to blocks b_i to b_{i+n-1} are conservatively protected. In other words, the WAFL write allocator considers them unavailable for new mutations; note that WAFL uses COW, and no block is ever written in place. On completion, Iron resets each quarantined integer to its claimed refcnt counterpart and returns any unused blocks back to the free space in the file system.

Given sufficient damage to a specific derived metadata, Iron might decide that the file system has run out of the resource tracked by that derived metadata. Sec. 4.6 describes how this case is handled.

4.5 Repairing Latent Corruption

As Sec. 3.2 explained, a latent corruption is a violation of some specific file system invariant, and is detected when a code-path trips on it. We reason about latent corruption by discussing the different permutations of metadata that are involved in the violated invariant.

Primary metadata only: Let's say that all the blocks involved in the violated invariant are of primary metadata. In WAFL, all relationships between primary metadata are captured in $\Psi(b)$ for a given primary metadata block b. In the example from Sec. 3.2, a client op can access the damaged inode only after accessing that directory. Therefore, a violation of such an invariant is conveniently detected as and when each primary metadata block is loaded, which leads to:

Conclusion 3. *The offline repair of a latent corruption that violates an invariant across primary metadata of WAFL can be no better than repair by Iron.*

Derived and derived/primary metadata: Derived metadata typically track persistent resources consumed by the file system, such as inodes and blocks. We explore this problem for blocks, and then extend the results to other derived metadata. The refcnt integer r_i tracks the consumption of the i^{th} block by the file system. Thus, r_i encodes a relationship with block b_i; b_i may be user data or metadata (primary or derived).

Let's say that a latent corruption had made r_i incorrect. As described next, several mutations might be persisted to the file system before this corruption is eventually detected. WAFL relies exclusively on the child pointer in b_i's parent block when it frees b_i (say due to a file truncation) and decrements r_i. On the other hand, the WAFL write allocator relies exclusively on r_i to check if block b_i is free. Thus, code paths that allocate and free blocks depend exclusively on derived and primary metadata, respectively, and expect them to be consistent. This split-brain behaviour can result in the morphing of this latent corruption even before it is detected. The corrupted r_i might be (A) higher or (B) lower than the true value. If (A), WAFL might eventually leak b_i when all references to it have been dropped and r_i remains non-zero. The leaked block will be detected by a subsequent run of Iron. Two possibilities exist with (B): In case (B1): an eventual decrement causes r_i to underflow, which is detected as a violation. In case (B2), the WAFL write allocator might incorrectly assign b_i to a new write when r_i becomes zero, and the original contents of b_i are lost to the file system. If b_i originally contained metadata, any access through an older reference would detect manifest corruption (signature and context mismatch)[6]. In this case as well as case (B1), the file system is marked as inconsistent, is taken offline, and repair is invoked.

Conclusions 1 and 2 show that Iron handles the manifest corruption of case (B2) no worse than offline repair does. In case (B1), if the decrement has been triggered by a client op, Iron increments the i^{th} claimed refcnt integer almost immediately after mount because WAFL replays all client ops after any disruption. Thus, the subsequent decrement finds a zero value r_i but a nonzero claimed refcnt integer. Iron prevents any file system activity from underflowing a refcnt integer as long as it can decrement the corresponding claimed refcnt integer. If the decrement has not been triggered by a client op, b_i remains unclaimed until Iron gets to the file that refers to it. During this window, both r_i and its claimed refcnt counterpart are zero, and the WAFL write allocator may use b_i for a new block; that scenario is subsumed by case (B2).

Although offline repair averts the previously mentioned window because no new blocks are being written to the file system, it should be noted that case (B2) may also occur during runtime before the latent corruption is detected and recovery is initiated. Thus, in practice, the use of Iron does not introduce significant additional risk beyond what existed earlier.

This entire argument can be replicated for any resource that is similarly tracked by derived metadata and tracked separately by Iron shadow metadata. Latent corruption in derived metadata that is checked before client access is allowed to the file system can be found and repaired early on. This leads to:

Conclusion 4. *The repair by Iron of latent corruption*

[6] If b_i contained user data, it is lost. Independent of whether b_i contained user data or metadata, its re-allocation does not create a security risk because access via the original parent of b_i will fail the context check, and return an error instead of the new content stored in b_i.

that violates a relationship across derived and/or primary metadata is no worse than that by offline repair.

Miscellaneous metadata: WAFL maintains extensive auxiliary metadata that are computed using information from other derived metadata. Such auxiliary metadata typically are used to enable specific features or better file system performance. The WAFL file system can typically survive corruption to such metadata, but when Iron is invoked, it can repair these structures while the file system runs with decreased performance or with those specific features disabled. We have found that customers are willing to tolerate such temporal deficiencies for continued data availability.

4.6 Running Out of a Resource

The end of Sec. 4.4.2 discussed a problem scenario that relates to the quarantining of a sufficiently large amount of derived metadata. It might result in premature exhaustion of the file system resource tracked by that metadata, before Iron completes. For example, sufficient quarantining of the refcnt metadata might cause the file system to run out of space. The WAFL block allocator is designed to offline the file system gracefully in this case. Because the file system is still marked as corrupt (Iron never completed), the file system is now repaired by using Iron in offline mode (more information in Sec. 6). It is important to note that no mutations are lost in this scenario. To the best of our knowledge, this problem has not been encountered in the field thus far.

4.7 Three Phases of Iron

[1] Mount: ONTAP mounts the inconsistent WAFL file system when it is brought online with the Iron option. To allow faster access to clients, Iron limits the amount of metadata that is checked at mount. As described in Sec. 4.4.1, key metadata in the upper part of the file system tree hierarchy are checked. Based on the physical storage devices, various limits on file system resources are computed (such as number of blocks) and are used as ceilings for various global counters. Auxiliary metadata, such as hints for speeding up the search for free space, are checked. Based on those hints, the block allocator is primed by prefetching refcnt file blocks. Detection of manifest corruption results in the quarantining of refcnt integers and further loading of refcnt blocks until sufficient free space has been confirmed. The Iron status files are created and updated to reflect the checking performed thus far. As described in Sec. 4.4.1, Iron aborts if mount-time checks do not complete. Otherwise, client access is allowed.

[2] File system scan: Metadata are checked on-demand (based on client access) and through background scans, each of which selects an inode and walks its blocktree. Leaf nodes of metafiles are also checked. Progress indicator status files ensure that each block is checked once.

As client mutations are processed, the WAFL write allocator prefetches more blocks of the refcnt file to find more free space. The background walk of crucial derived metadata, such as the refcnt file, typically completes early, and all quarantining that affects free space accounting is in place. Recall that Iron status files track both validated and new data written by clients, so the validated portion of the file system continually increases. Due to space constraints, we do not describe our implementation in more detail.

[3] Completion: After the entire file system has been walked, the derived shadow metadata—which, now accurately represent all resource consumption—are swapped with their counterparts; quarantined structures are removed. Status files are deleted subsequently and the file system is marked as consistent.

5 Analysis and Some History

This section describes some deficiencies in the initial version of Iron and some improvements that were made over the years.

[1] New corruption: Regular file system access is allowed during online repair, which means that if the file system was originally corrupted by a software bug, it could reoccur during the repair process. Therefore, it is difficult for any efficient online repair tool to *guarantee* file system consistency on completion. Earlier versions of Iron also have this "flaw". Although it has never been observed, it is possible for the Iron status metadata to be corrupted by such a bug, which might have a bigger impact on the guarantees that Iron provides. Sec. 5.1 addresses this issue.

[2] Mount-time performance: The first version of Iron (circa 2003) checked the indirect blocks of the blocktrees of all derived metadata during mount. However, this meant longer mount times, and therefore a longer wait for restoration of client access to the file system. The mount phase was subsequently thinned, and larger derived metadata (that scale linearly with file system size) are now checked asynchronously post mount. Quarantining occurs at any level of the indirect blocktree of a derived metafile. Latent corruption in derived metadata is addressed by the hardening techniques of Sec. 5.1.

[3] Performance of client ops: In its original version, Iron checked the entire indirect blocktree of a given in-

ode before a client (or internal) op could access any block of that file. Thus, the original definition of $\Psi(b)$ included all indirect blocks in the blocktree of any inode in the ancestory hierarchy of b, thereby ensuring Rule #2 when exposing file attribute state to clients, such as size or block count. In its early days, WAFL was primarily optimized for homedir-style engineering workloads with many small to medium-sized files, and so the time to first-access of a file was not too significant. With the deployment of critical database and virtualization workloads on WAFL, GiB- and TiB-sized files became increasingly common. Irreversibility of attributes such as block count for files that host databases or VM disks is not a strict requirement. Thereafter, the definition of $\Psi(b)$ was refined (to that in Sec. 4.1) to exclude non-ancestor blocks of b in the file system tree, but without any risk to the repair process. Sec. 5.2 describes additional performance improvement.

5.1 WAFL Metadata Integrity Protection

Two techniques, incremental checksums and digest-based auditing [34], were introduced circa 2012 to protect much of the WAFL file system metadata from memory scribbles and logic bugs [23]. Sec. 7.5 of [34] shows the resultant drop in corruption incidents in customer systems, thereby dramatically reducing the need for Iron. In addition, there are two crucial implications for Iron: (1) Iron status files are now protected by these techniques, which squarely addresses the first deficiency described in Sec. 5; (2) it removes one key reason for the on-demand update of shadow derived metadata while processing client ops; more in the next section.

5.2 Lazy Block Claiming (LBC)

After mount-time outages were reduced, the one important remaining problem with Iron was the impact to client ops. As explained earlier, Iron must process the corresponding essential set before a file system op can be given access to a block. Because recursivity in derived metadata has been solved, the cost of processing any block in Ψ is dominated by: (1) loading and consulting/updating checked bitmap blocks, and (2) loading and updating claimed refcnt blocks. These steps require additional CPU cycles and random I/Os to storage; the randomness is also a function of client access patterns. *Lazy Block Claiming (LBC)* was introduced to address this overhead.

The on-demand update of claimed refcnt metadata (or any derived metadata) guarantees that resources accessed by a client op are henceforth protected, thereby preserving Rule #2. Thus, when a client op accesses block b_i, the on-demand increment of the i^{th} claimed refcnt integer protects b_i from being re-allocated due to a corrupted refcnt integer. Latent corruption in the refcnt metadata is eliminated by the integrity techniques of Sec. 5.1. And, given that manifest corruption results in quarantining, Rule #2 is now preserved even without on-demand updates of derived metadata, such as claimed refcnts. Thus, LBC avoids the afore-mentioned costs and enables improved client performance, independent of file size.

This means, after the file system is mounted, the claiming of the references to each block in the file system occurs only through the background scans. More importantly, knobs are provided that control the speed of those background scans. Thus, the customer can choose between reducing the impact of Iron scans to client workloads and how quickly Iron completes processing the entire file system.

5.3 Additional Enhancements to Iron

This section outlines in-progress and productized improvements; a future paper will cover them. First, concurrent access of Iron status files within the WAFL parallelism model [17] is required to truly minimize the impact of Iron on client performance. Second, the customer still experiences outage from the time the WAFL aggregate is offlined until Iron is started. Incremental Autoheal Iron [11] builds on the principles described in Sec. 4.1 to provide true zero disruption. When Autoheal detects a corruption at runtime, it tombstones or quarantines, simulates a minimal emptying of the buffer cache, and optionally kicks off a background scan to check and repair a defined set of metadata based on the corruption. Depending on the results of the scan, a larger subset of the metadata might be scanned next. Such incremental granular repair is ideal because, as mentioned earlier, only a few metadata blocks are typically damaged in WAFL. ONTAP 9.1 introduced FlexGroup technology [7, 42]—it allows a file system to span multiple physical nodes in a cluster. Offlining an entire FlexGroup on the detection of a corruption is obviously not an option; ONTAP 9.1 includes an early version of Autoheal.

6 Topics in Practice

This section presents some selected topics that relate to the implementation of Iron.

Location of status files: The implementation allows for Iron status files to be stored within the file system being repaired or in a different WAFL file system; both choices are equally safe. By storing it remotely, the customer can isolate to a separate set of storage devices the extra I/Os

required to read and update status files.

Offline mode for Iron: In this mode, Iron provides WAFLCheck-like behaviour. Thus, client access is disallowed and Iron cannot write to the physical storage of the file system. Iron stores its status files remotely. Corruptions that Iron fixes are appended as a sequence of tuples to a log file stored in a different file system. Each tuple includes the contents of the file system block and its physical location that the WAFL write allocator chose. Eventually, the administrator is given the choice to commit all or none of the repair. If the former choice is made, the log file is "replayed" and the content of the log is written out at the appropriate locations in the file system.

Aggregates and FlexVols: ONTAP hosts and exports hundreds of FlexVol® volumes on a shared pool of of physical storage called an *aggregate* [20]. Each FlexVol and aggregate is a WAFL file system. When corruption is detected, the aggregate file system is tagged as corrupt, is offlined, and is eventually remounted with Iron. Hundreds of applications hosted on the FlexVols gain early access to their data. Our field data show that typically a handful of blocks from a few FlexVols are damaged. At worst, a few of the applications might be halted if they access tombstoned structures; they can be restarted after that data is recovered from backup or from lost+found. However, other applications see minimal disruption.

Field data: In an analysis of corruptions seen across ~250,000 customer systems during a recent six-month period, approximately a third were attributed to software bugs, another third to media errors (while RAID was running in degraded mode), and the rest to a mix of manual configuration error or unknown reasons. In each case, the total number of corrupted metadata blocks was less than 10. In very rare cases when hundreds of blocks are damaged (due to silent hardware failures), customers typically restore from backup/snapshots or use offline repair.

7 Evaluation

In this section, we present the performance characteristics of Iron. As explained earlier, a WAFL file system being repaired has at worst tens of damaged metadata blocks. The extra cost of repairing those blocks is undetectable compared with the cost of checking the entire file system.. Therefore, no actual corruption is required in the datasets of the following experiments. We discuss client outage times, the overhead of running Iron, and how it interferes with a real-world workload. Unless otherwise mentioned, all experiments were conducted on a lower-end system with 16 Intel Sandy Bridge cores and 64 GiB DRAM to accentuate the impact of Iron.

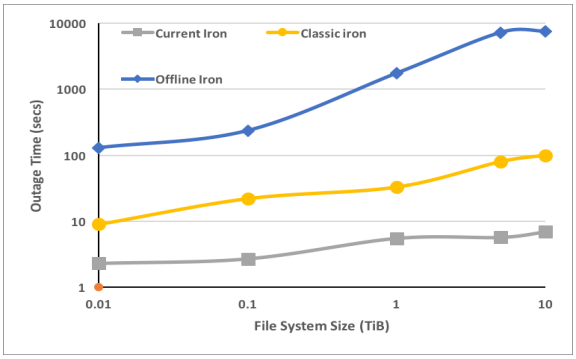


Figure 1: Client outage time in seconds on a logarithmic scale with increasing file system size.

7.1 Memory and Storage Overhead

Iron metafiles are paged in and out of the buffer cache like any other file in the file system. The storage space that the metafiles consume is approximately 32 MiB (checked bitmap) and 0.5 GiB (claimed refcnt) per TiB of file system size, and is 4 MiB (link count) per million inodes in the file system. Together with other metafiles (not presented in this paper), it adds up to around 0.05% of the file system size. The in-memory data structures that Iron uses add up to a few KiB of extra memory. This extra requirement in memory and storage is negligible on all configurations of ONTAP.

7.2 Outage Time with Iron

Fig. 1 plots on a logarithmic scale client outage time with increasing file system size (used space) on the lower-end system. Outage was measured from when the command to online the WAFL file system (with the Iron option) was issued to when the file system was first exposed to clients. Thus, this experiment measured the time taken to check metadata during mount. Numerous drives were used to avoid I/O bottlenecks, and no other load was applied on the system. The experiment was run thrice: with Iron in offline mode, with classic Iron (mentioned in Sec. 5) in which derived metadata were checked during mount, and with the current version of Iron in which indirect blocks of all large derived metafiles were checked post-mount. Iron employs the same level of parallelism for checking metadata in each mode, thereby ensuring fairness.

Iron in offline mode performs similarly to the now-obsolete WAFLCheck tool, and outage time is obviously the time to complete Iron. Offline repair is clearly impractical for enterprises—a 10 TiB file system takes 2+ hours to repair. Outage times with the current version of Iron are an order of magnitude less than the times reported by classic Iron; 6.9s and 100s, respectively, for 10

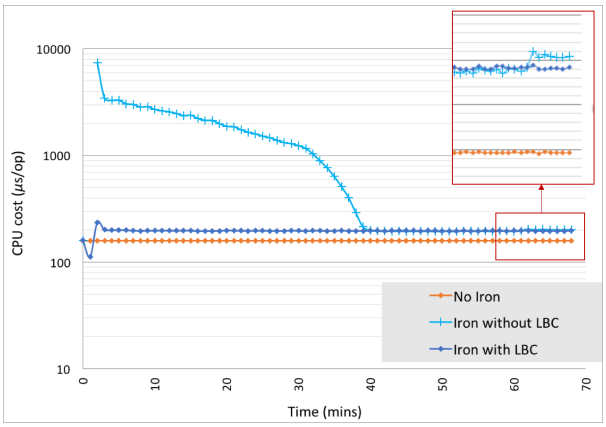


Figure 2: CPU cost (μs/op) of random reads

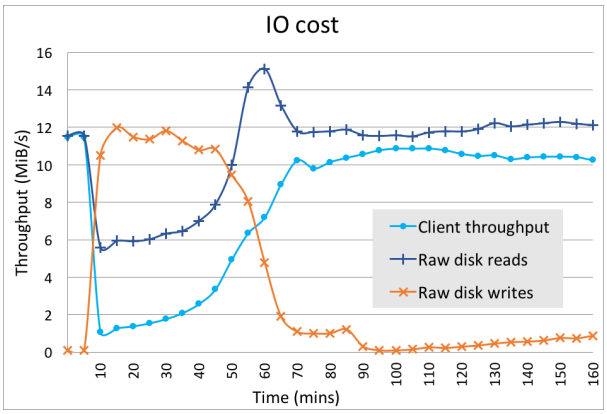


Figure 3: Throughput in MiB/s when random reads are directed to a repairing (without LBC) file system

TiB. Outage times with the current Iron tool are almost independent of the file system size; mount without Iron (not shown) takes around 1s for any file system size.

7.3 Performance Overhead of Iron

Next, we present the overhead associated with Iron in terms of CPU cycles and storage I/Os—the two important metrics that IT architects use for sizing storage systems and applications. LBC was instrumental in making Iron's overhead more predictable and therefore practical. The worst case overhead on low-end systems is 25%. Note that most of our systems do not experience that level of overhead. We used a worst-case random read workload—it reduced the overlap between the essential set for a given client read with that of a previous one—thereby maximizing the amount of on-demand work that Iron performs. A read-only workload was used to keep the analysis simple; there was no material change in the results when client writes were added in.

7.3.1 Cost in CPU Cycles

Several NFS clients directed a random read workload of 25 MiB/s to a 18 TiB dataset that comprised 450 files, each of size 40 GiB on the lower-end system. To avoid perturbation from I/O bottlenecks, the storage was all-SSD. The experiment was run without Iron, with Iron and LBC disabled, and with current Iron (LBC enabled). To make the comparison fair, the background Iron scan was disabled for the first one hour of the run without LBC—so all Iron work was triggered only on-demand by the client reads.

CPU cost is computed by adding up all the cycles that the file system code paths (including Iron) use and dividing that value by the number of client operations serviced for a given time interval. Fig. 2 compares that cost (measured as μs/op) on a logarithmic scale. The baseline read op cost averages 160 μs' worth of CPU cycles; it includes the processing for misses in the buffer cache.

In the Iron run without LBC, about 4 to 8 primary metadata blocks are checked for each client read in the early portion of the experiment. Each checked indirect block may have up to 256 children that are unlikely to be colocated, and requires random updates to the claimed refcnt metafile. In the early portion of the experiment, this costs an extra 7.8 ms' worth of cycles. Over time, a client read finds that more of its essential set is already checked, and the Iron overhead drops. Almost all metadata has been checked by the 40-min mark, and the cost flattens to about 190 μs; the extra 30 μs is the unavoidable cost of consulting the checked bitmap. The small bump at the 60-min mark coincides with the start of the background scan (which completes soon after because on-demand checking has already done the job). With LBC, the overhead of random I/Os to the claimed refcnt metafile is moved from client ops to the slow-running background scan. Thus, after the initial spike to check important metadata, such as inodes and directory entries, the cost flattens to 200 μs; the still-running background scan costs the extra 10 μs.

7.3.2 Cost in I/O Bandwidth

The previous experiment was modified to use SATA hard drives (so storage I/Os were no longer "cheap"), and the client load was lowered to 11.5 MiB/s which is commensurate with the I/O capability of that storage media.

Fig. 3 shows the client throughput and the raw reads and writes to the drives in MiB/s. To simplify the analysis, Iron without LBC was started at the 10-min mark of the experiment, and the background walk of the file system was disabled (until the 110-min mark). The client

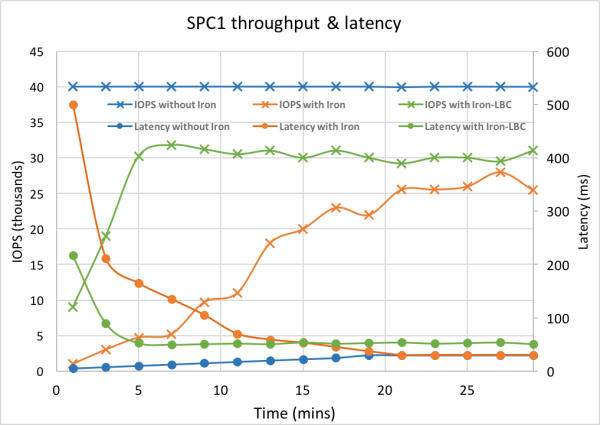


Figure 4: Impact of Iron with and without LBC on latency (right-side y-axis) and on throughput (left-side y-axis) at a steady applied load of 40k SPC-1 IOPS.

throughput and raw reads are identical until the 10-min mark. The remount (to start Iron) at the 10-min mark empties the buffer cache. This is followed by a spike in writes to storage as Iron status metafiles undergo frequent updates. The remaining disk bandwidth is divided between reading the user inode leaf nodes and Iron status files. Thus, for the first 40 mins of Iron only about 10% to 18% of the disk bandwidth is used for reading the leaf nodes of user files. By 90 mins the essential set for most client reads has already been checked, but checked bitmap consultations require a continual and fixed amount of read bandwidth. The impact due to background scans is seen after the 110-min mark. With LBC enabled (not shown here), the drop in client throughput is mostly a function of the rate at which Iron background scans run, which is typically set to a low default.

7.4 Impact on Clients

Several clients were used to apply a steady load of reads/writes to model the query/update ops of a database/OLTP application; this was based on the Storage Performance Council Benchmark-1 (SPC-1) [16]. Iron was started soon after. LBC was designed primarily for helping database/OLTP applications, which are quite latency-sensitive to any additional CPU or I/O overhead. To accentuate the impact of Iron, the experiment was configured on a low-end system with 8 AMD Opteron cores, 32 GiB DRAM, and SATA HDDs. The background Iron scans were allowed to run in this experiment.

Fig. 4 shows that without Iron the clients achieve the entire applied throughput of 40k IOPS with average latencies under 30 ms. With Iron, both metrics improve as a larger portion of the metadata is checked. With LBC, these metrics are 2 to 5 times better early on, and they soon converge to a steady 75% of the applied throughput. In theory, WAFL parallelism should be unaffected by Iron because checking (both on-demand and by way of scan) can run concurrently with other client operations in the WAFL MP model [17]. However, one last project to achieve full parallelism is still in progress, after which we expect the impact of Iron (with LBC) to be much smaller. Because the background scans limit some of the potential parallelism, client operations in the run with LBC show poorer latencies past the half-way point. In the run without LBC, the on-demand work has finished much of the scan's job by that point. The slow increase in latency in the baseline run (until 20 min) is due to the initial aging of the file system.

Many of our customer systems use SSDs and are not low-end, and therefore see less impact with Iron. As mentioned earlier, Iron is run under the supervision of NetApp support, and customers are aware that an inconsistent file system is being recovered. We find that they greatly appreciate the continued uptime for their applications, even with reduced performance. As improvements to Iron have reduced the impact to clients over the years, we also find that customers have become less concerned with Iron completion times as long as progress indicators provide a time estimate. But, as an example, Iron completion time with a default scan speed on the lower-end system is 0.48 hour per TiB dataset resident on SSDs and 1 hour per TiB dataset resident on hard drives, even while sustaining a random-read client workload of 470 MiB/s and 255 MiB/s, respectively. Impact on home-directory style workloads is not presented due to lack of space. The impact is typically less than that on database/OLTP workloads because the files themselves are small, and each indirect block has few children. However, datasets with very large directories (millions of entries) are affected to a greater extent; future work is planned to make the checking of directories truly asynchronous to client operations.

8 Conclusion

This paper explains the importance of online repair to enterprises. It explains how Iron provides the same quality of repair as offline repair does, even while allowing client access to the file system. It presents some implementation detail, history, and performance evaluation. To the best of our knowledge, this publication is the first to present fully functional enterprise quality online repair. A follow-up paper will present implementation details and the enhancements mentioned in Sec. 5.3. We thank the many WAFL engineers who contributed to Iron over the years; they are too many to list. We also thank our reviewers and shepherd for their invaluable feedback.

References

[1] Checking ZFS file system integrity. https://docs.oracle.com/cd/E23823_01/html/819-5461/gbbwa.html#scrolltoc.

[2] Disks and aggregates power guide. https://library.netapp.com/ecm/ecm_download_file/ECMLP2496263.

[3] How you schedule automatic raid-level scrubs. https://library.netapp.com/ecmdocs/ECMP1196912/html/GUID-A2F3A870-5C8D-4A68-AC8C-912946CECAC0.html.

[4] Kernel Bug Tracker. http://bugzilla.kernel.org/.

[5] Linux btrfs blog posts. http://marc.merlins.org/perso/btrfs/post_2014-03-19_Btrfs-Tips_-Btrfs-Scrub-and-Btrfs-Filesystem-Repair.html.

[6] Metrocluster for clustered data ontap 8.3.2. https://storageconsortium.de/content/sites/default/files/WP_NetApp%20Metrocluster%20for%20Clustered%20Data%20ONTAP%208.3.2.pdf.

[7] Scalability and performance using flexgroup volumes power guide. http://docs.netapp.com/ontap-9/index.jsp?topic=%2Fcom.netapp.doc.pow-fg-mgmt%2FGUID-A304BBC1-C00C-4E7A-989E-7C5A0E505146.html.

[8] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004.

[9] Robert Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, 2005.

[10] Steve Best. JFS Overview. http://www.ibm.com/developerworks/library/l-jfs.html, 2000.

[11] Sushrut Bhowmik, Vinay Kumar, Sreenath Korrakuti, Arun Pandey, and Sateesh Pola. Automatic incremental repair of granular filesystem objects. pending patent application.

[12] Eric J. Bina and Perry A. Emrath. A Faster fsck for BSD Unix. In *Proceedings of the USENIX Winter Conference*, January 1989.

[13] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[14] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proceedings of the fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 12–25, 1995.

[15] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2004.

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

[17] Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 419–434, 2016.

[18] Matthew Curtis-Maury, Ram Kesavan, and Mrinal K. Bhattacharjee. Scalable write allocation in the WAFL file system. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, August 2017.

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

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

[21] Daniel Fryer, Kuei Sun, Rahat Mahmood, Ting-Hao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *Proceedings of 10th USENIX Conference on File and Storage Technologies (FAST)*, February 2012.

[22] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of 1st USENIX Conference on Operating Systems Design and Implementation (OSDI)*, November 1994.

[23] Jim Gray. Why do computers stop and what can be done about it? Tandem Technical Report 85.7, June 1985.

[24] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-

Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[25] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, November 1987.

[26] Val Henson. The Many Faces of fsck. http://lwn.net/Articles/248180/, 2007.

[27] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing fsck time for ext2 file systems. In *Linux Symposium*, pages 395–407, 2006.

[28] 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 2nd Conference on Hot Topics in System Dependency (HotDep)*, 2006.

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

[30] Microsoft Inc. Building the next generation file system for windows: Refs. https://blogs.msdn.microsoft.com/b8/2012/01/16/building-the-next-generation-file-system-for-windows-refs/, 2012.

[31] NetApp Inc. Overview of wafliron. `https://kb.netapp.com/support/index?page=content\&id=3011877`, 2016.

[32] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Algorithms and data structures for efficient free space reclamation in wafl. In *15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[33] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. Efficient free space reclamation in WAFL. *ACM Transactions on Storage (TOS)*, 13, October 2017.

[34] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL copy-on-write file system. In *15th Usenix Conference on File and Storage Technologies (FAST)*, 2017.

[35] Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A memory soft error measurement on production systems. In *USENIX Annual Technical Conference (ATC)*, June 2007.

[36] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast file-system checker. *ACM Transactions on Storage (TOS)*, 10(1):2:1–2:28, January 2014.

[37] Joshua MacDonald, Hans Reiser, and Alex Zarochentcev. http://www.namesys.com/txn-doc.html, 2002.

[38] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Tranactions on Electron Devices*, 26(1), 1979.

[39] Marshall Kirk McKusick. Running 'fsck' in the Background. In *BSDCon '02*, 2002.

[40] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, 1986.

[41] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, 40(1):41–50, 1996.

[42] Justin Parisi. Netapp flexgroup volumes: An evolution of nas. `https://blog.netapp.com/blogs/netapp-flexgroup-volumes-an-evolution-of-nas/`.

[43] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD International Conference on Management of Data*, pages 109–116, June 1988.

[44] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1), 1992.

[45] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2007.

[46] Thomas J.E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D.E. Long, Andy Hospodor, and Spencer Ng. Disk Scrubbing in Large Archival Storage Systems. In *IEEE 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2004.

[47] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying File System Protection. In *Proceedings of USENIX Annual Technical Conference*, pages 79–90, June 2001.

[48] Rajesh Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html, 2006.

[49] Stephen C. Tweedie. Journaling the Linux ext2fs

File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, 1998.

[50] Wikipedia. Btrfs. en.wikipedia.org/wiki/Btrfs, 2009.

[51] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference (FSE)*, pages 327–336, September 2003.

[52] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A ZFS case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[53] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices

Arash Tavakkol[†], Juan Gómez-Luna[†], Mohammad Sadrosadati[†], Saugata Ghose[‡], Onur Mutlu[†‡]
[†]*ETH Zürich* [‡]*Carnegie Mellon University*

Abstract

Solid-state drives (SSDs) are used in a wide array of computer systems today, including in datacenters and enterprise servers. As the I/O demands of these systems continue to increase, manufacturers are evolving SSD architectures to keep up with this demand. For example, manufacturers have introduced new high-bandwidth interfaces to replace the conventional SATA host–interface protocol. These new interfaces, such as the NVMe protocol, are designed specifically to enable the high amounts of concurrent I/O bandwidth that SSDs are capable of delivering.

While modern SSDs with sophisticated features such as the NVMe protocol are already on the market, existing SSD simulation tools have fallen behind, as they do not capture these new features. We find that state-of-the-art SSD simulators have three shortcomings that prevent them from accurately modeling the performance of real off-the-shelf SSDs. First, these simulators do not model *critical features of new protocols* (e.g., NVMe), such as their use of multiple application-level queues for requests and the elimination of OS intervention for I/O request processing. Second, these simulators often do not accurately capture the impact of advanced SSD maintenance algorithms (e.g., garbage collection), as they do not properly or quickly emulate *steady-state conditions* that can significantly change the behavior of these algorithms in real SSDs. Third, these simulators do not capture the full *end-to-end latency* of I/O requests, which can incorrectly skew the results reported for SSDs that make use of emerging non-volatile memory technologies. By not accurately modeling these three features, existing simulators report results that *deviate significantly* from real SSD performance.

In this work, we introduce a new simulator, called MQSim, that *accurately* models the performance of both modern SSDs and conventional SATA-based SSDs. MQSim faithfully models new high-bandwidth protocol implementations, steady-state SSD conditions, and the full end-to-end latency of requests in modern SSDs. We validate MQSim, showing that it reports performance results that are only 6%-18% apart from the measured actual performance of four real state-of-the-art SSDs. We show that by modeling critical features of modern SSDs, MQSim uncovers several real and important issues that were not captured by existing simulators, such as the performance impact of inter-flow interference. We have released MQSim as an open-source tool, and we hope that it can enable researchers to explore directions in new and different areas.

1 Introduction

Solid-state drives (SSDs) are widely used in today's computer systems. Due to their high throughput, low response time, and decreasing cost, SSDs have replaced traditional magnetic hard disk drives (HDDs) in many datacenters and enterprise servers, as well as in consumer devices. As the I/O demand of both enterprise and consumer applications continues to grow, SSD architectures are rapidly evolving to deliver improved performance.

For example, a major innovation has been the introduction of new host interfaces to the SSD. In the past, many SSDs made use of the Serial Advanced Technology Attachment (SATA) protocol [67], which was originally designed for HDDs. Over time, SATA has proven to be inefficient for SSDs, as it cannot enable the fast I/O accesses and millions of I/O operations per second (IOPS) that contemporary SSDs are capable of delivering. New protocols such as NVMe [63] overcome these barriers as they are designed specifically for the high throughput available in SSDs. NVMe enables high throughput and low latency for I/O requests through its use of the *multi-queue SSD* (MQ-SSD) concept. While SATA exposes only a single request port to the OS, MQ-SSD protocols provide *multiple* request queues to *directly expose* applications to the SSD device controller. This allows (1) an application to bypass OS intervention for I/O request processing, and (2) the SSD controller to schedule I/O requests based on how busy the SSD's resources are. As a result, the SSD can make higher-performance I/O request scheduling decisions.

As SSDs and their associated protocols evolve to keep pace with changing system demands, the research community needs simulation tools that reliably model these new features. Unfortunately, state-of-the-art SSD simulators do *not* model a number of key properties of modern SSDs *that are already on the market*. We evaluate several real modern SSDs, and find that state-of-the-art simulators do *not* capture three features that are critical to *accurately* model modern SSD behavior.

First, these simulators do not correctly model the *multi-queue approach used in modern SSD protocols*. Instead, they implement only the single-queue approach used in HDD-based protocols such as SATA. As a result, existing simulators do not capture (1) the high amount of request-level parallelism and (2) the lack of OS intervention in modern SSDs.

Second, many simulators do not adequately model *steady-state behavior* within a reasonable amount of simulation time. A number of fundamental SSD maintenance algorithms, such as garbage collection [11–13, 23], are *not* executed when an SSD is new (i.e., no data has been written to the drive). As a result, manufacturers design these maintenance algorithms to work best when an SSD reaches the steady-state operating point (i.e., after all of the pages within the SSD have been written to at least once) [71]. However, simulators that cannot capture steady-state behavior (within a reasonable

simulation time) perform these maintenance algorithms on a new SSD. As such, many existing simulators do *not* adequately capture algorithm behavior under realistic conditions, and often report unrealistic SSD performance results (as we discuss in Section 3.2).

Third, these simulators do not capture the full *end-to-end latency* of performing I/O requests. Existing simulators capture only the part of the request latency that takes place during intra-SSD operations. However, many emerging high-speed non-volatile memories greatly reduce the latency of intra-SSD operations, and, thus, the uncaptured parts of the latency now make up a significant portion of the overall request latency. For example, in Intel Optane SSDs, which make use of 3D XPoint memory [9, 25], the overhead of processing a request and transferring data over the system I/O bus (e.g., PCIe) is much higher than the memory access latency [16]. By not capturing the full end-to-end latency, existing simulators do not report the true performance of SSDs with new and emerging memory technologies.

Based on our evaluation of real modern SSDs, we find that these three features are essential for a simulator to capture. Because existing simulators do not model these features adequately, their results deviate significantly from the performance of real SSDs. **Our goal** in this work is to develop a new SSD simulator that can *faithfully* model the features and performance of both modern multi-queue SSDs and conventional SATA-based SSDs.

To this end, we introduce *MQSim*, a new simulator that provides an accurate and flexible framework for evaluating SSDs. MQSim addresses the three shortcomings we found in existing simulators, by (1) providing detailed models of both conventional (e.g., SATA) and modern (e.g., NVMe) host interfaces; (2) accurately and quickly modeling steady-state SSD behavior; and (3) measuring the full end-to-end latency of a request, from the time an application enqueues a request to the time the request response arrives at the host. To allow MQSim to adapt easily to future SSD developments, we employ a modular design for the simulator. Our modular approach allows users to easily modify the implementation of a single component (e.g., I/O scheduler, address mapping) without the need to change other parts of the simulator. We provide two execution modes for MQSim: (1) standalone execution, and (2) integrated execution with the gem5 full-system simulator [8]. We validate the performance reported by MQSim using several real SSDs. We find that the response time results reported by MQSim are very close to the response times of the real SSDs, with an average (maximum) error of only 11% (18%) for real storage workload traces.

By faithfully modeling the major features found in modern SSDs, MQSim can uncover several issues that existing simulators are unable to demonstrate. One such issue is the performance impact of *inter-flow interference* in modern MQ-SSDs. For two or more concurrent *flows* (i.e., streams of I/O requests from multiple applications), there are three major sources of interference: (1) the write cache, (2) the mapping table, and (3) the I/O scheduler. Using MQSim, we find that inter-flow interference leads to significant *unfairness* (i.e., the interference slows down each flow unequally) in modern SSDs. This is a major concern, as fairness is a first-class design goal in modern computing platforms [4, 17, 19, 31, 37, 56–60, 66, 73–76, 80, 84, 88]. Unfairness reduces the predictability of the I/O latency and throughput for each flow, and can allow a malicious flow to deny or delay I/O service to other, benign flows.

We have made MQSim available as an open source tool to the research community [1]. We hope that MQSim enables researchers to explore directions in several new and different areas.

We make the following key contributions in this work:

- We use real off-the-shelf SSDs to show that state-of-the-art SSD simulators do *not* adequately capture three important properties of modern SSDs: (1) the multi-queue model used by modern host–interface protocols such as NVMe, (2) steady-state SSD behavior, and (3) the end-to-end I/O request latency.
- We introduce MQSim, a simulator that accurately models both modern NVMe-based and conventional SATA-based SSDs. To our knowledge, MQSim is the first publicly-available SSD simulator to faithfully model the NVMe protocol. We validate the results reported by MQSim against several real state-of-the-art multi-queue SSDs.
- We demonstrate how MQSim can uncover important issues in modern SSDs that existing simulators *cannot* capture, such as the impact of inter-flow interference on fairness and system performance.

2 Background

In this section, we provide a brief background on multi-queue SSD (MQ-SSD) devices. First, we discuss the internal organization of an MQ-SSD (Section 2.1). Next, we discuss host–interface protocols commonly used by SSDs (Section 2.2). Finally, we discuss how the SSD flash translation layer (FTL) handles requests and performs maintenance tasks (Section 2.3).

2.1 SSD Internals

Modern MQ-SSDs are typically built using NAND flash memory chips. NAND flash memory [11, 12] supports read and write operations at the granularity of a flash *page* (typically 4 kB). Inside the NAND flash chips, multiple pages are grouped together into a flash *block*, which is the granularity at which erase operations take place. Flash writes can take place only to pages that are erased (i.e., *free*). To minimize the write latency, MQ-SSDs perform *out-of-place* updates (i.e., when a logical page is updated, its data is written to a different, free physical page, and the logical-to-physical mapping is updated). This avoids the need to erase the old physical page during a write operation. Instead, the old page is marked as *invalid*, and a *garbage collection* procedure [11–13, 23] reclaims invalid physical pages in the background.

Figure 1 shows the internal organization of an MQ-SSD. The components inside the MQ-SSD are divided into two groups: (1) the *back end*, which includes the memory devices; and (2) the *front end*, which includes the control and management units. The memory devices (e.g., NAND flash memory [11, 12], phase-change

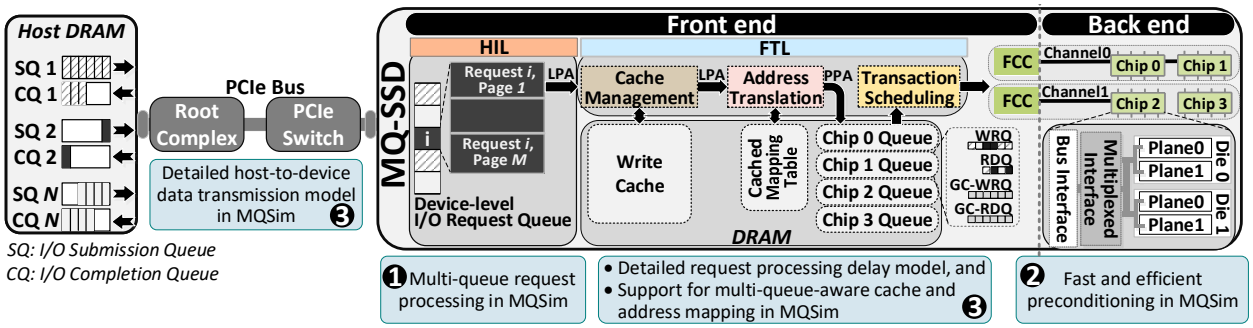


Figure 1: Organization of an MQ-SSD. As highlighted in the figure (❶, ❷, ❸), our MQSim simulator captures several aspects of MQ-SSDs not modeled by existing simulators.

memory [42], STT-MRAM [40], 3D XPoint [9]) in the back end are organized in a highly-hierarchical manner to maximize I/O concurrency. The back end contains multiple independent *bus channels*, which connect the memory devices to the front end. Each channel connects to one or more memory *chips*. For a NAND flash memory based SSD, each NAND flash chip is typically divided into multiple *dies*, where each die can independently execute memory commands. All of the dies within a chip share a common communication interface. Each die is made up of one or more *planes*, which are arrays of flash cells. Each plane contains multiple blocks. Multiple planes within a single die can execute memory operations in parallel *only* if each plane is executing the same command on the same address offset within the plane.

In an MQ-SSD, the front end includes three major components [47]. (1) The *host–interface logic* (HIL) implements the protocol used to communicate with the host (Section 2.2). (2) The *flash translation layer* (FTL) manages flash resources and processes I/O requests (Section 2.3). (3) The *flash chip controllers* (FCCs) send commands to and transfer data to/from the memory chips in the back end. The front end contains on-board DRAM, which is used by the three components to cache application data and store data structures for flash management.

2.2 Host–Interface Logic

The HIL plays a critical role in leveraging the internal parallelism of the NAND flash memory to provide higher I/O performance to the host. The SATA protocol [67] is commonly used for conventional SSDs, due to widespread support for SATA on enterprise and client systems. SATA employs Native Command Queuing (NCQ), which allows the SSD to concurrently execute I/O requests. NCQ allows the SSD to schedule multiple I/O requests based on which back end resources are currently idle [29, 50].

The NVM Express (NVMe) protocol [63] was designed to alleviate the bottlenecks of SATA [90], and to enable scalable, high-bandwidth, and low-latency communication over the PCIe bus. When an application issues an I/O request in NVMe, it bypasses the I/O stack in the OS and the block layer queue, and instead directly inserts the request into a *submission queue* (SQ in Figure 1) dedicated to the application. The SSD then selects a request from the SQ, performs the request, and inserts the request's job completion information (e.g., ack, read data) into the request *completion queue* (CQ) for the corresponding application. NVMe has already been widely adopted in modern SSD products [30, 64, 79, 85, 86].

2.3 Flash Translation Layer

The FTL executes on a microprocessor within the SSD, performing I/O requests and flash management procedures [11, 12]. Handling an I/O request in the FTL requires four steps for an SSD using NVMe. First, when the HIL selects a request from the SQ, it inserts the request into a device-level queue. Second, the HIL breaks the request down into multiple *flash transactions*, where each transaction is at the granularity of a single page. Next, the FTL checks if the request is a write. If it is, and the MQ-SSD supports *write caching*, the *write cache management unit* stores the data for each transaction in the write cache space within DRAM, and asks the HIL to prepare a response. Otherwise, the FTL *translates* the logical page address (LPA) of the transaction into a physical page address (PPA), and enqueues the transaction into the corresponding *chip-level queue*. There are separate queues for reads (RDQ) and for writes (WRQ). The *transaction scheduling unit* (TSU) resolves resource contention among the pending transactions in the chip-level queue, and sends transactions that can be performed to its corresponding FCC [20, 78]. Finally, when all transactions for a request finish, the FTL asks the HIL to prepare a response, which is then delivered to the host.

The *address translation module* of the FTL plays a key role in implementing out-of-place updates. When a transaction writes to an LPA, a *page allocation scheme* assigns the LPA to a free PPA. The LPA-to-PPA mapping is recorded in a *mapping table*, which is stored within the non-volatile memory and cached in DRAM (to reduce the latency of mapping lookups) [24]. When a transaction reads from an LPA, the module searches for the LPA's mapping and retrieves the PPA.

The FTL is also responsible for memory wearout management (i.e., *wear-leveling*) and *garbage collection* (GC) [11–13, 23]. GC is triggered when the number of free pages drops below a threshold. The GC procedure reclaims invalidated pages, by selecting a candidate block with a high number of invalid pages, moving any valid pages in the block into a free block, and then erasing the candidate block. Any read and write transactions

generated during GC are inserted into dedicated read (GC-RDQ) and write (GC-WRQ) queues. This allows the transaction scheduling unit to schedule GC-related requests during idle periods.

3 Simulation Challenges for Modern MQ-SSDs

In this section, we compare the capabilities of state-of-the-art SSD simulators to the common features of the modern SSD devices. As shown in Figure 1, we identify three significant features of modern SSDs that are *not* supported by current simulation tools: ❶ multi-queue support, ❷ fast modeling of steady-state behavior, and ❸ proper modeling of the end-to-end request latency. While some of these features are also present in some conventional SSDs, their lack of support in existing simulators is more critical when we evaluate modern and emerging MQ-SSDs, resulting in large deviations between simulation results and measured performance.

3.1 Multi-Queue Support

A fundamental difference of a modern MQ-SSD from a conventional SSD is its use of multiple queues that directly expose the device controller to applications [90]. For conventional SSDs, the OS I/O scheduler coordinates concurrent accesses to the storage devices and ensures fairness for co-running applications [66, 68]. MQ-SSDs eliminate the OS I/O scheduler, and are themselves responsible for fairly servicing I/O requests from concurrently-running applications and guaranteeing high responsiveness. Exposing application-level queues to the storage device enables the use of many optimized management techniques in the MQ-SSD controller, which can provide high performance and a high level of both fairness and responsiveness. This is mainly due to the fact that the device controller can make better scheduling decisions than the OS, as the device controller knows the current status of the SSD's internal resources.

We investigate how the performance of a flow[1] changes when the flow is concurrently executed with other flows on real MQ-SSDs. We conduct a set of experiments where we control the intensity of synthetic workloads that run on four new off-the-shelf MQ-SSDs released between 2016 and 2017 (see Table 4 and Appendix A). In each experiment, there are two flows, `Flow-1` and `Flow-2`, where each flow always keeps its I/O queue full with only sequential read accesses of 4 kB average request size. We control the intensity of a flow by adjusting its I/O queue depth. A deeper I/O queue results in a more intensive flow. We hold the I/O queue depth of `Flow-1` constant in all experiments, setting it to 8 requests. We sweep eight different values for the I/O queue depth of `Flow-2`, ranging from 8 to 1024 requests.

To quantify the I/O service fairness of each device, we measure the average *slowdown* of each executed flow, and then use the slowdown to calculate *fairness* using Equation 1. We define the slowdown of a flow f_i as $S_{f_i} = RT_{f_i}^{shared} / RT_{f_i}^{alone}$, where $RT_{f_i}^{shared}$ is the response time of f_i when it is run concurrently with other flows, and $RT_{f_i}^{alone}$ is the response time of f_i when it runs alone. Fairness (F) is calculated as [22, 56, 58]:

$$F = \frac{\underset{i}{MIN}\{S_{f_i}\}}{\underset{i}{MAX}\{S_{f_i}\}} \quad (1)$$

According to the above definition: $0 < F \leq 1$. Lower F values indicate higher differences between the minimum and maximum slowdowns of all concurrently-running flows, which we say is more unfair to the flow that is slowed down the most. Higher F values are desirable.

Figure 2 depicts the slowdown, normalized throughput (IOPS), and fairness results when we execute `Flow-1` and `Flow-2` concurrently on our four target MQ-SSDs (which we call SSD-A, SSD-B, SSD-C, and SSD-D). The x-axes in all of the plots in Figure 2 represent the queue depth (i.e., the flow intensity) of `Flow-2` in the experiments. For each SSD, we show three plots from left to right: (1) the slowdown and normalized throughput of `Flow-1`, (2) the slowdown and normalized throughput of `Flow-2`, and (3) fairness.

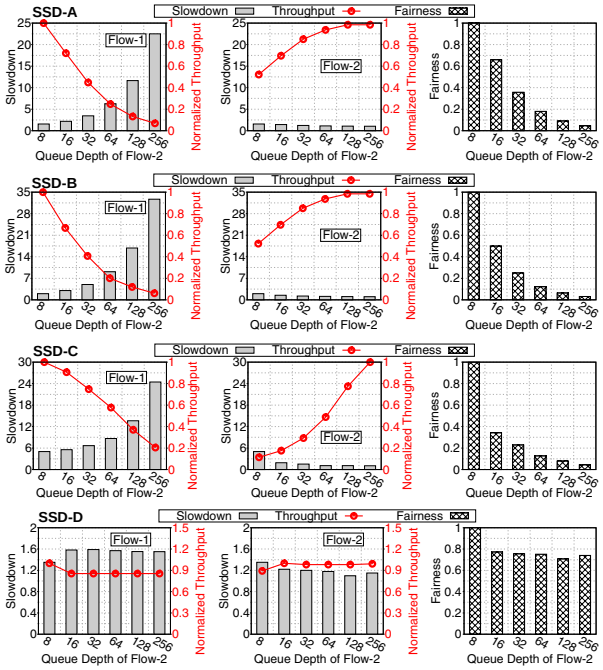


Figure 2: Performance of `Flow-1` (left) and `Flow-2` (center), and fairness (right), when flows are concurrently executed with different intensities on four real MQ-SSDs.

We make four major observations from Figure 2. First, in SSD-A, SSD-B, and SSD-C, the throughput of `Flow-2` substantially increases proportionately with the queue depth. Aside from the maximum bandwidth available from the SSD, there is no limit on the throughput of each I/O flow. Second, `Flow-1` is slowed down significantly due to interference from `Flow-2` when the I/O queue depth of `Flow-2` is much greater than that of `Flow-1`. Third, for SSD-A, SSD-B, and SSD-C, the slowdown of `Flow-2` becomes almost negligible (i.e., its

[1] We assume that each *I/O flow* uses a separate I/O queue.

value approaches 1) as the intensity of `Flow-2` increases. Fourth, SSD-D limits the maximum throughput of each flow, and thus the negative impact of `Flow-2` on the performance of `Flow-1` is well controlled. Further experiments with a higher number of flows reveal that one flow cannot exploit more than a quarter of the full I/O bandwidth of SSD-D, indicating that SSD-D has some level of internal fairness control. In contrast, one flow can unfairly exploit the full I/O capabilities of the other three SSDs.

We conclude that (1) the relative intensity of each flow significantly impacts the throughput delivered to each flow; and (2) MQ-SSDs with fairness controls, such as SSD-D, perform differently from MQ-SSDs without fairness controls when the relative intensities of concurrently-running flows differ. Thus, to accurately model the performance of MQ-SSDs, an SSD simulator needs to model multiple queues and enable multiple concurrently-running flows.

3.2 Steady-State Behavior

SSD performance evaluation standards explicitly clarify that the SSD performance should be reported in the steady state [71].[2] As a consequence, *pre-conditioning* (i.e., quickly reaching steady state) is an essential requirement for SSD device performance evaluation, in order to ensure that the results are collected in the steady state. This policy is important for three reasons. First, the garbage collection (GC) activities are invoked only when the device has performed a certain number of writes, which causes the number of free pages in the SSD to drop below the GC threshold. GC activities interfere with user I/O activity and can significantly affect the sustained device performance. However, a fresh out-of-the-box (FOB) device is unlikely to execute GC. Hence, performance results on an FOB device are unrealistic as they would *not* account for GC [71]. Second, the steady-state benefits of the write cache may be lower than the short-term benefits, particularly for write-heavy workloads. More precisely, in the steady state, the write cache is filled with application data and warmed up, and it is highly likely that no free slot can be allocated to new write requests. This leads to cache evictions and increased flash write traffic in the back end [33]. Third, the physical data placement of currently-running applications is highly dependent on the device usage history and the data placement of previous processes. For example, which physical pages are currently free in the SSD depends on how previous I/O requests wrote to and invalidated physical pages. As a result, channel- and chip-level parallelism in SSDs is limited in the steady state.

Although a number of works do successfully precondition and simulate steady-state behavior, many previous studies have not explored the effect of steady-state behavior on their proposals. Instead, their simulations start with an FOB SSD, and never reach steady state (e.g., when each physical page of the SSD has been written to at least once). Most well-known storage traces are *not* large enough to fill the entire storage space of a modern SSD. Figure 3 shows the total write volume of popular storage workloads [6, 53–55, 61]. We observe that most of the workloads have a total write volume that is much smaller than the storage capacity of most SSDs, with an average write volume of 60 GB. Even for the few workloads that are large enough to fill the SSD, it is time consuming for many existing simulators to simulate each I/O request and reach steady state (see Section 5). Therefore, it is crucial to have a simulator that enables efficient and high-performance steady-state simulation of SSDs.

3.3 Real End-to-End Latency

Request latency is a critical factor of MQ-SSD performance, since it affects how long an application stalls on an I/O request. The end-to-end latency of an I/O request, from the time it is inserted into the host submission queue to the time the response is sent back from the MQ-SSD device to the completion queue, includes seven different parts, as we show in Figure 4. Existing simulation tools model only some parts of the end-to-end latency, which are usually considered to be the dominant parts of the end-to-end latency [3, 26, 27, 35, 38].

Figure 4a illustrates the end-to-end latency diagram for a small 4 kB read request in a typical NAND flash-based MQ-SSD. It includes I/O job enqueuing in the submission queue (SQ) ❶, host-to-device I/O job transfer over the PCIe bus ❷, address translation and transaction scheduling in the FTL ❸, read command and address transfer to the flash chip ❹, flash chip read ❺, read data transfer over the Open NAND Flash Interface (ONFI) [65] bus ❻, and device-to-host read data transfer over the PCIe bus ❼. Steps ❺ and ❻ are *assumed* to be the most time-consuming parts in the end-to-end request processing. Considering typical latency values for an 8 kB page read operation, the I/O job insertion (< 1 µs, as measured on our real SSDs), the FTL request processing on a multicore processor (1 µs) [47] (assuming a mapping table cache hit), and the I/O job and data transfer over the PCIe bus (4 µs) [41, 46] make negligible contributions compared to the flash read (50-110 µs) [49, 51, 52, 69] and the ONFI NV-DDR2 [65] flash transfer (20 µs).

However, the above assumption is unrealistic due to two major reasons. First, for some I/O requests, FTL re-

[2] Based on the SNIA definition [71], a device is in the *steady state* if its performance variation is limited to a deterministic range.

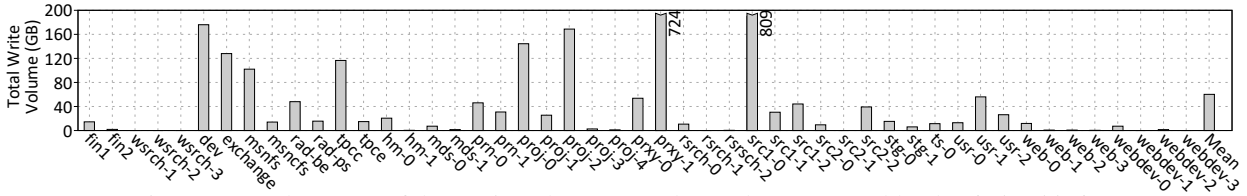
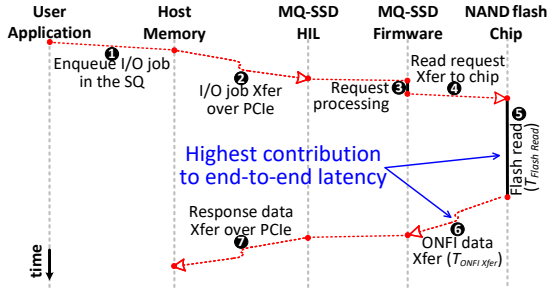
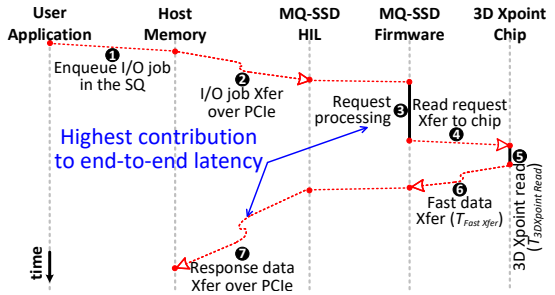


Figure 3: Total amount of data written by commonly-used storage workloads [6, 53–55, 61].



(a) NAND flash memory



(b) 3D XPoint memory

Figure 4: Timing diagram for a 4 kB read request in (a) NAND-flash and (b) 3D XPoint MQ-SSDs.

quest processing may *not* always be negligible, and can even become comparable to the flash read access time. For example, prior work [26] shows that if the FTL uses page-level address mapping, then a workload without locality incurs a large number of misses in the cached mapping table (CMT). In case of a miss in the CMT, the user read operation stalls until the mapping data is read from the SSD back end and transferred to the front end [24]. This can lead to a substantial increase in the latency of Step ❸ in Figure 4a, which can become even longer than the combined latency of Steps ❺ and ❻. In an MQ-SSD, as a greater number of I/O flows execute concurrently, there is more contention for the CMT, leading to a larger number of CMT misses.

Second, as shown in Figure 4b, cutting-edge non-volatile memory technologies, such as 3D XPoint [7, 9, 16, 48], dramatically reduce the access and data transfer times of the MQ-SSD back end, by as much as three orders of magnitude compared to that of NAND flash [25, 40, 42, 43]. The total latency of the 3D XPoint read and transfer ($< 1\,\mu s$) contributes less than 10% to the end-to-end I/O request processing latency ($<10\,\mu s$) [7, 16]. In this case, a conventional simulation tool would be inaccurate, as it does *not* model the major steps contributing to the end-to-end latency.

In summary, a detailed, realistic model of end-to-end latency is key for accurate simulation of modern SSD devices with (1) multiple I/O flows that can potentially lead to a significant increase in CMT (cached mapping table) misses, and (2) very-fast NVM technologies such as 3D XPoint that greatly reduce raw memory read/write latencies. Existing simulation tools do not provide accurate performance results for such devices.

4 Modeling a Modern MQ-SSD with MQSim

To our knowledge, there is no SSD modeling tool that supports multi-queue I/O execution, fast and efficient modeling of the SSD's steady-state behavior, and a full end-to-end request latency estimation. In this work, we present MQSim, a new simulation framework that is developed from scratch to support all of these three important features that are required for accurate performance modeling and design space exploration of modern MQ-SSDs. Although mainly designed for MQ-SSD simulation, MQSim also supports simulation of the conventional SATA-based SSDs that implement native command queuing (NCQ). Our new simulator models all of the components shown in Figure 1, which exist in modern SSDs. Table 1 provides a quick comparison between MQSim and previous SSD simulators.

MQSim is a discrete-event simulator written in C++ and is released under the permissive MIT License [1]. Figure 5 depicts a high-level view of MQSim's main components and their interaction. In this section, we briefly describe these components and explain their novel features with respect to the previous simulators.

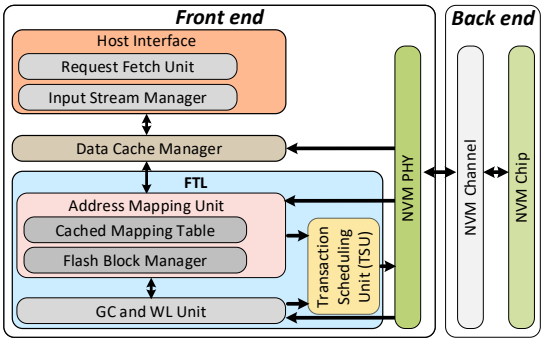


Figure 5: High-level view of MQSim components.

4.1 SSD Back End Model

MQSim provides a simple yet detailed model of the flash memory chips. It considers three major latency components of the SSD back end: (1) address and command transfer to the memory chip; (2) flash memory read/

Table 1: A quick comparison between MQSim and existing SSD modeling tools.

Tool	Multi-Queue Support	Preconditioning	End-to-end Latency	Built-in Implementation of SSD Components
MQSim	Multi-queue scheduling and prioritization	Fast and automatic (enabled by default)	Detailed model of the end-to-end latency	All major components that exist in modern SSDs
Existing Tools	Not supported	Manual, optional, and long execution time	Missing some constant- or variable-latency components	Implementation is missing for some major components

write execution for different technologies that store 1, 2, or 3 bits per cell [32]; and (3) data transfer to/from memory chips. MQSim's flash model considers the constraints of die- and plane-level parallelism, and advanced command execution [65]. One important new feature of MQSim is that it can be configured or easily modified to simulate new NVM chips (e.g., those that do not need erase-before-write). Due to decoupling of the NVM chip communication interface from the chip's internal implementation of the memory operations, one can modify the NVM chip of MQSim without the need to change the implementation of the other MQSim components.

Another new feature of MQSim is that it decouples the sizes of read and write operations. This feature helps to exploit large page sizes of modern flash memory chips in that can enable better write performance, while preventing the negative effects of large page sizes on read performance. For flash chip *writes*, the operation is always page-sized [11, 12]. MQSim's data cache controller can delay writes to eliminate write-back of partially-updated logical pages (where the update size is smaller than the physical page size). When a partially-updated logical page should be written back to the flash storage, the unchanged sub-pages (sectors) of the logical page are first read from the physical page that stores page data. Then, unchanged and updated pieces of the page are merged. In the last step, the entire page data is written to a new free physical page. For flash chip *reads*, the operation could be smaller than the physical page size. When a read operation finishes, only the data pieces that are requested in the I/O request are transferred from flash chips to the SSD controller, avoiding the data transfer overhead of large physical pages.

4.2 SSD Front End Model

The front end model of MQSim includes all of the basic components of a modern SSD controller and provides many new features that do not exist in previous SSD modeling tools.

4.2.1 Host–Interface Model

The host interface component of MQSim provides both NVMe multi-queue (MQ) and SATA native command queue models for a modern SSD. To our knowledge, MQSim is the first modeling tool that supports MQ I/O request processing. There is a request fetch unit within the host interface of MQSim that fetches and schedules application I/O requests from different input queues. The NVMe host interface provides users with a parameter, called `QueueFetchSize`, that can be used to tune the behavior of the request fetch unit, in order to accurately model the behavior of real MQ-SSDs. This parameter defines the maximum number of I/O requests from each SQ that can be concurrently serviced in the MQ-SSD. More precisely, at any given time, the number of I/O requests that are fetched from a host SQ to the device-level queue is always less than or equal to `QueueFetchSize`. This parameter has a large impact on the MQ-SSD multi-flow request processing characteristics discussed in Section 3.1 (i.e., on maximum achievable throughput per I/O flow and probability of inter-flow interference). Appendix A.3 analyzes the effect of this parameter on performance.

MQSim also models different priority classes for host-side request queues, which are part of the NVMe standard specification [63].

4.2.2 Data Cache Manager

MQSim has a data cache manager component that implements a DRAM-based cache with the least-recently-used (LRU) replacement policy. The DRAM cache can be configured to cache (1) recently-written data (default mode), (2) recently-read data, or (3) both recently-written and recently-read data. A new feature of MQSim's cache manager, compared to previous SSD modeling tools, is that it implements a DRAM access model in which the contention among the concurrent accesses to DRAM chips and the latency of DRAM commands are considered. The DRAM cache models in MQSim can be extended to make use of detailed and fast DRAM simulators, such as Ramulator [2, 39], to perform detailed studies of the effect of DRAM cache performance on the overall MQ-SSD performance. We leave this to future work.

4.2.3 FTL Components

MQSim implements all the main FTL components, including (1) the address translation unit, (2) the garbage collection (GC) and wear-leveling (WL) unit, and (3) the transaction scheduling unit. MQSim provides different options for each of these components, including state-of-the-art address translation strategies [24, 78], GC candidate block selection algorithms [10, 18, 23, 45, 81, 91], and transaction scheduling schemes [34, 87]. MQSim also implements several state-of-the-art GC and flash management mechanisms, including preemptible GC I/O scheduling [44], intra-plane data movement from one physical page to another physical page using copyback read and write command pairs [27], and program/erase suspension [87] to reduce the interference of GC operations with application I/O requests. One novel feature of MQSim is that all of its FTL components support multi-flow (i.e., multi-input queue) request processing. For example, the address mapping unit can partition the cached mapping table space among the concurrently running flows. This inherent support of multi-queue-aware request processing facilitates the design space exploration of performance isolation and QoS schemes for MQ-SSDs.

4.3 Modeling End-to-End Latency

In addition to the flash operation and internal data transfer latency (steps ❸, ❹, ❺, and ❻ in Figure 4), there is a mix of variable and constant latencies that MQSim models to determine the end-to-end request latency.

Variable Latencies. These are the variable request processing times in FTL that result from contention in the cached mapping table and the DRAM write cache. Depending on the request type (either read or write) and the request's logical address, the request processing time in FTL includes some of the following items: (1) the time required to read/write from/to the data cache, and (2) the

time to fetch mapping data from flash storage in case of a miss in the cached address mapping table.

Constant Latencies. These include the times required to transmit the I/O job information, the entire user data, and the I/O completion information over the PCIe bus, and the firmware (FTL) execution time on the controller's microprocessor. The PCIe transmission latencies are calculated based on a simple packet latency model provided by Xilinx [41] that considers: (1) the PCIe communication bandwidth, (2) the payload and header sizes of the PCIe Transaction Layer Packets (TLP), (3) the size of the NVMe management data structures, and d) the size of the application data. The firmware execution time is estimated using both a CPU and cache latency model [1].

4.4 Modeling Steady-State Behavior

The basic assumption of MQSim is that *all* simulations should be executed when the modeled device is in steady state. To model the steady-state behavior, MQSim, *by default*, *automatically* executes a preconditioning function before starting the actual simulation process. This function performs preconditioning in a short time (e.g., less than 8 min when running `tpcc` [53] on an 800 GB MQ-SSD) without the need to execute additional I/O requests. During preconditioning, all available physical pages of the modeled SSD are transitioned to either a valid or invalid state, based on the steady-state valid/invalid page distribution model provided in [82] (only very few flash blocks are assumed to remain free and are added to the free block pool). MQSim pre-processes the input trace to extract the LPA (logical page address) access characteristics of the application I/O requests in the trace, and then uses the extracted information as inputs to the valid/invalid page distribution model. In addition, input trace characteristics, such as the average write arrival rate and the distribution of write addresses, are used to warm up the write cache.

4.5 Execution Modes

MQSim provides two modes of operation: (i) *standalone* mode, where it is fed a real disk trace or a synthetic workload, and (ii) *integrated* mode, where it is fed disk requests from an execution-driven engine (e.g., gem5 [8]).

5 Comparison with Previous Simulators

The increasing usage of SSDs in modern computing systems has boosted interest in SSD design space exploration. To this end, several simulators have been developed in recent years. Table 2 summarizes the features of MQSim and popular existing SSD modeling tools. The table also shows the *average error rates* for the performance of real storage workloads reported by each simulator, compared to the performance measured on four real MQ-SSDs (see Appendix A.1 for our methodology).

Existing tools either do not model some major components of modern SSDs or provide very simplistic component models that lead to unrealistic I/O request latency estimation. In contrast, MQSim provides detailed implementations for all of the major components of modern SSDs. MQSim is written in C++ and has 13K lines of code (LOC). Next, we discuss the main advantages of MQSim compared to the previous tools.

Host–Interface Logic. As Table 2 shows, most of the existing simulators assume a very simplistic HIL model with no explicit management mechanism for the I/O request queue. This leads to an unrealistic SSD model regarding the requirements of both NVMe and SATA protocols. As we mention in Section 3, the concurrent execution of I/O flows presents many challenges for performance predictability and fairness in MQ-SSDs. No ex-

Table 2: Comparison of MQSim with previous SSD modeling tools.

Simulator	HIL Protocol		Execution Mode				End-to-End Latency					Front-End Components							LOC[14]	Simulation Error (%)			
	NVMe MQ	SATA NCQ	Alone[1]	Full[2]	Emul[3]	Prec[4]	NVM R/W[5]	NVM Xfer	FTL Proc[6]	Cache Acc.[7]	Host Xfer[8]	Map P[9]	Map H[10]	GC	Write Cache	TSU[11]	WRL[12]	MQ FTL[13]		SSD-A	SSD-B	SSD-C	SSD-D
MQSim	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	13K	8	6	18	14
SSDModel [3]			✓				✓	✓			✓	✓		✓			✓		1K	91	155	196	136
FlashSim [38]			✓				✓	✓			✓	✓	✓	✓			✓		8K	99	259	310	138
SSDSim [27]			✓				✓	✓				✓		✓	✓				5K	70	68	74	85
NANDFlashSim [32]							✓	✓											7K	–	–	–	–
VSSIM [92]					✓		✓	✓				✓	✓	✓	✓		✓		6K	–	–	–	–
WiscSim [26]		✓	✓				✓	✓	✓			✓	✓	✓			✓		7K	95	277	324	135
SimpleSSD [35]			✓	✓			✓	✓					✓	✓		✓	✓		7K	–	–	–	–

[1] Standalone execution [2] Integrated execution with full-system simulator [3] SSD emulation for real system
[4] Fast and accurate preconditioning of the modeled SSD to enable accurate steady-state results
[5] Flash (NVM) read/write timing [6] FTL request processing overhead [7] Accurate modeling of write cache access latency
[8] Host-to-device and device-to-host data transfer delay [9] Page-level address mapping [10] Hybrid address mapping
[11] FTL transaction scheduling unit [12] FTL wear-leveling unit [13] Built-in support for multi-queue-aware request processing in FTL
[14] Lines of source code

isting simulator implements NVMe and multi-queue I/O request management, and, hence, accurately models the behavior of MQ-SSDs. Also, except for WiscSim, we find that no existing simulator implements an accurate model of the SATA protocol and NCQ request processing. This leads to unrealistic SATA device simulation, as NCQ-based I/O scheduling plays a key role in the performance of real SSD devices [15, 26].

Steady-State Simulation. To our knowledge, accurate and fast steady-state behavior modeling is not provided by many existing SSD modeling tools. Among the tools listed in Table 2, only SSDSim provides a function, called `make_aged`, to change the status of a set of physical pages to valid before starting the actual execution of an input trace. This simple method *cannot* accurately replicate the steady-state behavior of an SSD for two reasons. First, after the execution of `make_aged`, the physical blocks would include only valid pages or only free pages. This is far from the steady-state status of blocks in real devices, where each non-free block has a mix of valid and invalid pages [28, 81, 82]. Second, the steady-state status of the data cache is *not* modeled, i.e., the simulation starts with a completely empty write cache.

In general, it *is* possible to bring these simulators to steady state. However, there is no *fast* pre-conditioning support for them, and pre-conditioning *must* be performed by executing traces. Preconditioning an existing simulator requires users to generate traces with a large enough number of I/O requests, and can significantly slow down the simulator, especially when a high-capacity SSD is modeled. For example, our studies with SSDSim show that pre-conditioning may increase the simulation time up to 80x if an 800 GB SSD is modeled.[3]

Detailed End-to-End Latency Model. As described in Section 3.3, the end-to-end latency of an application I/O request includes different components. Table 2 shows that latency modeling in existing simulators is mainly focused on the latency of the flash chip operation and the SSD internal data transfer. As we explain in Section 3.3, this is an unrealistic model of the end-to-end I/O request processing latency, even for a conventional SSD.

To study the accuracy of the existing tools in modeling real devices, we create four models for the four real SSDs shown in Table 4 in each simulator, and execute three real traces, i.e., `tpcc`, `tpce`, and `exchange`. We exclude the simulators that do *not* support trace-based execution. The four rightmost columns of Table 2 show the average error rate of each simulator in modeling the performance (i.e., read and write latency) of these four real devices. The error rates of the four evaluated simulators are almost one order of magnitude higher than that of MQSim. We believe that these high error rates are due to four major reasons: (1) the lack of write cache or inaccurate modeling of the write cache access latency, (2) the lack of built-in support for steady-state modeling, (3) incomplete modeling of the request processing latency in FTL, and (4) the lack of modeling of the host-to-device communication latency.

[3]The increase in simulation time depends on the access pattern, intensity, and mix of I/O requests (read vs. write) of the workload.

6 Research Directions Enabled by MQSim

MQSim is a flexible simulation tool that enables different studies on both modern and conventional SSD devices. In this section, we discuss two new research directions enabled by MQSim, which could not be explored easily using existing simulation tools. First, we use MQSim to perform a detailed analysis of inter-flow interference in a modern MQ-SSD (Section 6.1). We explain how sharing different internal resources in an MQ-SSD, such as the write cache, cached mapping table, and back end resources, can introduce fairness issues. Second, we explain how the full-system simulation mode of MQSim can enable detailed application-level studies (Section 6.2).

6.1 Design Space Exploration of Fairness and QoS Techniques for MQ-SSDs

As we describe in Section 1, fairness and QoS should be considered as first-class design criteria for modern datacenter SSDs. MQSim provides an accurate framework to study inter-flow interference, thus enables the ability to devise interference-aware MQ-SSD management algorithms for sharing of the internal MQ-SSD resources. As we show in Section 3.1, concurrently running two I/O flows might lead to disproportionate slowdowns for each flow, greatly degrading fairness and proportional progress. This is particularly important in high-end SSD devices, which provide higher throughput per I/O flow, as we show in Appendix A.3.

We find that this inter-flow interference is mainly the result of contention that takes place at three locations in an MQ-SSD: 1) the write cache in the front end, 2) the cached mapping table (CMT) in the front end, and 3) the storage resources in the back end. In this section, we use MQSim to explore the impact of these three points of contention on performance and fairness, which cannot be explored accurately using existing simulators.

6.1.1 Methodology

MQ-SSD Configuration. Table 3 lists the specification of the MQ-SSD that we model in MQSim for our contention studies.

Metrics. To measure performance, we use *weighted speedup* (WS) [70] of the average response time (RT), which represents the overall efficiency and system-level

Table 3: Configuration of the simulated SSD.

SSD Organization	Host interface: PCIe 3.0 (NVMe 1.2) User capacity: 480 GB Write cache: 256 MB, CMT: 4 MB 8 channels, 4 chips per channel `QueueFetchSize` = 512
Flash Communication Interface	ONFI 3.1 (NV-DDR2) Width: 8 bit, Rate: 333 MT/s
Flash Microarchitecture	8 KiB page, 448 B metadata per page, 256 pages per block, 2048 blocks per plane, 2 planes per die
Flash Access Parameters	Read latency: 75 µs, Program latency: 750 µs, Erase latency: 3.8 ms

throughput [21] provided by an MQ-SSD during the concurrent execution of multiple flows:

$$\text{WS} = \sum_i \frac{RT_i^{alone}}{RT_i^{shared}} \quad (2)$$

where RT_i^{alone} and RT_i^{shared} are defined in Section 3.1.

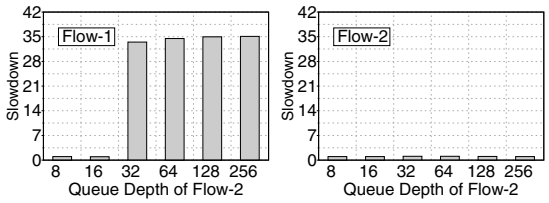
To demonstrate the effect of inter-flow interference on fairness, we report *slowdown* and *fairness (F)* metrics, as defined in Section 3.1.

6.1.2 Contention at the Write Cache

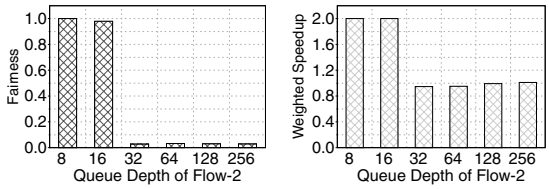
One point of contention among concurrently-running flows in an MQ-SSD is the write cache. For flows with low to moderate write intensity (where the average depth of the I/O queue less than 16), or with high spatial locality, the write cache decreases the response time of write requests, by avoiding the need for the requests to wait for the write to complete to the underlying memory. For flows with high write intensity or with highly-random accesses, the write requests fill up the limited capacity of the write cache quickly, causing significant cache thrashing and limiting the decrease in write request response time. Such flows not only do *not* benefit from the write cache themselves, but also prevent other lower-write-intensity flows from benefiting from the write cache, leading to a large performance loss for the lower-write-intensity flows.

To understand how the contention at the write cache affects system performance and fairness, we perform a set of experiments where we run two flows, `Flow-1` and `Flow-2`, both of which perform only random-access write requests. In both flows, the average request size is set to 8 kB. We set `Flow-1` to have a moderate write intensity, by limiting the queue depth to 8 requests. We vary the queue depth of `Flow-2` from 8 requests to 256 requests, to control the write intensity of the flow. In order to isolate the effect of write cache interference in our experiments, we (1) assign each flow to a dedicated subset of back end resources (i.e., `Flow-1` uses Channels 1–4, and `Flow-2` uses Channels 5–8), to avoid introducing any interference in the back end; and (2) use a perfect CMT, where all address translation requests are hits, to avoid interference due to limited CMT capacity.

Figure 6a shows the slowdown of each flow when the two flows run concurrently, compared to when each flow runs alone. Figure 6b shows the fairness and performance of the system when the two flows run concurrently. We make four key observations from the figures. First, `Flow-1` is slowed down significantly when `Flow-2` has a high write intensity (i.e., its queue depth is greater than 16), indicating that at high write intensities, `Flow-2` induces write cache trashing. Second, the slowdown of `Flow-2` is negligible, because of the low write intensity of `Flow-1`. Third, fairness degrades greatly, as a result of the write cache contention, when `Flow-2` has a high write intensity. Fourth, write cache contention causes an MQ-SSD to be inefficient at concurrently running multiple I/O flows, as the weighted speedup is reduced by over 50% when `Flow-2` has a high write intensity compared to when it has a low write intensity.



(a) Slowdown of `Flow-1` (left) and `Flow-2` (right)



(b) Fairness (left) and system performance (right)

Figure 6: Impact of write cache contention.

We conclude that write cache contention leads to unfairness and overall performance degradation for concurrently-running flows when one flow has a high write intensity. In these cases, the high-write-intensity flow (1) does *not* benefit from the write cache; and (2) *prevents* other, lower-write-intensity flows from taking advantage of the write cache, even though the other flows would otherwise benefit from the cache. This motivates the need for fair write cache management algorithms for MQ-SSDs that take inter-flow interference and flow write intensity into account.

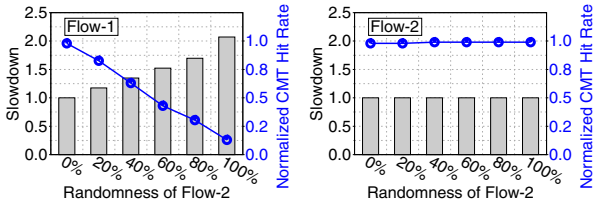
6.1.3 Contention at the Cached Mapping Table

As we discuss in Section 3.3, address translation can noticeably increase the end-to-end latency of an I/O request, especially for read requests. We find that for I/O flows with random access patterns, the cached mapping table (CMT) miss rate is high due to poor reuse of address translation mappings, which causes the I/O requests generated by the flow to stall for long periods of time during address translation. This is not true for I/O flows with sequential accesses, for which the CMT miss rate remains low due to spatial locality. However, when two I/O flows run concurrently, where one flow has a random access pattern and another flow has a sequential access pattern, the poor locality of the flow with the random access pattern may cause *both flows* to have high CMT miss rates.

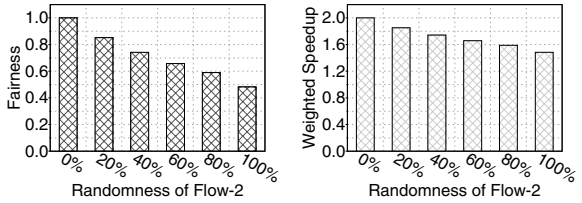
To understand how contention at the CMT affects system performance and fairness, we perform a set of experiments where we concurrently run two flows that issue read requests with an average request size of 8 kB. In these experiments, `Flow-1` has a fully-sequential access pattern, and `Flow-2` has a random access pattern for a fraction of the total execution time, and has a sequential access pattern for the remaining time. We vary the *randomness* (i.e., the fraction of the execution time with a random access pattern) of `Flow-2`. To isolate the effects of CMT contention, we assign `Flow-1` to Channels 1–4 in the back end, and assign `Flow-2` to Channels 5–8.

Figure 7a shows the slowdown and change in CMT hit rate of each flow when `Flow-1` and `Flow-2` run concur-

rently, compared to when each flow runs alone. Figure 7b shows the fairness and overall performance of the system when the two flows run concurrently. We make two observations from the figures. First, as the randomness of `Flow-2` increases, the CMT hit rate of `Flow-1` decreases, while the CMT hit rate of `Flow-2` remains constant. This indicates that the randomness of `Flow-2` introduces contention at the CMT, which hurts the CMT hit ratio of `Flow-1`. Second, as the CMT hit rate of `Flow-1` decreases, the flow experiences a greater slowdown, with a 2.1x slowdown when `Flow-2`'s access pattern is completely random. Third, as the randomness of `Flow-2` increases, both fairness and overall system performance decrease, as the interference introduced by `Flow-2` hurts the performance of `Flow-1` without providing any noticeable benefit to `Flow-2`.



(a) Slowdown and CMT hit rate (normalized to the hit rate when `Flow-2` randomness is 0%) for `Flow-1` (left) and `Flow-2` (right)



(b) Fairness (left) and system performance (right)

Figure 7: Impact of CMT contention.

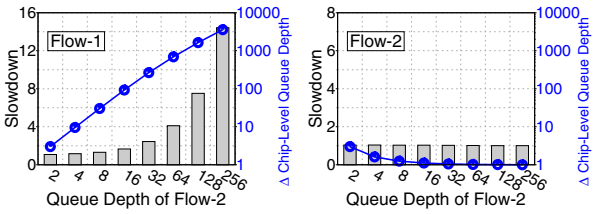
We conclude that the CMT contention induced by an I/O flow with a random access pattern disproportionately slows down concurrently-running flows with sequential access patterns, which would otherwise benefit from the CMT, leading to high unfairness and system performance degradation. To avoid such unfairness and performance loss, an MQ-SSD should use CMT management algorithms that are aware of inter-flow interference.

6.1.4 Contention at the Back End Resources

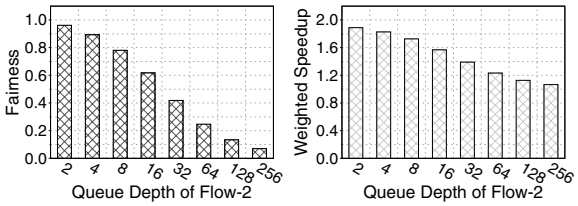
A third point of contention is at the back end resources within an MQ-SSD (see Section 2.1). A high-intensity flow can use up most of the back end resources if the flow issues a large number of requests in a short period of time. This stalls the requests issued by a low-intensity concurrently-running flow, as the requests cannot be serviced before the back end resources finish servicing requests from the high-intensity flow.

To understand how contention at the back end resources affects system performance and fairness, we perform a set of experiments where we concurrently run two I/O flows that issue random reads with a request size of 8 kB. `Flow-1` is a low-intensity I/O flow, as we limit its submission queue size (see Section 2.2) to 2 requests. We vary the submission queue size of `Flow-2` from 2 requests to 256 requests, to control the flow intensity. In order to isolate the effect of back end resource contention, we disable the write cache, and simulate a CMT where address translation requests always hit.

Figure 8a shows the slowdown when `Flow-1` and `Flow-2` run concurrently, and the change in the average chip-level queue depth (i.e., the number of requests waiting to be serviced by the back end; see Section 2.3) for each flow during concurrent execution, compared to the depth when each flow runs alone. Figure 8b shows the fairness and overall performance of the system when the two flows run concurrently. We make four observations from the figures. First, the average chip-level queue depth of `Flow-1` increases significantly when the intensity of `Flow-2` increases. Second, `Flow-1` is slows down significantly when we increase the host-side queue depth of `Flow-2` beyond 16. For example, when `Flow-2` is at the highest intensity that we test (with a host-side queue depth of 256 requests), `Flow-1` slows down by 14.4x. Third, the effect of inter-flow interference on `Flow-2` is negligible, as its slowdown is almost equal to 1 for host-side queue depths larger than 4. Fourth, the asymmetric slowdowns (i.e., the large slowdown for `Flow-1` and the lack of slowdown for `Flow-2`) cause both fairness and the overall system performance to decrease.



(a) Slowdown and average chip-level queue depth of `Flow-1` (left) and `Flow-2` (right)



(b) Fairness (left) and system performance (right)

Figure 8: Impact of back end resource contention.

We conclude that a high-intensity flow can significantly increase the depth of the chip-level queues and thus lead to a large slow-down for concurrently-running low-intensity flows. The FTL transaction scheduling unit must be aware of the inter-flow interference at the MQ-SSD back end to make the per-flow performance more fair and thus keep the overall performance high.

6.2 Application-Level Studies

To study the effect of SSD device-level design choices on application-level performance metrics, such as instructions per cycle (IPC), an SSD simulator must be integrated and run together with a full-system simulator. We integrate MQSim with gem5 [8] to provide a complete

model of multi-queue I/O execution and a complete computer system. As Table 2 shows, among existing SSD simulators, only SimpleSSD [35] is integrated with a full-system simulator, and SimpleSSD does *not* simulate multi-queue I/O execution. In this section, we show the effectiveness of our integrated simulator, by studying how changes to `QueueFetchSize` (see Section 4.2.1) affect the IPC of concurrently-executing applications due to storage-level interference.

We conduct a set of experiments, running instances of file server (`fs`) [77], mail server (`ms`) [77], web server (`ws`) [77], and IOzone large file access (`io`) [62] applications using the integrated execution mode of MQSim. We first execute each application alone (i.e., without interference from other applications), and then concurrently execute the application with a second application to study the effect of inter-application interference. To isolate the effect of inter-flow interference, where each flow belongs to one application, we assign each application to a single processor core and a single memory channel. We test two different values of `QueueFetchSize` (16 entries and 1024 entries) to examine how `QueueFetchSize` affects inter-application interference. For these experiments, we measure *application slowdown* (S_{app}), which is calculated as $S_{app_i} = IPC^{alone}_{app_i} / IPC^{shared}_{app_i}$, and use application slowdown to determine fairness using Equation 1.

Figure 9 shows the slowdown of each application and the system fairness for six pairs of concurrently-executing applications. On the x-axis, we list the applications used in each pair, along with the value of `QueueFetchSize` that we use. We make two observations from the figure. First, for application pairs where one of the applications is `ms` or `ws`, the impact of `QueueFetchSize` on fairness is negligible. Both `ms` and `ws` benefit mainly from caching a large part of their data set in main memory, and hence issue very few requests to the SSD. This keeps storage-level interference low, as `ms` and `ws` do not contend often for access to the SSD with the other applications that they are paired with. Second, `fs` and `io` have high storage access intensities, and hence interfere significantly when they are paired together. In this case, we observe that a large `QueueFetchSize` value leads to 60% fairness reduction.

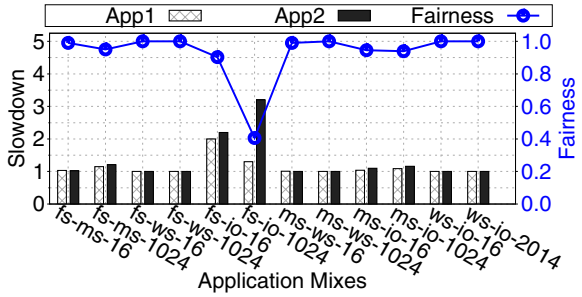


Figure 9: Application-level impact of `QueueFetchSize`.

We conclude that full-system behavior can greatly impact the fairness and performance of I/O flows on an MQ-SSD, as it affects the storage-level intensity of each flow.

7 Related Work

To our knowledge, MQSim is the first simulator that (1) accurately simulates both modern and conventional SSDs, (2) faithfully models modern host–interface protocols such as NVMe, and (3) supports the accurate simulation of SSDs that use emerging ultra-fast memory technologies. We compare MQSim to existing state-of-the-art SSD simulation tools in Section 5, and show that MQSim provides greater capabilities and accurate results. In this section, we provide a brief summary of other related works.

A number of prior works consider the performance and implementation challenges of MQ-SSDs [5, 31, 89, 90]. Xu et al. [89] analyze the effect of MQ-SSDs on the performance of modern hyper-scale and database applications. Awad et al. [5] evaluate the impact of different NVMe host-interface implementations on the system performance. Vučinić et al. [83] show that the current NVMe protocol will be a performance bottleneck in future PCM-based storage devices. The authors modify the NVMe standard in order to improve its performance for future PCM-based SSDs.

Other works [31, 72] focus on managing multiple flows in modern SSDs. Song and Yang [72] partition the SSD back end resources among concurrently-running I/O flows to provide performance isolation and alleviate inter-flow interference. Jun and Shin [31] propose a device-level scheduling technique for MQ-SSDs with built-in virtualization support.

None of these previous studies provide a simulation framework for MQ-SSDs or study the sources of inter-flow interference inside MQ-SSDs.

8 Conclusion

We introduce MQSim, a new simulator that accurately captures the behavior of both modern multi-queue SSDs and conventional SATA-based SSDs. MQSim faithfully models a number of critical features absent in existing state-of-the-art simulators, including (1) modern multi-queue-based host–interface protocols (e.g., NVMe), (2) the steady-state behavior of SSDs, and (3) the end-to-end latency of I/O requests. MQSim can be run as a standalone tool, or integrated with a full-system simulator. We validate MQSim against real off-the-shelf SSDs, and demonstrate that it provides highly-accurate results. By accurately modeling modern SSDs, MQSim can uncover important issues that cannot be modeled accurately using existing simulators, such as the impact of inter-flow interference. We have released MQSim as an open-source tool [1], and we hope that MQSim enables researchers to explore new ideas and directions.

Acknowledgments

We thank our shepherd Haryadi Gunawi and the anonymous referees for their feedback on this work. We thank our industrial partners, especially Google, Huawei, Intel, and VMware, for their generous support.

References

[1] MQSim GitHub Repository. https://github.com/CMU-SAFARI/MQSim.

[2] Ramulator GitHub Repository. https://github.com/CMU-SAFARI/ramulator.

[3] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design Tradeoffs for SSD Performance. In *USENIX ATC* (2008).

[4] AUSAVARUNGNIRUN, R., CHANG, K. K.-W., SUBRAMANIAN, L., LOH, G. H., AND MUTLU, O. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *ISCA* (2012).

[5] AWAD, A., KETTERING, B., AND SOLIHIN, Y. Non-Volatile Memory Host Controller Interface Performance Analysis in High-Performance I/O Systems. In *ISPASS* (2015).

[6] BATES, K., AND MCNUTT, B. UMass Rrace Repository. http://traces.cs.umass.edu/.

[7] BILLI, E. How NVMe and 3D XPoint Will Create a New Datacenter Architecture. In *FMS* (2016).

[8] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., SEN, R., SEWELL, K., SHOAIB, M., VAISH, N., HILL, M. D., AND WOOD, D. A. The gem5 Simulator. *SIGARCH Comput. Archit. News* (2011).

[9] BOURZAC, K. Has Intel Created a Universal Memory Technology? *IEEE Spectrum* (2017).

[10] BUX, W., AND ILIADIS, I. Performance of Greedy Garbage Collection in Flash-Based Solid-State Drives. *Perform. Eval.* (2010).

[11] CAI, Y., GHOSE, S., HARATSCH, E. F., LUO, Y., AND MUTLU, O. Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives. *Proc. IEEE* (2017).

[12] CAI, Y., GHOSE, S., HARATSCH, E. F., LUO, Y., AND MUTLU, O. Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery. arXiv:1711.11427 [cs:AR], 2017.

[13] CHANG, L.-P., KUO, T.-W., AND LO, S.-W. Real-Time Garbage Collection for Flash-Memory Storage Systems of Real-Time Embedded Systems. *TECS* (2004).

[14] CHEN, F., HOU, B., AND LEE, R. Internal Parallelism of Flash Memory-Based Solid-State Drives. *TOS* (2016).

[15] CHEN, F., LEE, R., AND ZHANG, X. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-Speed Data Processing. In *HPCA* (2011).

[16] COULSON, R. 3D XPoint Technology Drives System Architecture. In *SNIA Storage Industry Summit* (2016).

[17] DAS, R., MUTLU, O., MOSCIBRODA, T., AND DAS, C. R. Application-Aware Prioritization Mechanisms for On-Chip Networks. In *MICRO* (2009).

[18] DESNOYERS, P. Analytic Modeling of SSD Write Performance. In *SYSTOR* (2012).

[19] EBRAHIMI, E., LEE, C. J., MUTLU, O., AND PATT, Y. N. Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *ASPLOS* (2010).

[20] ELYASI, N., ARJOMAND, M., SIVASUBRAMANIAM, A., KANDEMIR, M. T., DAS, C. R., AND JUNG, M. Exploiting Intra-Request Slack to Improve SSD Performance. In *ASPLOS* (2017).

[21] EYERMAN, S., AND EECKHOUT, L. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* (2008).

[22] GABOR, R., WEISS, S., AND MENDELSON, A. Fairness and Throughput in Switch on Event Multithreading. In *MICRO* (2006).

[23] GAL, E., AND TOLEDO, S. Algorithms and Data Structures for Flash Memories. *CSUR* (2005).

[24] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *ASPLOS* (2009).

[25] HANDY, J. 3D XPoint: Speed at What Cost? In *FMS* (2017).

[26] HE, J., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The Unwritten Contract of Solid State Drives. In *EuroSys* (2017).

[27] HU, Y., JIANG, H., FENG, D., TIAN, L., LUO, H., AND ZHANG, S. Performance Impact and Interplay of SSD Parallelism Through Advanced Commands, Allocation Strategy and Data Granularity. In *ICS* (2011).

[28] ILIADIS, I. Rectifying Pitfalls in the Performance Evaluation of Flash Solid-State Drives. *Perform. Eval.* (2014).

[29] INTEL CORPORATION. Intel SSD DC S3500 Series Datasheet, 2015.

[30] INTEL CORPORATION. Intel 3D NAND SSD DC P4500 Series Datasheet, 2017.

[31] JUN, B., AND SHIN, D. Workload-Aware Budget Compensation Scheduling for NVMe Solid State Drives. In *NVMSA* (2015).

[32] JUNG, M., CHOI, W., GAO, S., WILSON III, E. H., DONOFRIO, D., SHALF, J., AND KANDEMIR, M. T. NANDFlashSim: High-Fidelity, Microarchitecture-Aware NAND Flash Memory Simulation. *TOS* (2016).

[33] JUNG, M., AND KANDEMIR, M. Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications. In *SIGMETRICS* (2013).

[34] JUNG, M., AND KANDEMIR, M. T. Sprinkler: Maximizing Resource Utilization in Many-Chip Solid State Disks. In *HPCA* (2014).

[35] JUNG, M., ZHANG, J., ABULILA, A., KWON, M., SHAHIDI, N., SHALF, J., KIM, N. S., AND KANDEMIR, M. SimpleSSD: Modeling Solid State Drives for Holistic System Simulation. *CAL* (2017).

[36] KIM, J., KIM, J., PARK, P., KIM, J., AND KIM, J. SSD Performance Modeling Using Bottleneck Analysis. *CAL* (2017).

[37] KIM, Y., PAPAMICHAEL, M., MUTLU, O., AND HARCHOL-BALTER, M. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO* (2010).

[38] KIM, Y., TAURAS, B., GUPTA, A., AND URGAONKAR, B. FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. In *SIMUL* (2009).

[39] KIM, Y., YANG, W., AND MUTLU, O. Ramulator: A Fast and Extensible DRAM Simulator. *CAL* (2016).

[40] KÜLTÜRSAY, E., KANDEMIR, M., SIVASUBRAMANIAM, A., AND MUTLU, O. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative. In *ISPASS* (2013).

[41] LAWLEY, J. Understanding Performance of PCI Express Systems. XILINX White Paper, 2014.

[42] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA* (2009).

[43] LEE, B. C., ZHOU, P., YANG, J., ZHANG, Y., ZHAO, B., IPEK, E., MUTLU, O., AND BURGER, D. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* (2010).

[44] LEE, J., KIM, Y., SHIPMAN, G. M., ORAL, S., AND KIM, J. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. *TC* (2013).

[45] LI, Y., LEE, P. P., AND LUI, J. Stochastic Modeling of Large-Scale Solid-State Storage Systems: Analysis, Design Tradeoffs and Optimization. In *SIGMETRICS* (2013).

[46] LIU, J., MAMIDALA, A., VISHNU, A., AND PANDA, D. K. Performance Evaluation of InfiniBand with PCI Express. In *CONNECT* (2004).

[47] MARVELL. Marvell 88SS1093 Flash Memory Controller, 2017.

[48] MICRON TECHNOLOGY, INC. Breakthrough Nonvolatile Memory Technology. `https://www.micron.com/products/advanced-solutions/3d-xpoint-technology`.

[49] MICRON TECHNOLOGY, INC. NAND Flash Memory - MT29E64G08CECBB Datasheet, 2009.

[50] MICRON TECHNOLOGY, INC. M500 2.5-Inch SATA NAND Flash SSD Series Datasheet, 2013.

[51] MICRON TECHNOLOGY, INC. NAND Flash Memory - MLC+ MT29F256G08CKCAB Datasheet, 2014.

[52] MICRON TECHNOLOGY, INC. NAND Flash Memory - MT29E128G08CBCCB Datasheet, 2016.

[53] MICROSOFT CORPORATION. Microsoft Enterprise Traces. `http://iotta.snia.org/traces/130`.

[54] MICROSOFT CORPORATION. Microsoft Production Server Traces. `http://iotta.snia.org/traces/158`.

[55] MICROSOFT CORPORATION. Microsoft Research Cambridge Traces. `http://iotta.snia.org/traces/388`.

[56] MOSCIBRODA, T., AND MUTLU, O. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX Security* (2007).

[57] MUTLU, O. Memory Scaling: A Systems Architecture Perspective. In *IMW* (2013).

[58] MUTLU, O., AND MOSCIBRODA, T. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO* (2007).

[59] MUTLU, O., AND MOSCIBRODA, T. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA* (2008).

[60] MUTLU, O., AND SUBRAMANIAN, L. Research Problems and Opportunities in Memory Systems. *SUPERFRI* (2015).

[61] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *EuroSys* (2009).

[62] NORCOTT, W. D., AND CAPPS, D. IOzone Filesystem Benchmark, 2003.

[63] NVM EXPRESS WORKGROUP. NVM Express Specification, Revision 1.2, 2014.

[64] OCZ. RD400/400A Series Datasheet, 2016.

[65] ONFI WORKGROUP. Open NAND Flash Interface Specification, Revision 4.0, 2014.

[66] PARK, S., AND SHEN, K. FIOS: A Fair, Efficient Flash I/O Scheduler. In *FAST* (2012).

[67] SATA-IO. Serial ATA Revision 3.3. `http://www.sata-io.org`, 2016.

[68] SHEN, K., AND PARK, S. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *USENIX ATC* (2013).

[69] SK HYNIX INC. F26 32Gb MLC NAND Flash Memory TSOP Legacy, 2011.

[70] SNAVELY, A., AND TULLSEN, D. M. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *ASPLOS* (2000).

[71] SNIA TECHNICAL POSITION. Solid State Storage (SSS) Performance Test Specification (PTS) Enterprise, version 1.1, 2013.

[72] SONG, X., YANG, J., AND CHEN, H. Architecting Flash-Based Solid-State Drive for High-Performance I/O Virtualization. *CAL* (2014).

[73] SUBRAMANIAN, L., LEE, D., SESHADRI, V., RASTOGI, H., AND MUTLU, O. The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost. In *ICCD* (2014).

[74] SUBRAMANIAN, L., LEE, D., SESHADRI, V., RASTOGI, H., AND MUTLU, O. BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling. *TPDS* (2016).

[75] SUBRAMANIAN, L., SESHADRI, V., GHOSH, A., KHAN, S., AND MUTLU, O. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory. In *MICRO* (2015).

[76] SUBRAMANIAN, L., SESHADRI, V., KIM, Y., JAIYEN, B., AND MUTLU, O. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *HPCA* (2013).

[77] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A Flexible Framework for File System Benchmarking. *USENIX; login* (2016).

[78] TAVAKKOL, A., MEHRVARZY, P., ARJOMAND, M., AND SARBAZI-AZAD, H. Performance Evaluation of Dynamic Page Allocation Strategies in SSDs. *TOMPECS* (2016).

[79] TOSHIBA CORPORATION. PX04PMB Series Datasheet, 2016.

[80] USUI, H., SUBRAMANIAN, L., CHANG, K. K.-W., AND MUTLU, O. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. *TACO* (2016).

[81] VAN HOUDT, B. A Mean Field Model for a Class of Garbage Collection Algorithms in Flash-based Solid State Drives. In *SIGMETRICS* (2013).

[82] VAN HOUDT, B. On the Necessity of Hot and Cold Data Identification to Reduce the Write Amplification in Flash-Based SSDs. *Perform. Eval.* (2014).

[83] VUČINIĆ, D., WANG, Q., GUYOT, C., MATEESCU, R., BLAGOJEVIĆ, F., FRANCA-NETO, L., LE MOAL, D., BUNKER, T., XU, J., SWANSON, S., AND BANDIĆ, Z. DC Express: Shortest Latency Protocol for Reading Phase Change Memory Over PCI Express. In *FAST* (2014).

[84] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *OSDI* (1994).

[85] WESTERN DIGITAL CORPORATION. HGST Ultrastar SN200 Series Datasheet, 2017.

[86] WESTERN DIGITAL CORPORATION. SanDisk Skyhawk & Skyhawk Ultra NVMe PCIe SSD Datasheet, 2017.

[87] WU, G., AND HE, X. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *FAST* (2012).

[88] XIANG, X., GHOSE, S., MUTLU, O., AND TZENG, N.-F. A Model for Application Slowdown Estimation in On-Chip Networks and Its Use for Improving System Fairness and Performance. In *ICCD* (2016).

[89] XU, Q., SIYAMWALA, H., GHOSH, M., AWASTHI, M., SURI, T., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance Characterization of Hyper-Scale Applications on NVMe SSDs. In *SIGMETRICS* (2015).

[90] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *SYSTOR* (2015).

[91] YANG, M.-C., CHANG, Y.-M., TSAO, C.-W., HUANG, P.-C., CHANG, Y.-H., AND KUO, T.-W. Garbage Collection and Wear Leveling for Flash Memory: Past and Future. In *SMARTCOMP* (2014).

[92] YOO, J., WON, Y., HWANG, J., KANG, S., CHOIL, J., YOON, S., AND CHA, J. VSSIM: Virtual Machine Based SSD Simulator. In *MSST* (2013).

A MQSim Validation

A.1 Evaluation Methodology

To validate the accuracy of MQSim, we compare its performance results to four state-of-the-art MQ-SSDs (SSD-A, SSD-B, SSD-C, and SSD-D) manufactured between 2016 and 2017. Table 4 lists key properties of the four MQ-SSDs. We precondition each device with full-load write traffic to write to 70% of the available logical space [71]. The device preconditioning process includes two 4-hour phases. In the first phase, we perform sequential writes, while in the second phase, we perform random writes. We perform real-system experiments on a server that contains an Intel Xeon E3-1240 v6 3.70GHz processor and 32 GB of DDR4 main memory. The system uses Ubuntu 16.04.2 with version 2.6.27 of the Linux kernel, and the OS is stored in a 500 GB Western Digital HDD. We run the `fio` benchmark tool for performance evaluations, and all storage devices are connected to the PCIe bus as add-in cards.

Table 4: Key characteristics of real MQ-SSDs.

Code	Production Year	Capacity	Flash Technology
SSD-A	2016	800 GB	MLC
SSD-B	2016	256 GB	MLC
SSD-C	2017	1 TB	TLC
SSD-D	2016	512 GB	TLC

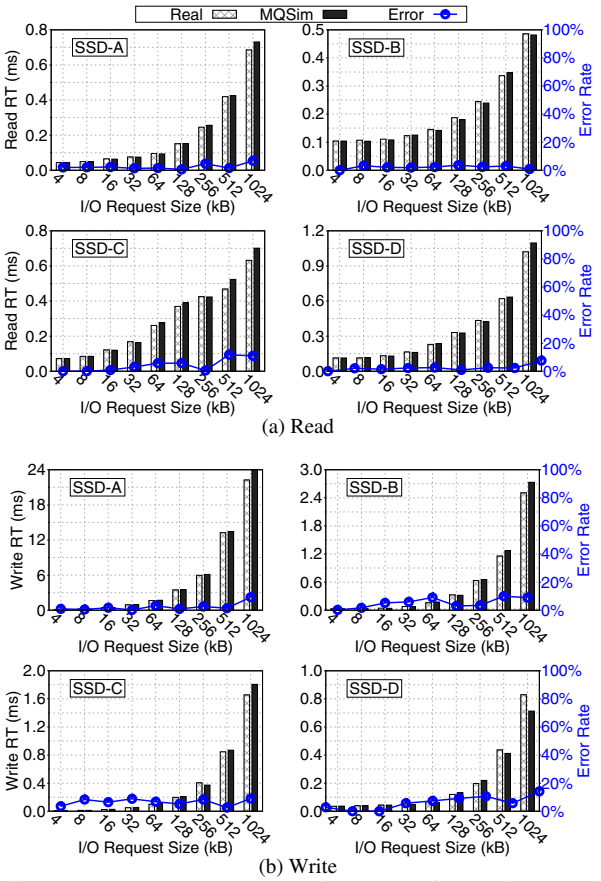
We validate our simulator with four different configurations that correspond to our four real MQ-SSDs. To this end, we extract the main structural parameters of each real SSD using a microbenchmarking program. This program analyzes and estimates the SSD's internal configuration (e.g., NAND flash page size, NAND flash read/write latency, number of channels in the SSD, address mapping strategy, write cache size) based on the methods described in prior SSD modeling studies [14, 15, 36]. We have open-sourced our microbenchmark [1]. For garbage collection (GC) management, we enable all of the advanced GC mechanisms described in Section 4.2.3, except write suspension, in MQSim. According to the specifications of the flash chips used in two of the SSD devices, write suspension is not supported.

A.2 Performance Validation

We validate MQSim against real devices using both synthetic and real workloads. Our synthetic workloads issue random accesses, and consist of only read requests or only write requests, where we set the queue depth to 1 request.

Figure 10 compares the read and write request response time[4] measured on our four real MQ-SSDs with the latencies reported by MQSim for our synthetic workloads. The plots in Figure 10a and 10b show the read and the write latencies, respectively. The x-axes reflect different I/O request sizes, ranging from 4 kB to 1 MB. The blue curves show the error percentage of the simulation model. We observe that across all request sizes, the response times reported by MQSim match very closely

[4]Response time is defined as the time from when a host request is enqueued in the submission queue to when the SSD response is enqueued in the completion queue.



(a) Read

(b) Write

Figure 10: Average response time (RT) for read (a) and write (b) requests, reported by MQSim, compared to RT measured on four real MQ-SSD devices, for synthetic workloads. The blue curves show the error rates of MQSim's reported latency.

with the measured response times of the real devices, especially for SSD-B and SSD-D. Averaged across all four MQ-SSDs and all I/O request sizes, the error rates for read and write requests are 2.9% and 4.9%, respectively.

Figure 11 shows the accuracy of the request response time reported by MQSim as a cumulative distribution function (CDF), for three real workloads [53]: `tpcc`, `tpce`, and `exchange`. We observe that MQSim's reported response times are very accurate when compared to the response times measured on the real MQ-SSDs. The average error rates for SSD-A, SSD-B, SSD-C, and SSD-D are 8%, 6%, 18%, and 14%, respectively.

We conclude that MQSim accurately models the performance of real MQ-SSDs.

A.3 Multi-Queue Simulation

To validate the accuracy of the multi-queue I/O execution model in MQSim, we conduct a set of simulation experiments using two I/O flows, `Flow-1` and `Flow-2`, where each flow generates only sequential read requests. We maintain a constant request intensity for `Flow-1`, by setting its I/O queue depth to 8 requests. We vary the intensity of `Flow-2` across our experiments, by varying the I/O queue depth between 8 entries and 256 entries.

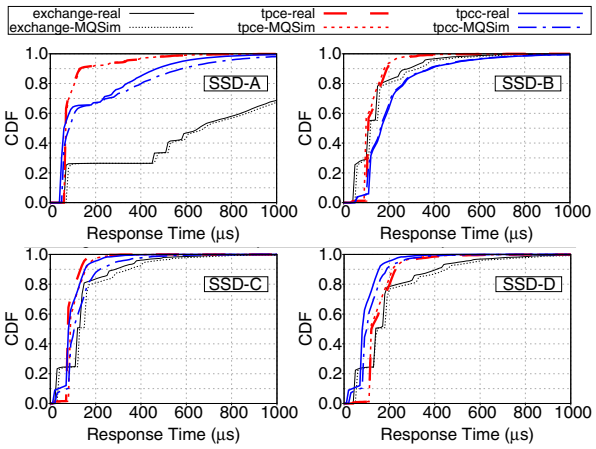
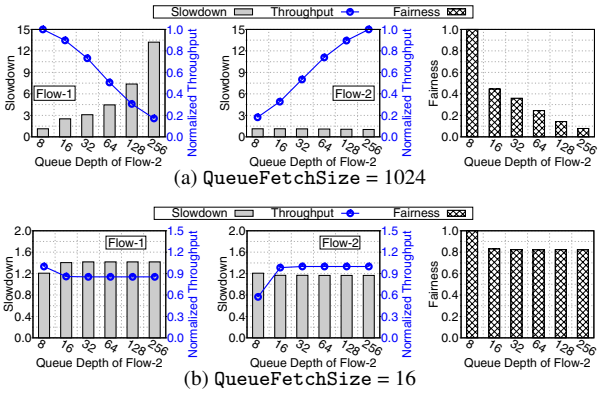


Figure 11: Comparison of response time CDF when running real workloads on MQSim and on real MQ-SSDs.

For each `Flow-2` I/O queue depth, we test two different values (16 and 1024) of `QueueFetchSize` (see Section 4.2.1).

Figure 12 shows the slowdown and normalized throughput (IOPS) of `Flow-1` (left) and `Flow-2` (center), and the fairness (see Section 3.1) of the system (right). We make two key observations from the figure.



(a) `QueueFetchSize` = 1024

(b) `QueueFetchSize` = 16

Figure 12: MQSim successfully models the multi-queue I/O processing model in (a) SSD-A, SSD-B, and SSD-C, and (b) SSD-D (compare with Figure 2).

First, we find that MQSim successfully models the behavior of real MQ-SSDs that are optimized for higher per-flow throughput (e.g., SSD-A, SSD-B, SSD-C) when the value of `QueueFetchSize` is equal to 1024. Figure 12a shows similar trends for slowdown, throughput, and fairness to the measurements we perform on real MQ-SSDs, which we show in Figure 2. When `QueueFetchSize` is set to 1024, a higher number of I/O requests from each flow are fetched into the device-level queue of the MQ-SSD. In both our MQSim results and the measured results on real MQ-SSDs, we observe that as the intensity of `Flow-2` increases, its throughput increases significantly with little slowdown, while the throughput of `Flow-1` decreases significantly, causing `Flow-1` to slow down greatly. This occurs because when `Flow-2` has a high intensity, it unfairly uses most of the back end resources in the MQ-SSD, causing requests from `Flow-1` to wait for longer latencies before they can be serviced.

Second, MQSim accurately models the behavior of real MQ-SSD products that implement mechanisms to control inter-flow interference, such as SSD-D, when `QueueFetchSize` is set to 16. We see that the trends in Figure 12b are similar to those observed in our measured results from SSD-D in Figure 2. When `QueueFetchSize` is set to 16, only a limited number of I/O requests for each concurrently-running flow are serviced by the back end, preventing any one flow from unfairly using most of the resources within the MQ-SSD. As a result, even when `Flow-2` has a high intensity, `Flow-1` does not experience significant slowdown.

We conclude that by adjusting `QueueFetchSize`, MQSim successfully models different multi-queue I/O processing mechanisms in modern MQ-SSD devices.

A.4 Steady-State Behavior Modeling

As we discuss in Section 4.4, MQSim pre-conditions the flash storage space and warms up the SSD data cache based on the characteristics of the co-running workloads. To validate the steady-state model in MQSim, we conduct a set of experiments using MQSim under high write intensity, and compare the results to those from real MQ-SSD devices. Figure 13 plots the read and write response times for (1) actual I/O execution on SSD-B (which is representative of the general behavior of the state-of-the-art SSDs we examine); (2) *MQSim-NoPrec*, where MQSim is run *without* pre-conditioning, and (3) *MQSim-Prec*, where MQSim is run *with* pre-conditioning.

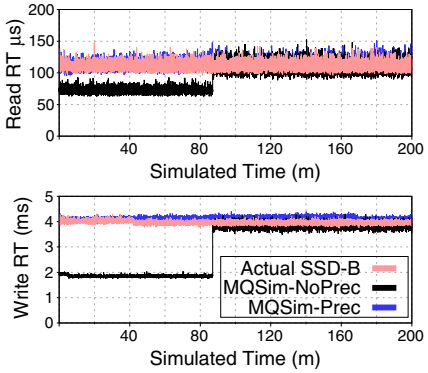


Figure 13: MQSim accurately models the steady-state read and write response times (RT) of SSD-B, using fast preconditioning.

We make two observations from the figure. First, MQSim with pre-conditioning successfully follows the response time results extracted from SSD-B. Second, MQSim without pre-conditioning reports lower response time results at the beginning of the experiment, since the simulated SSD is not yet in steady state. Once the whole storage space is written, the response time results become similar to the real device, as garbage collection and write cache evictions now take place in simulation at a rate similar to the rate measured on SSD-B. We conclude that MQSim's pre-conditioning quickly and accurately models the steady-state behavior of real MQ-SSDs.

PEN: Design and Evaluation of Partial-Erase for 3D NAND-Based High Density SSDs

Chun-Yi Liu, Jagadish B. Kotra, *Myoungsoo Jung, Mahmut T. Kandemir
{cql5513, jbk5155, kandemir}@cse.psu.edu, *mj@camelab.org
The Pennsylvania State University, *Yonsei University

Abstract

3D NAND flash memories promise unprecedented flash storage capacities, which can be extremely important in certain application domains where both storage capacity and performance are first-class target metrics. However a block of 3D NAND flash contains many more pages than its 2D counterpart. This increased number of pages-per-block has numerous ramifications such as the longer erase latency, higher garbage collection costs, and increased write amplification factors, which can collectively prevent the 3D NAND flash products from becoming the mainstream in high-performance storage domain. In this paper, we introduce PEN, an architecture-level mechanism that enables partial-erase of flash blocks. Using our proposed partial-erase support, we also discuss how one can build a custom garbage collector for two types of flash translation layers (FTLs), namely, block-level FTL and hybrid FTL. Our experimental evaluations of PEN with a set of diverse real storage workloads indicate that the proposed approach can shorten the write latency by 44.3% and 47.9% for block-level FTL and hybrid FTL, respectively.

1 Introduction

NAND flash based solid state disks (SSDs) have become one of the dominant storage components in different computing domains, ranging from embedded systems to general purpose workstations to high-performance datacenters [6, 13, 22, 57]. High-performance computing employs SSDs in various ways such as an SSD cache [41, 55] or a bursty buffer [39], to mitigate the performance bottlenecks imposed by the conventional hard disk drives (HDDs).

While SSDs can significantly improve the overall system performance, there is also an emerging trend that pushes SSDs toward an entirely different direction [20, 31, 51]. Specifically, major flash vendors amplify storage capacity by transitioning from 2D NAND flash to 3D NAND flash. The 3D NAND flash technology layers flash cells vertically, which can increase the size of the individual flash dies. For example, Samsung stacks 100 layers of *charge trap flash* (CTF) cells, and as a result can achieve 1 Terabit density flash dies without any modification to the existing flash interface [10].

Layering multiple CTF cells increases the number of pages in a physical block, rather than the number of blocks within a die, which makes the internal micro-architecture of 3D NAND different compared to the 2D planar flash. Consider the vertical architecture of a particular 3D NAND flash [20], *VNAND*. VNAND increases the die density by stacking more layers, but in this architecture, CTF cells across the different layers share a same set of pillars (channel), thereby increasing the number of pages per block. Furthermore, as all the block-related control circuits of VNAND reside on the block decoder of the top layer due to staircase-like control gate [20], this decoder area controls the signals to all CTF cells of the underlying layers. Consequently, as one stacks more layers, the amount of such block-related control circuits for each block increases. However, the area from where one can control all layers is limited, which in turn reduces the number of blocks but increases the number of pages per block. Owing to this, a block of VNAND contains at least 3 times more pages per block compared to the 2D flash.

Unfortunately, the new structure of VNAND can exacerbate the overheads incurred due to garbage collection (GC), which is one of the well-known performance bottlenecks in modern SSDs. Specifically, the peripheral circuits and the micro-architecture of VNAND's [19, 26] large-granularity erase makes the latency characteristics of 3D NAND worse compared to 2D flash. In addition, the large number of pages per block can potentially accommodate more valid data that a flash firmware needs to migrate for each GC. The longer erase time and relatively more valid pages to migrate (i.e., a series of reads and writes) can have a significant performance impact on GC operations and may in turn render 3D flash difficult to directly replace 2D flash in many designs of high-performance SSD.

In this work, we propose *PEN*, a novel strategy to enable **P**artial **E**rase for 3D **N**AND flash technology. PEN

alleviates the GC overheads by introducing a finer erase unit in 3D NAND, which can reduce number of valid pages copied during a GC, thereby reducing the GC latency. To the best of our knowledge, this is the first work that investigates *partial-erase* for 3D NAND, starting from the circuit level and evaluating its architectural ramifications from both the performance and reliability angles. Our contributions can be summarized as follows:

- We present a comprehensive architectural support, "partial-erase operation" that addresses the potential performance degradation imposed by 3D NAND flash. The proposed low-level operation can selectively reset multiple pages instead of erasing a bulk block by modifying the 3D NAND peripheral circuits, page decoders, and command logics, with minimum area overhead.
- While our partial-erase operation can be leveraged to alleviate performance degradation, the number of pages per reset should be determined at the design time. However, it is a non-trivial task to statically decide the operation granularity of such partial-erase operation. This inflexibility in turn can lead to a large mapping table size or introduce more valid page migrations. We propose a novel GC algorithm, called "M-Merge" that keeps the original mapping table size when the partial-erase operation is employed. In addition, M-Merge can adaptively decide the optimal partial-erase granularity by considering program-disturbance issues at a run-time.
- We demonstrate that the performance degradation in 3D NAND flash stems from the increased number of valid pages copied during a GC operation. The evaluation studies using a set of 12 real storage workloads reveal that our PEN mechanism (putting partial-erase operation and M-Merge together) can significantly reduce the valid page copies during a GC operation, thereby improving the overall system performance.

2 Background

2.1 NAND Flash Organization

NAND flash memory consists of several blocks constituting a plane, as shown in Figure 1. Each block is made up of a number of pages. Page is a unit of read and write, and its size varies from 2KB to 32KB [1]. A traditional 2D NAND flash typically consists of 128 to 192 such pages per block. The number of pages per block increases [10] in a 3D NAND flash, and it can be as high as 576 [29].

Figure 1 shows a vertical-channel 3D NAND flash implemented by Samsung [20]. NAND flash cells connected in series form a pillar (channel) with top and bottom select transistors represented by UpperST and LowerST, respectively. Cells in the same horizontal axis form a page, represented by P-0, P-1, etc. The upper select

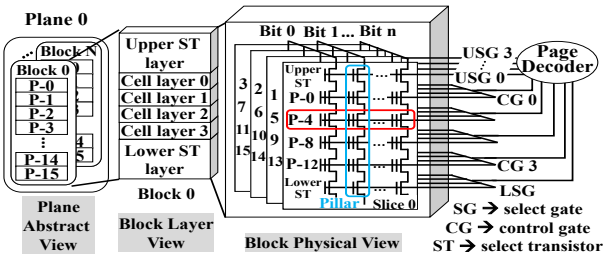


Figure 1: Vertical-channel NAND flash circuit.

gates (USG) and the control gates (CG) are used in accessing a page corresponding to an I/O request. USG is used to select the corresponding slice and the CG is used to select the corresponding page in a slice.

The basic operations in a NAND flash chip are read, write, and erase [46]. The erase operation resets the data to value "1" in a page. It is performed at a *block-granularity*; the data in all the pages in a block are reset per erase operation. In a NAND flash device, the number of erase cycles is limited. Typically, an erase operation is implemented in two phases: (1) data-erase and (2) erase-verify. During the data-erase phase, the data in all the pages in a victim block are reset, and the erase-verify phase checks if all the pages have been successfully reset or not. The entire erase operation is iteratively repeated until the data in all the pages are successfully reset.

The data-erase circuit implementation of an erase operation is vendor-specific. There are two popular implementations: (a) bulk data-erase (implemented in Samsung SSDs) [20] and (b) gate-induced drain leak (GIDL) (implemented in Toshiba and Macronix SSDs) [27, 31, 51] data-erase. Bulk data-erase operation imposes a high voltage (typically 20V) to the shared substrate (containing multiple blocks), while the CGs for the block being erased are set to 0V. Hence, all the pages in a block are erased per erase operation unlike the GIDL implementation. In GIDL, the data-erase operation is implemented at a pillar granularity, and all the pillars in the same block are erased simultaneously. More specifically, it is implemented by imposing high voltage to USGs, LSG and bitlines, while the voltage of CGs is set to different values based on the strength of GIDL. The erase-verify implementation is achieved by imposing CGs with 0V, while SGs are imposed with bypass voltage. An unsuccessfully erased page is identified by measuring the current passing through the channels when bitline voltage changes from 0V to floating.

2.2 Flash Translation Layer (FTL)

The NAND flash vendors implement a Flash Translation Layer (FTL) [3, 15, 18, 21, 38, 40] to keep track of the physical location of a page in flash chips. FTL imple-

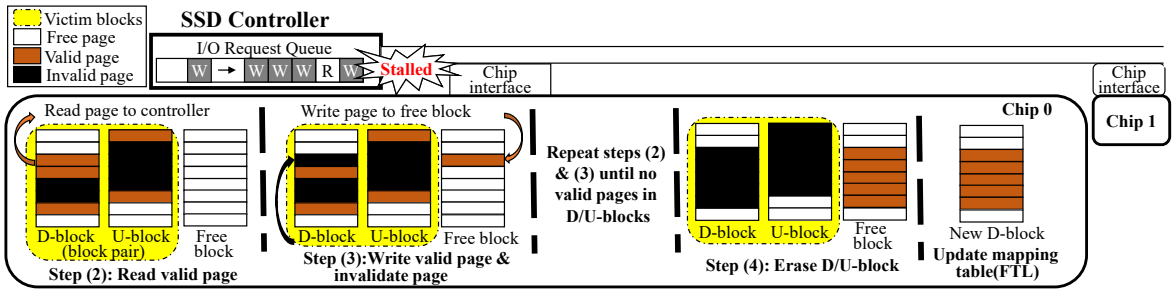


Figure 2: NFTL GC (Merge) overview.

ments two major functionalities, namely, address mapping and garbage collection.

2.2.1 Address Mapping

FTL maintains a data structure called mapping table. It maps a given logical page/block address to the physical location. Each read/write I/O request has to be translated by FTL to route the request to a corresponding physical page. When a logical page is updated, the old physical page is marked as "invalid," since SSDs do not support in-place update. The number of invalid pages in chip increases as the write requests are processed. Address mapping can be broadly classified into three categories based on the granularity at which the mappings are managed viz., (a) page-level, (b) block-level, and (c) hybrid mapping.

Page-level Mapping: This address mapping implementation needs a huge mapping table to manage translations at a page granularity. Note that a 1TB SSD requires at least 1GB mapping table. While SSD capacity doubles as the number of stacked NAND layers increases, such a huge mapping table becomes a major issue for SSD design (in terms of both price and power consumption).

Block-level Mapping: Block-level mappings only store the mapping information per block, and therefore, the size of mapping table is smaller than other mappings.

A well-known block-level implementation is NFTL [3]. In NFTL, the mapping information of a block consists of a Data block (D-block) and Update block (U-block). A D- and U-block together are referred to as a block-pair. D-block represents the actual data block where a page is originally mapped to, while U-block represents the block to which the updated pages from D-block are written to, leaving the corresponding paired D-block page invalid. Typically, the number of U-blocks is much smaller than the number of D-blocks; so, multiple D-blocks compete for a U-block.

A new write of NFTL is performed on the mapped page in the D-block, while an updated write to the same address is logged in the paired U-block. As a result, a read to an address may have to search (read) pages from the U-block sequentially to retrieve the latest copy. Hence, the read and write performance can be slower in NFTL compared to that of the page-level mapping.

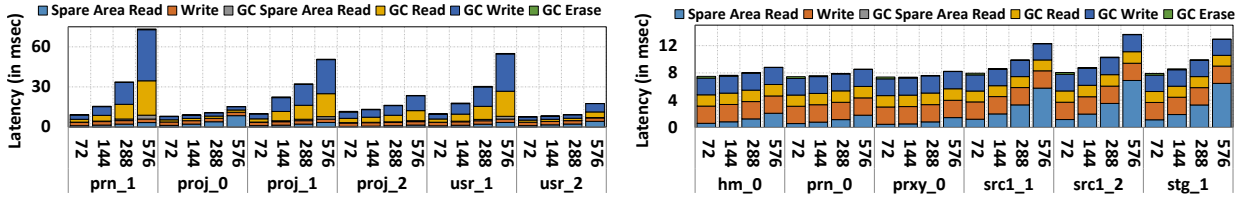
Hybrid Mapping: The hybrid mapping combines the best of the two previous mappings by (a) adopting the block-level mapping to reduce the mapping table size, and (b) utilizing partial (or small) amount of the page-level mapping table to accelerate the performance. Various such proposals include Superblock [21], FAST [40], DFTL [15], and LAST [38]. In this work, we only focus on the Superblock FTL implementation.[1]

2.2.2 Garbage Collection (GC)

GC is triggered by the write requests or the controller firmware to clear the invalid pages left in the flash chips, so that SSDs have enough free pages for the future writes. GC typically contains four steps: (1) selecting victim blocks, (2) reading valid pages from the victim blocks, (3) writing the valid pages into the reserved free blocks, and (4) erasing the victim blocks. Typically, steps (1) and (4) are executed only once, while steps (2) and (3) are executed repeatedly until no valid page is left in the victim blocks. Since GC changes the FTL address mapping, different FTLs implement their GC algorithms.

Figure 2 illustrates the GC in NFTL, which is mainly achieved by **Merge** operation. Merge *copies* all valid pages contained in victim block pair to a reserved free block, after which the victim block pair is erased. There are two scenarios when a GC will be triggered: (a) fully-utilized U-block and (b) unpaired D-block. In scenario (a), the paired U-block has no free page, and therefore, GC needs to be triggered to clear the invalid pages in this block pair. In scenario (b), the D-block corresponding to a write request does not have a U-block paired with it; as a result, GC is triggered on another block pair to reclaim a free U-block. We assume that the victim block pair for step (1) has already been selected. The second step involves reading the valid page from the victim block pair. In the third step, the page read in the second step is writ-

[1] Our proposal works equally well for other hybrid FTL implementations as well.



(a) High write-intensive workloads

(b) Low write-intensive workloads

Figure 3: Write latency breakdown for different block sizes in the case of NFTL.

ten into the free block and the read page is invalidated in the victim block. These steps, (2) and (3), are repeated for all the valid pages in the victim blocks, as depicted in the figure. Once all the valid pages are copied, in step (4), all the pages in the block are erased using an erase operation, also shown in the figure. Finally, the FTL address mapping is updated to reflect the new D-block.

As depicted in Figure 2, the on-demand read/write I/O requests *cannot* be served by the SSD chips during the GC and are stalled in the per-channel DRAM queue [4, 32–36, 43, 48, 52, 56] in the SSD controller. As a result, the GC can negatively affect the application performance [23–25]

2.3 Effect of Block Size

2.3.1 Effect of Block Size on Performance

With the increase in density for the 3D NAND flash, the number of valid pages per block increases. As a result, the number of pages to be copied from the victim block to the free block during a GC also increases. Consequently, the GC duration increases, in turn increasing the access latency for the read and write I/O requests, thereby degrading the overall performance significantly. This issue is widely referred to as the "Big Block" problem [54].

We performed experiments to quantitatively demonstrate the relationship between GC and the number of pages per block in the case of NFTL (block-level FTL)[2]. All the configurations tested have the same SSD capacity to prevent the capacity from affecting the GC triggered frequency. To keep the same capacity, we ensure that a plane has the same number of pages, but the number of pages per block are varied across different configurations. Here are the four evaluated configurations (blocks per plane, **pages per block**): (a) (15104, **72**), (b) (7552, **144**), (c) (3776, **288**), and (d) (1888, **576**). The rest of parameters can be found in Table 1. All four configurations are evaluated on the SSDSim [17] simulator using 12 write-dominant workloads shown in Table 2.

[2]We observed that Superblock FTL has a similar trend.

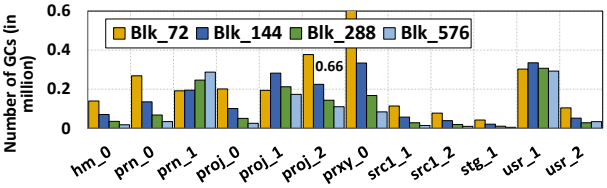


Figure 4: Effect of block sizes on the number of GC invocations in the case of NFTL.

Figures 3a and 3b show the breakdown of average write access latencies (in milliseconds) for the different block sizes. As can be observed, as the number of pages per block increases, the latency increase and becomes maximum for a block with 576 pages. The access latency incurred by a write request includes (1) the time spent in performing the actual write to the pages, and (2) the time spent in waiting in the I/O request queue if this write triggers a GC. The GC time which effects the wait time of a write I/O request can further be broken down into (a) GC spare-area read, (b) GC Read, (c) GC Write, and (d) GC Erase, as shown in Figures 3a and 3b. The GC spare-area read time accounts for the time spent in reading the page status to identify the valid pages. The GC read/write time accounts for copying the identified valid pages from the victim blocks to a free block, while the GC erase time accounts for the time spent in performing the erase of the victim blocks.

Figures 3a and 3b plot the write access latency breakdown for the high and low write-intensive workloads, respectively. In Figure 3a, as the number of pages per block increases, the time spent in copying the valid pages (which includes GC read/write) increases from 58% for a block with 72 pages to 79% for a block with 576 pages. This result indicates that reducing the number of valid pages to be copied is crucial under the high-intensive workloads. In figure 3b, the time spent in copying the valid pages remains the same, but the time spent in reading the spare-area increases due to the inherent design of NFTL.

Figure 4 shows the number of GC invocations for different block sizes for all workloads. In general, the number of GC invocations are halved as the number of pages

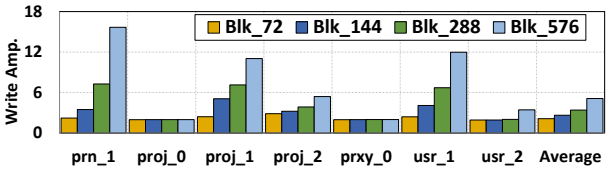


Figure 5: Effect of block sizes on writes amplification in the case of NFTL.

per block doubles; so, the time spent in erasing the blocks decreases. However, in the high write-intensive workloads, such as prn_1 and proj_1, the number of GCs increases as the number of pages per block doubles. This is because the configuration with a fewer number of blocks (larger block) has fewer competing blocks, thereby resulting in higher number of GCs.

2.3.2 Impact of Block Size on Lifetime

Write amplification factor is the ratio of the amount of data which the host writes and the amount of writes that actually occurs on the flash media (including the GC writes) [16].

Figure 5 shows the write amplification varying block sizes for high write-intensive workloads. Write amplification for the high write-intensive workloads increases, on average, from 2.1 for a block with 72 pages to 5.1 for a block with 576 pages. This is because, as the number of pages per block increases, the corresponding number of valid pages per block also increases. Since the erase operation during a GC is at a block-level, all the valid pages from the victim block need to be copied to a free block, thereby increasing the write-amplification. Such increased write-amplification may shorten the lifetime of SSDs. We also observed that Superblock FTL [21] has a similar trend. However, page-level FTLs do not suffer from the increased write-amplification because of smart page allocation and victim block selection algorithms [2, 12, 16, 42, 44, 45].

3 Overview of Partial-Erase Operation

Due to the large number of pages in a block, 3D NAND-based flash storage can be subjected to excessive valid page copy overhead. To reduce the number of valid page copies in 3D NAND, we propose a *partial-erase operation for 3D NAND flash* (PEN). Unlike the block-level erase operation, our proposed partial-erase operation performs erase at a "partial block" (PB) granularity. Since our proposal enables partial-erase, the amount of valid pages that need to be copied during a GC is reduced significantly, eventually improving the write latency and the overall I/O throughput. Also, since the number of pages copied during a GC reduces, the overall number of writes induced by GCs is also reduced, and this in turn results in lower write-amplification and increased lifetime.

Our proposal consists of both hardware and software changes. On the hardware side, our hardware modifications for the partial-erase operation support both the partial data-erase phase and the partial erase-verify phase. The implementation of the partial data-erase phase includes inhibiting the erase of pages other than the pages in a PB of a block. Similarly, the implementation of partial erase-verify phase includes only verifying the pages in PB to decide if the partial-erase operation needs to continue erasing the non-erased pages in PB or not.

The partial-erase operation may incur additional program disturbances to the neighboring pages, causing the neighboring cells' data to be modified. To solve the disturbance, we provide a software-based modification, since the hardware-based solutions [11, 14] typically reduce the 3D NAND array density, which may trigger further GCs. Another reason is that the hardware-based solutions can only mitigate the disturbance, so the data in the boundary pages may still be corrupted with more partial-erase operations. Therefore, a software solution is more preferable.

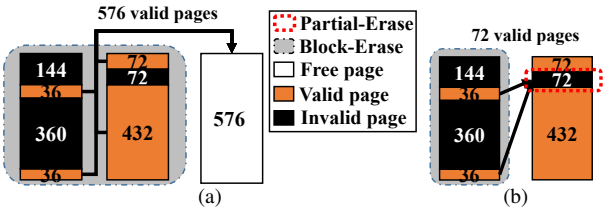
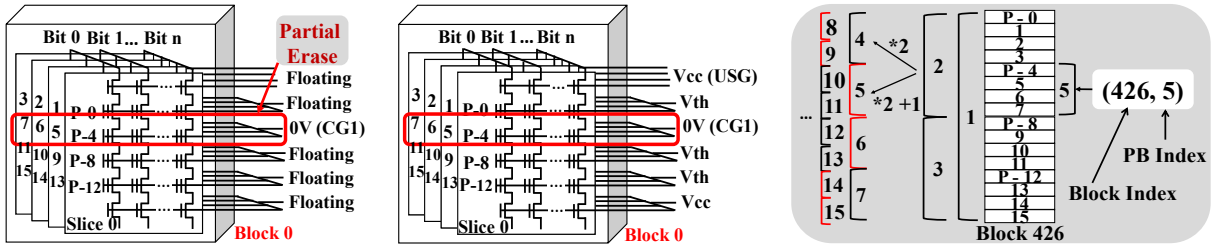


Figure 6: (a) depicts the baseline without partial-erase support necessitating 576 valid pages to be copied from the victim blocks to the new block. (b) depicts our partial-erase support which only necessitates 72 valid pages to be copied.

On the software side, we propose a modified merge (M-Merge) algorithm in GC that utilizes our partial-erase operation to reduce the number of valid pages copied from the victim blocks. One major contribution of our M-Merge algorithm is that, the valid pages in victim blocks are "consolidated" into fewer blocks, unlike in the baseline GC algorithm where all valid pages are copied to free blocks. Therefore, in our proposed algorithm, we copy very few pages during GC. Another contribution is that the proposed M-merge algorithm is aware of the possible disturbance by the partial-erase, so the data in non-erased pages will not be corrupted by disturbances. Figure 6 shows the difference between the baseline GC and M-Merge based GC. In this example, our M-Merge only copies 72 valid pages during a GC; in comparison, the baseline GC copies a total of 576 valid pages.



(a) Partial data-erase phase. (b) Partial erase-verify phase. (c) Partial block index.

Figure 7: Partial-Erase operation design.

4 Controller Hardware for Partial-Erase

4.1 Peripheral Circuit Modifications

The peripheral circuit modifications for our proposed partial data-erase and partial erase-verify phases are covered in this subsection. In the baseline erase implementations of VNAND (bulk-erase) circuits, the erase-inhibition technique is used to restrict the erase operation to the victim block. That is, only the CGs of the victim block are set to 0V, and the rest of the blocks are floating.[3] Therefore, only the cells in the victim block are under high negative voltage difference, which resets all the cells in the victim block. In contrast, in the data-erase phase of the partial-erase operation, the erase-inhibition is achieved at a partial-block (PB) granularity, as shown in Figure 7a. In other words, only CG1 of PBs are set to 0V, instead of the entire block as in the baseline bulk-erase implementation. Figure 7a shows the implementation of the partial data-erase for a PB with 4 pages in the 3D NAND flash. In this example, only pages P4-P7 are erased, while the other pages retain their earlier contents as their corresponding CGs are set to floating.

The partial erase-verify phase verifies if all the pages in the PB are erased successfully during the data-erase phase or not. A modified read operation can be used as partial erase-verify. Instead of selecting the page by only one USG and one CG for read operation, multiple USGs and CGs are used to select all pages in a PB to realize the partial erase-verify phase. Then, the current on the bitlines can be measured to figure out the PB cells' erasing status. Figure 7b shows the implementation details of proposed partial erase-verify phase. The partial erase-verify phase is performed on the second 4 pages (P4-P7). The CG1 is set to 0V, so that the content in cells of second 4 pages will reflect to the current on bitlines. The GIDL-based erase can be modified similarly.

The hardware modification for our partial-erase operation mainly adds an extra circuit for the control logic in the page decoder (PD), as illustrated in Figure 8.

[3]The CG is disconnected; so, the voltage of the cells becomes floating.

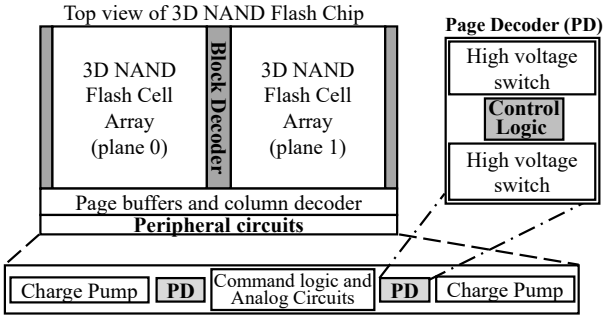


Figure 8: Partial-Erase circuit overview.

The required area for the control logic is at most the same as the original control logic area. The reason is that the proposed circuit has fewer cases to handle than the original one. The area overhead of the proposed enhancements can be calculated as follows: The peripheral circuits in 3D NAND flash chip are around $6 \sim 9\%$[19, 26, 29, 53], and the PD occupies about 4% of the peripheral circuits[47]. In the PD, only 17% of the area is devoted to the control logic[47]. Therefore, our hardware overhead is at most 0.07% of the whole area, which is negligible in a 3D NAND flash chip.

4.2 Boundary Program Disturbance

The additional program disturbance to the neighboring pages [5, 58] comes from later write (program) operation after the partial-erase. In Figure 7a, pages P0-P3 and P8-P11 represent the neighboring pages that are disturbed by the program operations after the partial-erase operation. Such 3D NAND program disturbance is known to be small compared to the 2D counterpart, owing to the 3D CTF cell itself [19]; so, the boundary pages can tolerate more extra program disturbance in 3D NAND compared to 2D. However, the data in those pages can still be corrupted if the same partial-block is repeatedly erased.

4.3 Indexing the Partial Blocks

We augment the original block-level erase operation command format shown in Figure 9 with partial-block

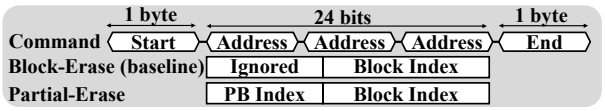


Figure 9: Partial-Erase command format.

index bits. These additional bits, which represent the PB index along with the already-existing block index bits, enable our proposed partial-erase operation.

To minimize the control circuitry and utilize the existing command convention and format used by the NAND flash, we restrict each PB to contain ρ number of pages, which can be expressed as:

$$\rho = \frac{\text{number of pages per block}}{2^l} \qquad \text{(PB size)}$$

,where l represents the number of times that a block is split into two smaller equal-sized partial-blocks. l can take values ranging from 0 to L, where L is the maximum number of times a block can be split. If l is 0, a block is *not* split causing the entire-block to be erased. If l equals to 1, an entire block is split into two partial-blocks so that a partial-erase can be performed on either partial-blocks (PBs). With restricted PB sizes and locations, the number of supported PBs for the chip can only be $2^{L+1}-1$. For example, if L is 6 in a chip containing 576 blocks, the number of pages in a PB can take the following values: 576, 288, 144, 72, 36, 18 and 9 pages. Hence, the possible PBs per block can be 127 $(1+2+4+8+16+32+64 = 2^{6+1}-1 = 127)$.

Figure 7c shows our PB indexing scheme. Instead of indexing a PB in a block with both PB size and location, we use a single PB-index to identify both the PB size and location. Thus, a tuple (block-index, PB-index) is used to identify the unique PB amongst multiple blocks in the 3D NAND flash. For example, PB $(426,2)$ contains two smaller PBs, PB $(426,4)$ and PB $(426,5)$. In addition, erasing one bigger PB is less expensive than erasing two corresponding smaller PBs, since the NAND chip can only execute one command at a time due to its internal control circuitry. As a result, the PB size used during GC highly affects the GC latency.

5 FTL for Partial-Erase

5.1 Partial Block vs. Smaller Block

The simplest approach to utilize partial-erase operation in current FTLs is to replace the block-erase directly with the partial-erase; so, the block size in new system shrinks to one of the possible PB size, which is fixed and cannot be dynamically changed. Although this option is feasible, it has two main drawbacks: (1) larger mapping table size and (2) fixed partial-erase granularity. Enlarging the mapping table size in modern block-level and hybrid FTLs are costly. For example, in the extreme case, where L is 6, 2^6 times of baseline mapping table size is required. On the other hand, employing a fixed partial-erase granularity can result in sub-optimal performance as:

• A finer fixed partial-erase granularity might necessitate more number of partial-erase operations to reclaim a large number of invalid pages in the victim block. These partial-erase overheads can be reduced by fewer coarser granularity partial-erase operations.

• A coarser fixed partial-erase granularity might result in more number of valid page copies.

Motivated by this, we introduce our M-merge algorithm which can keep original mapping table size and dynamically choose the optimal partial-erase granularity.

5.2 M-Merge for Block-level Mapping

We propose a new GC algorithm, **M-Merge**, for NFTL. Figure 10 shows the difference between the baseline Merge algorithm and our proposed M-Merge algorithm. Our M-Merge algorithm is based on a sub-operation called restore. **Restore** is an operation which is performed on one of the PBs in D-block and its corresponding valid pages in U-block. It is composed of three stages: (1) valid-page copy from D-block to U-block , (2) partial-erase of PB in D-block, and (3) valid-page copy from U-block to D-block. In the first stage, all the valid pages in the PB must be copied to the U-block or other blocks, so that a partial-erase can be performed for the PB which has only invalid and free pages in the second stage. After the PB is partial-erased, the third stage copies back the valid pages that belong to this PB from U-block. The cost of the restore operation is one partial-erase and several valid page copies corresponding to this PB. The cost of the restore operations in Figure 10 can be calculated as follows:

$$\begin{aligned} restore(PB\ 5) &= None(skipped) \\ restore(PB\ 9) &= 1\text{ PE} + 72\text{ page copies} \\ restore(PB\ 14) &= 2\text{ page copies} + 1\text{ PE} + 72\text{ page copies} \end{aligned}$$

5.2.1 M-Merge Examples

In the example shown in Figure 10, assuming that there are a total of 576 (434 in D-block + 142 in U-block) valid pages, when a GC is triggered, all these valid pages from D-block and U-block need to be copied to a free block. Hence, the cost of a GC merge operation in the baseline is the cost of copying these 576 pages to a free block along with the cost of erasing both the D- and U-blocks. However, our M-Merge algorithm uses two restore sub-operations to achieve the same goal.

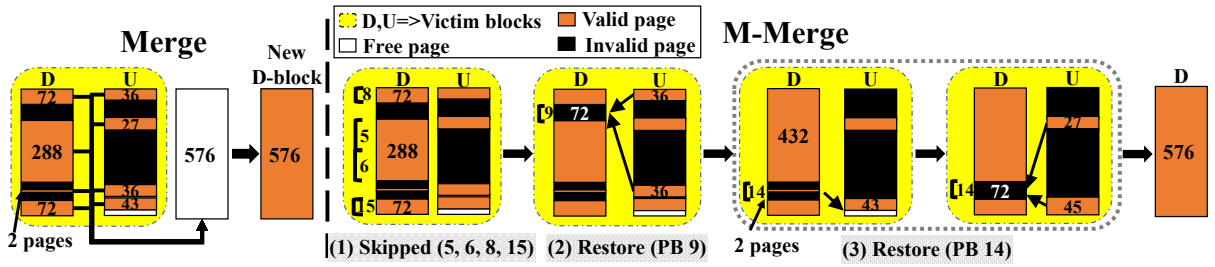


Figure 10: Merge and M-Merge operations.

M-Merge only executes the restore operations on D-blocks, while U-blocks are still block-erased in the end. We use D-block PB indices (Figure 7c) of 8, 9, 5, 6, 14, and 15 to explain M-merge. Note that the non-overlapped PBs form a complete block; hence, restoring all the non-overlapped PBs can guarantee that the U-block is erasable. We arrive at these PB indices by applying our Algorithm 1, which will be discussed shortly. As can be observed from Figure 10, PBs 8, 5, 6 and 15 have no invalid pages; so, M-Merge skips those PBs. PBs 9 and 14 contain invalid pages, so two restore operations need to be performed. PB 9 has only invalid pages, and hence, the first stage of the restore operation is skipped, and only the following two stages are executed. Thus, the corresponding 72 (36 + 36) valid pages in the U-block are copied back to PB 9. On the other hand, PB 14 has two valid pages, and as a result, those two pages have to be copied to the U-block in the first stage, and then the last two stages can be performed aiming PB 14. Overall, the total cost of M-Merge is the time taken for copying 72 + 2 + 72 = 146 valid pages, partial-erasing 2 PBs, and erasing 1 U-block.

Typically, a NAND flash read/write is 10 times faster than an erase operation [1]. The speedup brought by our proposed M-Merge operation over the conventional Merge operation is:[4]

$$speedup = \frac{(Merge\ time)}{(M\text{-}Merge\ time)} = \frac{(576+20)}{(146+20+10)} = 3.38\times.$$

M-Merge with Program Disturbance: We assume that the previous example executes 4 times; as a result, PB 9 is restored 4 times by M-merge, which is shown in Figure 11. As discussed in Section 4.2, the pages adjacent to PB 9 are disturbed when PB 9 is restored. To prevent the data corruptions caused by the disturbances, those disturbed pages will be restored by the next subsequent M-Merge operation that disturbs those pages. Note that the smallest PB has only 9 pages. At time $t1$, the PB 9 is restored. Thus, the upper PB (9 pages) and the lower PB (9 pages) are disturbed, but the data in those two layers are still consistent (not corrupted). At time $t2$, although only PB 9 is required to be restored, the disturbed pages (upper and lower PBs) are restored as well, since the data in these PBs may be corrupted due to this partial-erase operation. Hence, $9+72+9$ pages are restored. At time $t3$, only PB 9 is restored, since both upper and lower PBs are corrected by the previous restore operation at time $t2$. At time $t4$, two upper PBs, two lower PBs and PB 9 are restored to prevent the data corruption.

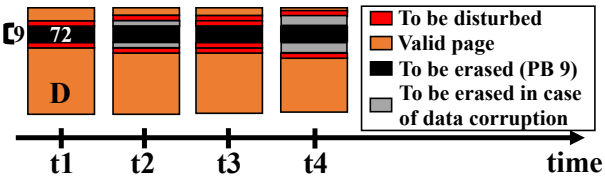


Figure 11: Data corruption prevention.

In summary, M-merge restores the possibly corrupted boundary pages to prevent the data corruption. Although this proposal increases the number of copied pages during M-merge, it still outperforms the baseline merge algorithm, which can be observed in Figure 15a.

M-merge wear-leveling: In the example shown in both Figure 10 and Figure 11, the PBs in the same block may be under different number of erase operations due to partial-erase operation. To mitigate wear-unleveling, the approach in [30] is adopted. To be more specific, a block can only be M-merged W times before a baseline Merge, where W is a (preset) wear-leveling parameter.

5.2.2 M-Merge Algorithm

$$cost[pb] = Min(restore(pb, disturb), \\ cost[pb*2] + cost[pb*2+1]) \quad (1)$$

Our M-Merge algorithm can be accomplished through various sequences of restores. However, the sequence that yields the "minimum" cost and preventing data corruption is preferred. To find such sequence, the recursive relationship in Equation (1) is introduced, which estimates the cost, represented by $cost(pb)$, of using restore for PB index (pb), comparing it with the total cost incurred for the PB indices ($pb*2$) and ($pb*2+1$). Then,

[4]The computation time can be omitted, since the execution time of NAND flash erase operation is on a millisecond scale.

the one which offers the minimum cost is chosen. Note that the restore operation is aware of disturbance, so all of the valid pages in the PB may be copied out and back in case of a data corruption, even though the PB has no invalid pages.

When a GC is triggered, the total cost for M-Merge operation is recursively estimated using Equation (1) starting with the whole block, which is PB 1. The estimated total cost for M-Merge operation is compared with that of baseline Merge operation. Based on the relative costs, the less expensive option is chosen. Note that, *our GC algorithm can switch between Merge and M-Merge operations dynamically based on the relative costs*, thus it is highly adaptive.

Algorithm 1: M-MERGE PLAN ALGORITHM

```
   Input: Dblk: D-block, dis: disturbed pbs in D-block
1  for pb ← max_pb to 1 do                                   // All PBs
2      cost[pb] ←RESTORE(Dblk, pb, dis);
3      trav[pb] ← leaf_PB;
4  for pb ← (max_pb/2) to 1 do              // Except the smallest PBs
5      if cost[pb∗2]+cost[pb∗2+1] < cost[pb] then // Equation (1)
6          cost[pb] ← cost[pb∗2]+cost[pb∗2+1];
7          trav[pb] ← internal_PB;
8  RstrSeq ← DFS-TRAVERSAL-LEAF-PB(cost, trav, 1);
9  toCopy ← SUM-OF-COPY(RstrSeq);
10 return (cost[1], RstrSeq, toCopy)
```

Algorithm 1 gives the pseudo-code that determines the minimum-cost restore sequence to perform M-Merge. After estimating the cost for M-Merge, the number of copied pages, which represents the number of pages to be copied in the first stage of all the restore operations, is calculated.

Algorithm 2: NFTL MERGE MODIFICATION

```
   Input: Dblk, Ublk
1  dis ← φ; corrupt ← True;
2  cost ← MERGE-COST(Dblk);
3  do                                                    // M-Merge cost
4      (mcost, RstrSeq, toCopy) ← M-MERGE-PLAN(Dblk, dis);
5      corrupt ← DISTURB-UPDATE(Dblk, RstrSeq, dis);
6  while corrupt ≠ True;
7  free_u ← FREE-PAGE(Ublk);
8  if free_u < toCopy then
9      (space, pb_u) ← LARGEST-INVALID-PB(Ublk);
10     mcost += PARTIAL-ERASE-TIME(pb_u);
11 if mcost < cost && toCopy < free_u + space &&
     Dblk.mmerge_count < W then                          // M-Merge
12     if free_u < toCopy then
13         PARTIAL-ERASE(pb_u);
14     for pb in RstrSeq do
15         DO-RESTORE(pb);
16 else                                                 // Baseline Merge
17     MERGE(Dblk);
```

Algorithm 2 presents the modifications proposed for the NFTL Merge operation. It initially calculates the costs for both the Merge and M-Merge operations, and then chooses the one with the lower cost. To prevent potential data corruption due to M-merge, Algorithm 2 is repeatedly called with the updated *PB*s' disturbance information. If NFTL decides to execute M-Merge based on the *toCopy* pages returned from Algorithm 1, a partial block in U-block may be erased to ensure that the restore sequence can be executed successfully.

M-Merge can generate the optimal restore sequence with minimum cost using Algorithm 1; however, executing the restore sequence is not guaranteed to be optimal due to insufficient free pages in U-block. In Algorithm 2, to provide more free pages for restore operations, we partial-erase the largest PB with all invalid pages before any restore operations. However, this PB in U-block may not be the optimal one, since a bigger PB may be generated during the execution of the restore sequence, not before it. In addition, the free pages in D-block and the newly-allocated block can also be used as temporary free pages. However, looking for these possibilities would take too much compute time. Hence, we choose the method in Algorithm 2.

5.3 M-Merge for Hybrid Mapping

Before describing our modifications to Superblock FTL, we briefly go over Superblock FTL [21]. In a superblock implementation, several adjacent logical blocks (say M), which is the basic unit of address mapping, are grouped to form a *superblock*, and each superblock contains several (physical) blocks (say N). The GC of Superblock FTL also employs the Merge operation at a block granularity. The GC for a superblock can be divided into intra- and inter-superblock GC.

Intra-superblock GC is triggered when a superblock has no free pages. The goal of the intra-superblock GC is to clear some free blocks for the subsequent write requests. Therefore, only the blocks with the minimum valid pages are merged by GC, and consequently, the read/write requests will not be stalled for too long.

Inter-superblock GC is triggered when there is no available free block in the NAND chip. The goal of the inter-superblock GC is to compact the victim superblock to the fewest number of physical blocks, which has only M physical blocks, so that the other superblocks can allocate available free blocks.

Superblock FTL can apply modifications similar to NFTL to reduce valid page copies by applying our M-Merge algorithm multiple times to physical blocks in a superblock. However, since the concept of D-block in the Superblock FTL is the cold data block (not the blocks to be restored), our modification must choose the D- and U-blocks using by M-Merge amongst the physical blocks in a superblock. Another important difference between NFTL and Superblock FTL M-Merge implementations

Algorithm 3: SUPERBLOCK MERGE MODIFICATION

```
Input: blkset: superblock physical block index set
cost ← SUPERBLOCK-MERGE-COST(blkset);
U-blkset ← BLK-SET-WITH-MIN-VALID-PAGES(blkset, M−N);
D-blkset ← blkset − U-blkset;
mcost ← 0;
for b in D-blkset do                                   // M-Merge cost
    dis ← φ; corrupt ← True;
    do
        (mcost_b, RstrSeq_b, toCopy_b) ← M-MERGE-PLAN(b, dis);
        corrupt ← DISTURB-UPDATE(b, RstrSeq, dis);
    while corrupt ≠ True;
    mcost += mcost_b;
if cost < mcost && blkset.mmerge_count ≤ W then // Baseline Merge
    SUPERBLOCK-MERGE(blkset);
else                                                   // M-Merge
    for b in D-blkset do
        if FREE-PAGE(U-blkset) < toCopy_b then
            ALLOC-FREE-BLOCK(U-blkset);
        for pb in RstrSeq_b do
            DO-RESTORE(pb);
```

is that the restore operation in Section 5.2 assumes the pages in D-block are in address order. However, they are out-of-order across the physical blocks in a superblock; as a result, the cost to perform restores needs to be adjusted accordingly.

Algorithm 3 gives the pseudo-code to determine whether it is beneficial to execute M-Merge and, if it is, how to perform M-Merge. The first step of Algorithm 3 is to decide *U-blkset* – the blocks to be erased. We pick $M-N$ blocks in the victim superblock with the minimum number of valid pages as *U-blkset*, so that the number of valid pages to be copied is minimal. The second step involves applying Algorithm 1 to all the blocks in *D-blkset* and estimating the total cost of M-Merge. The final step involves deciding whether it is beneficial to perform M-Merge or conventional Merge.

6 Evaluation

6.1 Experimental Setup

We used the "Flash core cell" model from HSIM [49] package in Synopsys HSPICE [50] to measure the partial-erase latency (in milliseconds) for a 3D NAND flash. Since the partial-erase latencies are governed by the cell with the slowest erase-rate and not by the number of cells per partial-block, the partial-erase latencies are only slightly better compared to the block-erase latency. Please refer to Table 1 for the details on the latencies. We used SSDSim [17] to evaluate our proposed M-merge algorithm for NFTL and Superblock FTL. Various parameters used in our SSDSim experiments are listed in Table 1. The read/write and block-level erase access latencies used in our SSDSim simulations are based on our empirical evaluations of the *real* 3D NAND chips. The 12 evaluated I/O workloads, whose characteristics are shown in Table 2, are from the revised SNIA traces [37].

SSD parameters	
(Page-read, Page-program, Block-erase)	(70μs, 900μs, 10ms)
(PB size (pages), partial-erase time (ms)) (L=6)	(288, 9.95), (144, 9.79), (72, 9.62), (36, 9.48), (18, 9.37), (9, 9.27)
(Channels, Chips, Dies, Planes, Blocks, Pages)	(8, 2, 2, 2, 1888, 576)
(Page Size, Spare area size)	(16KB, 1280B)
Total SSD capacity	1TB
(Over provision, Initial data)	(10%, 95%)
Number of tolerance disturbance	1
Wear leveling parameters (W)	16
NFTL parameters	
(Victim selection, GC free block threshold)	(Max invalid, 8%)
Superblock FTL parameters	
Logical:physical block ratio (M:N)	(4:5), (4:8)
Intra-/Inter-superblock GC threshold	(1 PBMT, 8%)

Table 1: Characteristics of the evaluated SSD.

trace	read reqs (in millions)	write reqs (in millions)	read data (in GBs)	write data (in GBs)	read coverage (in GBs)	write coverage (in GBs)
hm_0	1.417	2.576	9.96	20.47	1.84	1.63
prn_0	0.602	4.983	13.12	45.96	3.72	12.38
prn_1	8.464	2.77	181.35	30.78	73.78	11.52
proj_0	0.527	3.697	8.97	144.26	1.74	1.65
proj_1	21.143	2.497	750.36	25.57	693.5	9.03
proj_2	25.642	3.625	1015.9	168.68	409.37	115.13
prxy_0	0.384	12.135	3.04	53.8	0.29	0.7
src1_1	43.576	2.17	1485.6	30.34	116.69	4.16
src1_2	0.484	1.424	8.82	44.14	1.55	0.65
stg_1	1.4	0.796	79.52	5.98	79.42	0.39
usr_1	41.426	3.858	2079.2	56.12	651.16	24.56
usr_2	8.575	1.995	415.28	26.46	377.8	10.02

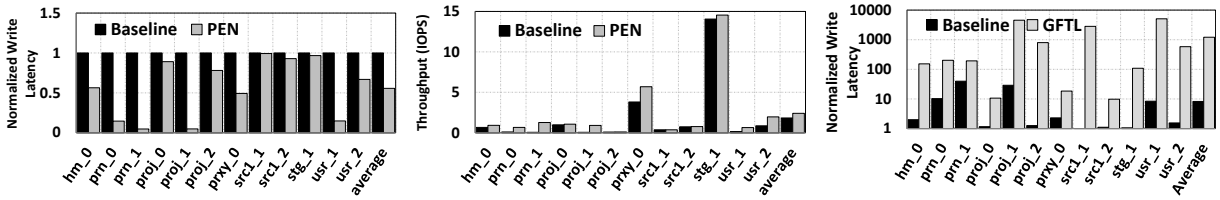
Table 2: Important characteristics of our workloads.

We use the following five metrics for our evaluations: (a) **average write latency**, (b) **throughput**, (c) **write amplification**, (d) **AEP**, and (e) **VEP**. AEP and VEP are the average and variance number of erase operations per page, respectively. Those two metrics track the number of erase operations at a finer granularity, page, unlike the coarse block granularity in the baseline. This is because, due to the partial-erase, the different pages in the same block can experience a different number of erases.

6.2 Experimental Results

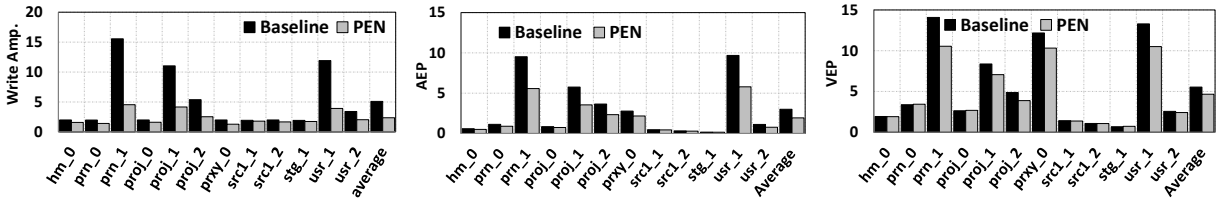
6.2.1 Block-Level FTL

Performance: Figure 12b shows the improvement in read/write throughput (IOPS) for our proposed PEN system (using partial-erase) over the baseline system (using block-erase only) for NFTL. Note that NFTL maintains the mapping at a unit of a block instead of a page; so, the GC trigger frequency is relevant not only to the ratio of the written data and the SSD capacity, but also to the page utilization of blocks; hence, a small amount of written data may trigger a large number of block merges.



(a) Normalized write latency for NFTL. (b) Throughput for NFTL. (c) Comparison with GFTL.

Figure 12: Performance improvements in the case of NFTL.



(a) Write amplification for NFTL. (b) AEP for NFTL. (c) VEP for NFTL.

Figure 13: Write amplification, AEP, and VEP improvements in the case of NFTL.

For example, although workload prn_1 only comprises of 30.78GB of written data, over 140,000 block merges are performed. Since M-merge can significantly reduce block merge overhead, the performance of the workloads with frequent block merge operations can be highly improved. On average, IOPS is improved by 1.43x over the baseline system in NFTL.

Figure 12a plots the write access latencies for our PEN system, *normalized* to the baseline system. The magnitude of improvement in the IOPS and write latencies is a function of the workload characteristics. For example, workloads like proj_1, usr_1, and prn_1, which have relatively high amounts of coverage (unique data), experience very high improvements. This is because, as the coverage in these workloads is very high, a majority of the U-blocks are already paired with a D-block resulting in an outage of free U-blocks. When a write is incurred to a page in an unpaired D-block, a block merge is triggered to reclaim a free U-block that can be paired with this D-block. Hence, for these workloads, the number of triggered block merges is very high, with each merge operation lasting several hundreds of milliseconds, owing to the increased number of valid page copies in the baseline. Since our M-Merge algorithm reduces the number of valid pages to be copied, our PEN system yields significant improvements in I/O throughput.

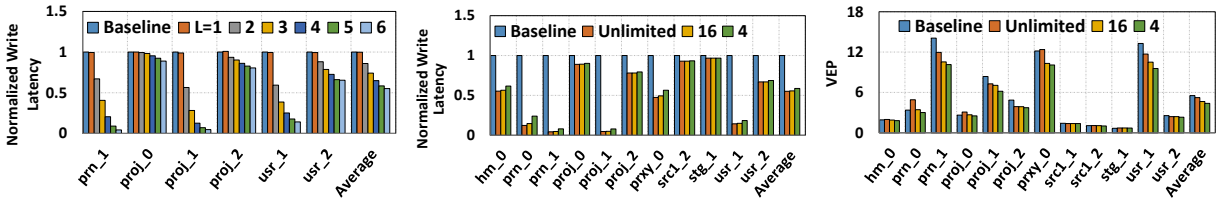
Write amplification: Figure 13a shows how our partial-erase enabled PEN system compared to the baseline system in terms of write-amplification. The write-amplification is reduced, on average, by 2.67x, compared to the baseline.

AEP and VEP: Figures 13b and 13c show the AEP and VEP, respectively. AEP improvements stem from the fewer erased pages during M-Merge, thanks to the partial-erase. On the contrary, VEP improvements result from the wear-leveling technique [30], the detailed sensitivity results of which can be found in Figure 14c.

GFTL comparison: Figure 12c plots the write latencies for the baseline and GFTL [9], *normalized* to our proposed PEN system. Note that GFTL is one kind of partial GC algorithm, which needs to reserve extreme long time for the block merge overhead to guarantee the constant request response time in the "Big Block". The figure shows that the GC algorithms designed for 2D NAND *cannot* directly be applied to the 3D NAND.

Sensitivity Results: Figure 14a plots the write latencies for our PEN system for different possible PB sizes (governed by L). As the possible number of PB sizes increases, PEN performs better, since a smaller PB reduces the copied valid pages in a PB during a block merge, ultimately reducing the overall block merge overheads.

Figures 14b and 14c plot the performance and reliability impact of the wear-leveling parameter, W, which is defined as the number of M-merge operations that can be performed on a block since the last baseline Merge for the block. A higher W value can provide better performance; but, it can also cause the severe skewness on erase count per page, which can be observed for prn_1, proj_0, and prxy_0 workload. A lower W value addresses the unevenness issue; but, it reduces the magnitude of performance improvement. As can be observed, W value 16 results in optimal performance and reliability. Figure 15a shows the write latency with and without considering the boundary program disturbance by the partial-erase operation. Our program-disturbance aware M-merge slightly increases the average write latency even under the most severe scenario where the NAND cells can only tolerate one partial-erase from the

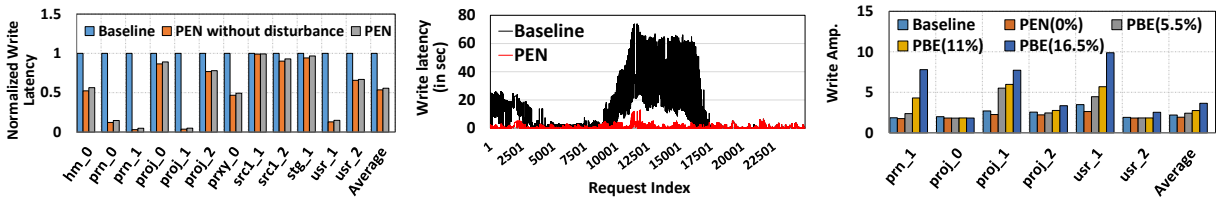


(a) Normalized write latency with different number of PB sizes for NFTL.

(b) Normalized write latency with different wear leveling parameters for NFTL.

(c) VEP with different wear leveling parameters for NFTL.

Figure 14: Sensitivity results in the case of NFTL.



(a) Normalized write latency with/without disturbance for NFTL.

(b) proj_1 write request time graph for NFTL.

(c) Write amplification comparison with [11].

Figure 15: Other aspects in the case of NFTL.

neighbor cells.

Variation of write latency over time: Figure 15b plots the write access latencies incurred by various I/O requests as they are processed in one of our workloads (proj_1). This graph clearly demonstrates that almost all the I/O write requests[5] experience reduced write access latency as they do *not* incur long stall times in the I/O Queue in our PEN system, unlike the baseline. As a result, we observe improvements in throughput.

Comparison with partial block erase (PBE) in [11]: Figure 15c compares PBE and PEN system. PBE enables the partial-erase by employing additional erase circuits between partial-blocks. Such implementation can mitigate the program-disturbance problem, but reduces the total available capacity. The detailed hardware and FTL modifications are not provided; as a result, we modeled the PBE system as a reduced capacity PEN system without program disturbance. The effect of loss in capacity by PBE can be observed in the reduced capacity change in the step from 0% to 16.5%. As a result, it incurs more GCs, and thus, PBE increases write amplification by 88% compared to PEN.

6.2.2 Hybrid FTL

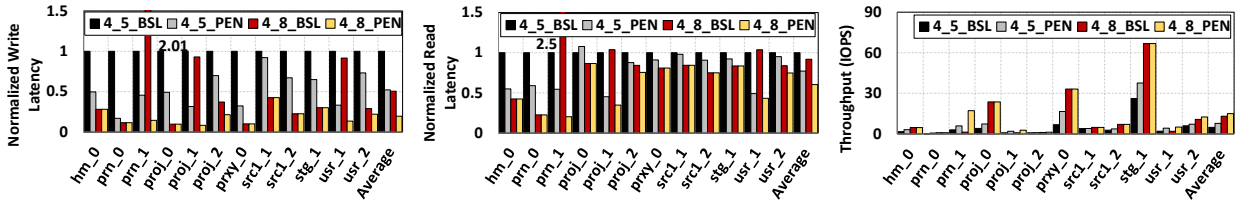
We now quantify the benefits of PEN over the baseline intra-/inter-superblock GC for Superblock FTL. We present the results for two different configurations, where the first one uses 4 logical blocks and 5 (physical) blocks, while the second one uses 4 logical blocks and 8 (physical) blocks. Figures 16a and 16b plot the normalized (with respect to the baseline) write and read latencies for the two configurations mentioned before. Workloads like proj_1 and usr_1 experience improved read/write latencies and enhanced throughput, due to the same reason explained in Section 6.2.1. In contrast, other workloads, especially hm_0 and stg_1, incur fewer inter-superblock GC; however, they incur more frequent intra-superblock GCs. Since the intra-superblock GC algorithm chooses the block in the superblock with the minimum number of valid pages as the victim block, not many valid pages need to be copied in the baseline. Hence, the avenues to improve the access latencies in PEN are minimized. Therefore, the cases in which PEN can outperform the intra-superblock GC are when the number of (physical) blocks is close to the number of logical blocks. Note that only prn_1 workload shows performance degradation in the baseline with more physical blocks, since the GC operations in the latter case are dominated by inter-superblock GC. Our PEN also reduces the write-amplification, AEP and VEP (Figures 17a, 17b, and 17c).

[5] We saw similar results in other workloads as well; but, we could not present them due to space constraints.

7 Related Work

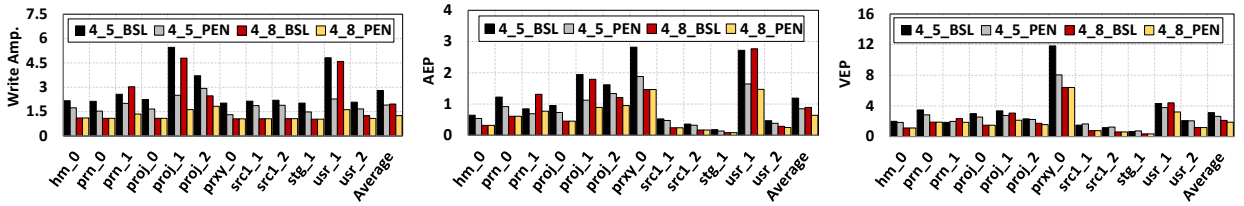
Partial-Erase proposals: Partial-Erase operation has been proposed [28] for 2D NAND flash, however, it did not pave way in to the actual products as it did not improve performance or reliability for 2D NAND flash. This is because the "Big block" problem is not as severe in 2D NAND compared to 3D NAND flash.

Partial-Erase operations for 3D NAND have recently been proposed in [11, 14]. However, due to the implementation diversity of 3D NAND, there are multiple ap-



(a) Normalized write latency. (b) Normalized read latency. (c) Throughput.

Figure 16: Performance improvements in the case of Superblock FTL. (BSL=Baseline)



(a) Write amplification. (b) AEP. (c) VEP.

Figure 17: Write amplification, AEP, and VEP improvements in the case of Superblock FTL. (BSL=Baseline)

proaches to enable such an operation. Partial block erase (PBE) [11] is discussed and compared against PEN in Section 6.2.1. Another implementation, subblock management [14] only allows three subblocks (two kinds of PB sizes) in a block. Such an implementation cannot provide any significant performance improvement, which can be observed in Figure 14a.

Besides the hardware-based partial-erase proposals, there also exist software-based proposals. Kim [30] proposes the strategy to address the wear-leveling problem caused by the partial-erase operation. Subblock erase [8] proposes a page-level FTL modification for the partial-erase operation in [11] to alleviate the "Big Block" problem. However, this proposal assumes that multiple partial-erase operations can be executed simultaneously, which necessitates non-trivial modifications to the underlying peripheral circuit components of the current NAND chips. In comparison, our PEN necessitates modest changes to the current peripheral circuitry. More importantly, the approach in [8] is only applicable to page-level FTL, which, as discussed before in Section 2.2.1, will become *impractical* in 3D NAND. In comparison, our approach focuses on the basic building block of GC, that is, the merge operation.

Partial GC proposals: We now compare our proposal to the "partial GC" research [7, 9, 23, 25] , conducted in the context of 2D NAND flash. Chang et al. [7] and Choudhari et al. [9] proposed periodic partial GC operations for real-time systems so that they provide minimal performance guarantees. GFTL [9] is discussed and compared in Section 6.2.1 and Figure 12c, respectively. AGCDGC and HIOS [23, 25] divide and distribute GC into more free-time slots, considering the address mapping and I/O request queue information, so that an SSD can have a more stable performance. Since these partial GC algorithms require additional knowledge to estimate free-time slots, they are FTL-specific and are not generic unlike our proposal. In addition, these partial GC algorithms still use coarse "block-level" erase operations, resulting in unnecessary valid pages copies during a GC operation, unlike our PEN.

8 Conclusion

In this paper, we propose and evaluate a novel partial-erase based PEN architecture in emerging 3D NAND flashes, which minimizes the number of valid pages copied during a GC operation. To show the effectiveness of our proposed partial-erase operation, we introduce our M-Merge algorithm that employs our partial-erase operation for NFTL and Superblock FTL. Our extensive experimental evaluations show that the average write latency under the proposed PEN system is reduced by 44.3% – 47.9%, compared to the baseline.

9 Acknowledgement

This research is supported by NSF grants 1439021, 1439057, 1409095, 1626251, 1629915, 1629129 and 1526750, and a grant from Intel. Dr. Jung is supported in part by NRF 2016R1C1B2015312, DOE DE-AC02-05CH 11231, IITP-2017-2017-0-01015, NRF-2015M3C4A7065645, and MemRay grant (2015-11-1731). Kandemir and Jung are the co-corresponding authors. The authors thank Prof. Youjip Won for shepherding this paper.

References

[1] Micron mt29f8g08baa datasheet. https://www.micron.com/products/nand-flash/, Feb. 2007.

[2] AGARWAL, R., AND MARROW, M. A closed-form expression for write amplification in NAND flash. In *GLOBECOM Workshops (GC Wkshps), 2010 IEEE* (2010), pp. 1846–1850.

[3] BAN, A. Flash file system. https://www.google.com/patents/US5404485, Apr. 4 1995. US Patent 5,404,485.

[4] BOOTH, J. D., KOTRA, J. B., ZHAO, H., KANDEMIR, M., AND RAGHAVAN, P. Phase detection with hidden markov models for dvfs on many-core processors. In *2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS)* (2015).

[5] CAI, Y., MUTLU, O., HARATSCH, E. F., AND MAI, K. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *2013 IEEE 31st International Conference on Computer Design (ICCD)* (2013), IEEE.

[6] CAULFIELD, A. M., AND SWANSON, S. Quicksan: A storage area network for fast, distributed, solid state disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013).

[7] CHANG, L.-P., KUO, T.-W., AND LO, S.-W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* (Nov. 2004).

[8] CHEN, T.-Y., CHANG, Y.-H., HO, C.-C., AND CHEN, S.-H. Enabling sub-blocks erase management to boost the performance of 3d NAND flash memory. In *Proceedings of the 53rd Annual Design Automation Conference* (2016), DAC '16.

[9] CHOUDHURI, S., AND GIVARGIS, T. Deterministic service guarantees for nand flash using partial block cleaning. In *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (2008), CODES+ISSS '08.

[10] CO., S. E. Samsung V-NAND. http://www.samsung.com/semiconductor/products/flash-storage/v-nand/, 2016.

[11] D'ABREU, M. A. Partial block erase for a three dimensional (3d) memory. https://www.google.tl/patents/US9286989, May 19 2015. US Patent 9,286,989.

[12] DESNOYERS, P. Analytic Models of SSD Write Performance. *ACM Transactions on Storage*, 2 (Mar. 2014), 1–25.

[13] DIRIK, C., AND JACOB, B. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (2009), ISCA '09.

[14] EUN CHU OH, J. K. Nonvolatile memory device and sub-block managing method thereof. https://www.google.com/patents/US20140063938, Mar. 6 2014. US Patent 2014/063938.

[15] GUPTA, A., KIM, Y., AND URGAONKAR, B. Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009).

[16] HU, X.-Y., ELEFTHERIOU, E., HAAS, R., ILIADIS, I., AND PLETKA, R. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), p. 10.

[17] HU, Y., JIANG, H., FENG, D., TIAN, L., LUO, H., AND ZHANG, S. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing* (2011).

[18] HUANG, J., BADAM, A., QURESHI, M. K., AND SCHWAN, K. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture* (2015).

[19] IM, J.-W., JEONG, W.-P., KIM, D.-H., NAM, S.-W., SHIM, D.-K., CHOI, M.-H., YOON, H.-J., KIM, D.-H., KIM, Y.-S., PARK, H.-W., AND OTHERS. 7.2 A 128gb 3b/cell V-NAND flash memory with 1gb/s I/O rate. In *2015 IEEE International Solid-State Circuits Conference-(ISSCC) Digest of Technical Papers* (2015), IEEE.

[20] JANG, J., KIM, H. S., CHO, W., CHO, H., KIM, J., SHIM, S. I., YOUNGGOAN, JEONG, J. H., SON, B. K., KIM, D. W., KIHYUN, SHIM, J. J., LIM, J. S., KIM, K. H., YI, S. Y., LIM, J. Y., CHUNG, D., MOON, H. C., HWANG, S., LEE, J. W., SON, Y. H., CHUNG, U. I., AND LEE, W. S. Vertical cell array using tcat(terabit cell array transistor) technology for ultra high density nand flash memory. In *2009 Symposium on VLSI Technology* (June 2009).

[21] JUNG, D., KANG, J.-U., JO, H., KIM, J.-S., AND LEE, J. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Transactions on Embedded Computing Systems* (Mar. 2010).

[22] JUNG, M. Exploring parallel data access methods in emerging non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems 28*, 3 (March 2017), 746–759.

[23] JUNG, M., CHOI, W., SRIKANTAIAH, S., YOO, J., AND KANDEMIR, M. T. Hios: A host interface i/o scheduler for solid state disks. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014).

[24] JUNG, M., AND KANDEMIR, M. Revisiting widely held ssd expectations and rethinking system-level implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems* (2013), SIGMETRICS '13.

[25] JUNG, M., PRABHAKAR, R., AND KANDEMIR, M. T. Taking garbage collection overheads off the critical path in ssds. In *Proceedings of the 13th International Middleware Conference* (2012).

[26] KANG, D., JEONG, W., KIM, C., KIM, D. H., CHO, Y. S., KANG, K. T., RYU, J., KANG, K. M., LEE, S., KIM, W., LEE, H., YU, J., CHOI, N., JANG, D. S., IHM, J. D., KIM, D., MIN, Y. S., KIM, M. S., PARK, A. S., SON, J. I., KIM, I. M., KWAK, P., JUNG, B. K., LEE, D. S., KIM, H., YANG, H. J., BYEON, D. S., PARK, K. T., KYUNG, K. H., AND CHOI, J. H. 7.1 256gb 3b/cell V-NAND flash memory with 48 stacked WL layers. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)* (Jan. 2016).

[27] KATSUMATA, R., KITO, M., FUKUZUMI, Y., KIDO, M., TANAKA, H., KOMORI, Y., ISHIDUKI, M., MATSUNAMI, J.,

FUJIWARA, T., NAGATA, Y., ZHANG, L., IWATA, Y., KIRISAWA, R., AOCHI, H., AND NITAYAMA, A. Pipe-shaped bics flash memory with 16 stacked layers and multi-level-cell operation for ultra high density storage devices. In *2009 Symposium on VLSI Technology* (June 2009).

[28] KI KIM, J. Partial block erase architecture for flash memory. `https://www.google.com/patents/US7804718`, Sept. 28 2010. US Patent 7,804,718.

[29] KIM, C., CHO, J. H., JEONG, W., PARK, I. H., PARK, H. W., KIM, D. H., KANG, D., LEE, S., LEE, J. S., KIM, W., PARK, J., AHN, Y. L., LEE, J., LEE, J. H., KIM, S., YOON, H. J., YU, J., CHOI, N., KWON, Y., KIM, N., JANG, H., PARK, J., SONG, S., PARK, Y., BANG, J., HONG, S., JEONG, B., KIM, H. J., LEE, C., MIN, Y. S., LEE, I., KIM, I. M., KIM, S. H., YOON, D., KIM, K. S., CHOI, Y., KIM, M., KIM, H., KWAK, P., IHM, J. D., BYEON, D. S., LEE, J. Y., PARK, K. T., AND KYUNG, K. H. 11.4 A 512gb 3b/cell 64-stacked WL 3d V-NAND flash memory. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)* (Feb. 2017).

[30] KIM, S.-H. Erasing method of non-volatile memory device. `https://www.google.com/patents/US9025389`, May 5 2015. US Patent 9,025,389.

[31] KIM, W., CHOI, S., SUNG, J., LEE, T., PARK, C., KO, H., JUNG, J., YOO, I., AND PARK, Y. Multi-layered vertical gate nand flash overcoming stacking limit for terabit density storage. In *2009 Symposium on VLSI Technology* (June 2009).

[32] KISLAL, O., , KANDEMIR, M. T., AND KOTRA, J. Cache-aware approximate computing for decision tree learning. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2016).

[33] KOTRA, J. B., ARJOMAND, M., GUTTMAN, D., KANDEMIR, M. T., AND DAS, C. R. Re-NUCA: A practical nuca architecture for reram based last-level caches. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2016).

[34] KOTRA, J. B., GUTTMAN, D., CHIDAMBARAM, N., AND KANDEMIR, M. T. Quantifying the potential benefits of on-chip near datacomputing in manycore processors. In *25th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2017).

[35] KOTRA, J. B., KIM, S., MADDURI, K., AND KANDEMIR, M. T. Congestion-aware memory management on numa platforms: A vmware esxi case study. In *IEEE International Symposium on Workload Characterization (IIWSC)* (2017).

[36] KOTRA, J. B., SHAHIDI, N., CHISHTI, Z. A., AND KANDEMIR, M. T. Hardware-software co-design to mitigate dram refresh overheads: A case for refresh-aware process scheduling. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[37] KWON, M., ZHANG, J., PARK, G., CHOI, W., DONOFRIO, D., SHALF, J., KANDEMIR, M., AND JUNG, M. Tracetracker: Hardware/software co-evaluation for large-scale i/o workload reconstruction.

[38] LEE, S., SHIN, D., KIM, Y.-J., AND KIM, J. Last: Locality-aware sector translation for nand flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.* (Oct. 2008).

[39] LEE, S.-W., MOON, B., PARK, C., KIM, J.-M., AND KIM, S.-W. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (2008).

[40] 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. Embed. Comput. Syst.* (July 2007).

[41] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized ssd cache for primary storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (June 2014), USENIX Association.

[42] LI, Y., LEE, P. P., LUI, J. C., AND XU, Y. Impact of Data Locality on Garbage Collection in SSDs: A General Analytical Study. pp. 305–315.

[43] LIU, J., KOTRA, J., DING, W., AND KANDEMIR, M. Network footprint reduction through data access and computation placement in noc-based manycores. In *Proceedings of the 52nd Annual Design Automation Conference (DAC)* (2015).

[44] LUOJIE, X., AND KURKOSKI, B. M. An improved analytic expression for write amplification in NAND flash. In *Computing, Networking and Communications (ICNC), 2012 International Conference on* (2012), pp. 497–501.

[45] PARK, C., LEE, S., WON, Y., AND AHN, S. Practical implication of analytical models for ssd write amplification. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (2017), ICPE '17, pp. 257–262.

[46] PRINCE, B. *Vertical 3D Memory Technologies*. John Wiley & Sons, Inc., 2014.

[47] RINO MICHELONI, LUCA CRIPPA, A. M. *Inside NAND Flash Memory*. Springer Netherlands, 2010.

[48] SWAMINATHAN, K., KOTRA, J., LIU, H., SAMPSON, J., KANDEMIR, M., AND NARAYANAN, V. Thermal-aware application scheduling on device-heterogeneous embedded architectures. In *2015 28th International Conference on VLSI Design* (2015).

[49] SYNOPSYS. Hsim simulation reference manual. `http://www.synopsys.com/Tools/Verification/AMSVerification/CircuitSimulation/HSIM/Pages/default.aspx`, 2016. Synopsys HSIM.

[50] SYNOPSYS. Hspice. `https://www.synopsys.com/tools/Verification/AMSVerification/CircuitSimulation/HSPICE/Pages/default.aspx`, 2016. Synopsys HSpice.

[51] TANAKA, H., KIDO, M., YAHASHI, K., OOMURA, M., KATSUMATA, R., KITO, M., FUKUZUMI, Y., SATO, M., NAGATA, Y., MATSUOKA, Y., IWATA, Y., AOCHI, H., AND NITAYAMA, A. Bit cost scalable technology with punch and plug process for ultra high density flash memory. In *2007 IEEE Symposium on VLSI Technology* (June 2007).

[52] TANG, X., KANDEMIR, M., YEDLAPALLI, P., AND KOTRA, J. Improving bank-level parallelism for irregular applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016).

[53] YAMASHITA, R., MAGIA, S., HIGUCHI, T., YONEYA, K., YAMAMURA, T., MIZUKOSHI, H., ZAITSU, S., YAMASHITA, M., TOYAMA, S., KAMAE, N., LEE, J., CHEN, S., TAO, J., MAK, W., ZHANG, X., YU, Y., UTSUNOMIYA, Y., KATO, Y., SAKAI, M., MATSUMOTO, M., CHIBVONGODZE, H.,

OOKUMA, N., YABE, H., TAIGOR, S., SAMINENI, R., KODAMA, T., KAMATA, Y., NAMAI, Y., HUYNH, J., WANG, S. E., HE, Y., PHAM, T., SARAF, V., PETKAR, A., WATANABE, M., HAYASHI, K., SWARNKAR, P., MIWA, H., PRADHAN, A., DEY, S., DWIBEDY, D., XAVIER, T., BALAGA, M., AGARWAL, S., KULKARNI, S., PAPASAHEB, Z., DEORA, S., HONG, P., WEI, M., BALAKRISHNAN, G., ARIKI, T., VERMA, K., SIAU, C., DONG, Y., LU, C. H., MIWA, T., AND MOOGAT, F. 11.1 A 512gb 3b/cell flash memory on 64-word-line-layer BiCS technology. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)* (Feb. 2017).

[54] YANG, M. C., CHANG, Y. M., TSAO, C. W., HUANG, P. C., CHANG, Y. H., AND KUO, T. W. Garbage collection and wear leveling for flash memory: Past and future. In *Smart Computing (SMARTCOMP), 2014 International Conference on* (Nov 2014).

[55] YANG, Q., AND REN, J. I-cash: Intelligently coupled array of ssd and hdd. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture* (Feb 2011).

[56] YEDLAPALLI, P., KOTRA, J., KULTURSAY, E., KANDEMIR, M., DAS, C. R., AND SIVASUBRAMANIAM, A. Meeting midway: Improving cmp performance with memory-side prefetching. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2013).

[57] YOO, J., WON, Y., KANG, S., CHOI, J., YOON, S., AND CHA, J. Analytical model of ssd parallelism. In *2014 4th International Conference On Simulation And Modeling Methodologies, Technologies And Applications (SIMULTECH)* (Aug 2014), pp. 551–559.

[58] ZHANG, J., PARK, G., SHIHAB, M. M., DONOFRIO, D., SHALF, J., AND JUNG, M. Opennvm: An open-sourced fpga-based nvm controller for low level memory characterization. In *2015 33rd IEEE International Conference on Computer Design (ICCD)* (Oct 2015), pp. 666–673.

The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator

Huaicheng Li, Mingzhe Hao, Michael Hao Tong,
Swaminatahan Sundararaman[†], Matias Bjørling[‡], Haryadi S. Gunawi

University of Chicago [†]Parallel Machines [‡]CNEX Labs

ABSTRACT: *FEMU is a software (QEMU-based) flash emulator for fostering future full-stack software/hardware SSD research. FEMU is cheap (open-sourced), relatively accurate (0.5-38% variance as a drop-in replacement of OpenChannel SSD), scalable (can support 32 parallel channels/chips), and extensible (support internal-only and split-level SSD research).*

1 Introduction

Cheap and extensible research platforms are a key ingredient in fostering wide-spread SSD research. SSD simulators such as DiskSim's SSD model [9], FlashSim [13] and SSDSim [16], despite their popularity, only support internal-SSD research but not kernel-level extensions. On the other hand, hardware research platforms such as FPGA boards [28, 34, 46], OpenSSD [7], or OpenChannel SSD [11], support full-stack software/hardware research but their high costs (thousands of dollars per device) impair large-scale SSD research.

This leaves software-based emulator such as QEMU-based VSSIM [45], FlashEm [47], and LightNVM's QEMU [6], as the cheap alternative platform. Unfortunately, the state of existing emulators is bleak; they are either outdated, non-scalable, or not open-sourced.

We argue that it is a critical time for storage research community to have a new software-based emulator (more in §2). To this end, we present FEMU, a QEMU-based flash emulator, with the following four "CASE" benefits.

First, FEMU is **c**heap ($0) as it will be an open-sourced software. FEMU has been successfully used in several projects, some of which appeared in top-tier OS and storage conferences [14, 43]. We hope FEMU will be useful to broader communities.

Second, FEMU is (relatively) **a**ccurate. For example, FEMU can be used as a drop-in replacement for OpenChannel SSD; thus, future research that extends LightNVM [11] can be performed on top of FEMU with relatively accurate results (*e.g.*, 0.5-38% variance in our tests). With FEMU, prototyping SSD-related kernel changes can be done without a real device.

Third, FEMU is **s**calable. As we optimized the QEMU stack with various techniques, such as exitless interrupt and skipping QEMU AIO components, FEMU can scale to 32 IO threads and still achieve a low latency (as low as 52µs under a 2.3GHz CPU). As a result, FEMU can accurately emulate 32 parallel channels/chips, without unintended queueing delays.

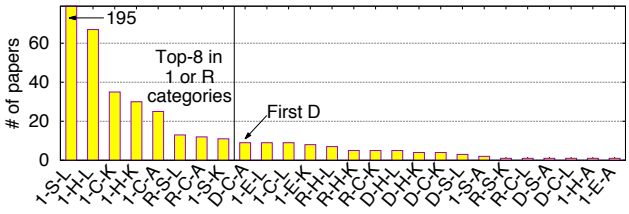


Figure 1: **Categorization of SSD research.** *The figure is explained in Section §2.1. The first bar reaches 195 papers.*

Finally, FEMU is **e**xtensible. Being a QEMU-based emulator, FEMU can support internal-SSD research (only FEMU layer modification), kernel-only research such as software-defined flash (only Guest OS modification on top of unmodified FEMU), and split-level research (both Guest OS and FEMU modifications). FEMU also provides many new features not existent in other emulators, such as OpenChannel and multi-device/RAID support, extensible interfaces via NVMe commands, and page-level latency variability.

2 Extended Motivation

2.1 The State of SSD Research Platforms: We reviewed 391 papers in more than 30 major systems and storage conferences and journals published in the last 10 years, and categorized them as follows:

1. What was the **scale** of the research? **[1]:** single SSD; **[R]:** RAID of SSDs (flash array); or **[D]:** distributed/multi-node SSDs.
2. What was the **platform** being used? **[C]:** commodity SSDs; **[E]:** software SSD emulators (VSSIM [45] or FlashEm [47]); **[H]:** hardware platforms (FPGA boards, OpenSSD [7], or OpenChannel SSD [6]); or **[S]:** trace-based simulators (DiskSim+SSD [9] or FlashSim [13] and SSDSim [16]).

3. What **layer** was modified? **[A]:** application layer; **[K]:** OS kernel; **[L]:** low-level SSD controller logic.

Note that some papers can fall into two sub-categories (*e.g.*, modify both the kernel and the SSD logic). Figure 1 shows the sorted order of the combined categories. For example, the most popular category is **1-S-L**, where 195 papers target only single SSD (**1**), use simulator (**S**), and modify the low-level SSD controller logic (**L**). However, simulators do not support running applications and operating systems.

2.2 The Lack of Large-Scale SSD Research: Our first motivation is the lack of papers in the distributed SSDs category (**D-**...), for example, for investigating the impact of SSD-related changes to distributed computing and graph frameworks. One plausible reason is the cost of managing hardware (procurement, installation, maintenance, etc.). The top-8 categories in Figure 1, a total of 324 papers (83%), target single SSD (**1-**...) and flash array (**R-**...). The highest **D** category is **D-C-A** (as highlighted in the figure), where only 9 papers use commodity SSDs (**C**) and modify the application layer (**A**). The next **D** category is **D-H-L**, where hardware platforms (**H**) are used for modifying the SSD controller logic (**L**). Unfortunately, most of the 6 papers in this category are from large companies with large research budget (*e.g.*, FPGA usage in Baidu [28] and Tencent [46]). Other hardware platforms such as OpenSSD [7] and OpenChannel SSD [6] also cost thousands of dollars each, impairing multi-node non-simulation research, especially in academia.

2.3 The Rise of Software-Defined Flash: Today, research on host-managed (aka. "software-defined" or "user-programmable") flash is growing [25, 28, 34, 35, 41, 46]. However, such research is mostly done on top of expensive and hard-to-program FPGA platforms. Recently, a more affordable and simpler platform is available, OpenChannel SSD [6], managed by Linux-based LightNVM [11]. Before its inception (2015), there were only 24 papers that performed kernel-only changes, since then, 11 papers have been published, showing the success of OpenChannel SSD.

However, there remains several issues. First, not all academic communities have budget to purchase such devices. Even if they do, while prototyping the kernel/application, it is preferable not to write too much to and wear out the device. Thus, replacing OpenChannel SSD (during kernel prototyping) with a software-based emulator is desirable.

2.4 The Rise of Split-Level Architecture: While most existing research modify a single layer (application/kernel/SSD), some recent works show the benefits of "split-level" architecture [8, 19, 24, 38, 42], wherein some functionalities move up to the OS kernel (**K**) and some other move down to the SSD firmware (**L**) [18, 31, 36]. So far, we found only 40 papers in split-level **K+L** category (*i.e.*, modify *both* the kernel and SSD logic layers), mostly done by companies with access to SSD controllers [19] or academic researchers with Linux+OpenSSD [21, 32] or with block-level emulators (*e.g.*, Linux+FlashEm) [29, 47]. OpenSSD with its single-threaded, single-CPU, whole-blocking GC architecture also has many known major limitations [43]. FlashEm also has limitations as we elaborate more below. Note that the kernel-level LightNVM is not a suitable platform for split-level research (*i.e.*, support **K**, but not **L**). This is because its SSD layer (*i.e.*, OpenChannel SSD) is not modifiable; the white-box part of OpenChannel SSD is the exposure of its internal channels and chips to be managed by software (Linux LightNVM), but the OpenChannel firmware logic itself is a black-box part.

2.5 The State of Existing Emulators: We are only aware of three popular software-based emulators: FlashEm, LightNVM's QEMU and VSSIM.

FlashEm [47] is an emulator built in the Linux block level layer, hence less portable; it is rigidly tied to its Linux version; to make changes, one must modify Linux kernel. FlashEm is not open-sourced and its development stopped two years ago (confirmed by the creators).

LightNVM's QEMU platform [6] is still in its early stage. Currently, it cannot emulate multiple channels (as in OpenChannel SSD) and is only used for basic testing of 1 target (1 chip behind 1 channel). Worse, LightNVM's QEMU performance is not scalable to emulate NAND latencies as it depends on vanilla QEMU NVMe interface (as shown in the NVMe line in Figure 2a).

VSSIM [45] is a QEMU/KVM-based platform that emulates NAND flash latencies on a RAM disk, and has been used in several papers. The major drawback of VSSIM is that it is built within QEMU's IDE interface implementation, which is not scalable. The upper-left red line (IDE line) in Figure 2a shows the user-perceived IO read latency through VSSIM without any NAND-delay emulation added. More concurrent IO threads (x-axis) easily multiply the average IO latency (y-axis). For example from 1 to 4 IO threads, the average latency spikes up from 152 to 583µs. The root cause is that IDE is not supported with virtualization optimizations.

With this drawback, emulating internal SSD parallelism is a challenge. VSSIM worked around the problem by only emulating NAND delays in another background thread in QEMU, disconnected from the main IO path. Thus, for multi-threaded applications, to collect accurate results, users solely depend on VSSIM's monitoring tool [45, Figure 3], which monitors the IO latencies emulated in the background thread. In other words, users

cannot simply time the multi-threaded applications (due to IDE poor scalability) at the user level.

Despite these limitations, we (and the community) are *greatly indebted* to VSSIM authors as VSSIM provides a base design for future QEMU-based SSD emulators. As five years have passed, it is time to build a new emulator to keep up with the technology trends.

3 FEMU

We now present FEMU design and implementation. FEMU is implemented in QEMU v2.9 in 3929 LOC and acts as a virtual block device to the Guest OS. A typical software/hardware stack for SSD research is {Application+Host OS+SSD device}. With FEMU, the stack is {Application+Guest OS+FEMU}. The LOC above excludes base OC extension structures from LightNVM's QEMU and FTL framework from VSSIM.

Due to space constraints, we omit the details of how FEMU works inside QEMU (*e.g.*, FEMU's FTL and GC management, IO queues), as they are similarly described in VSSIM paper [45, Section 3]. We put them in FEMU release document [1]. In the rest of the paper, we focus on the main challenges of designing FEMU: achieving scalability (§3.1) and accuracy (§3.2) and increasing usability and extensibility (§3.3).

Note that all latencies reported here are user-perceived (application-level) latencies on memory-backed virtual storage and 24 dual-thread (2x) CPU cores running at 2.3GHz. According to our experiments, the average latency is inversely proportional to CPU frequency, for example, QEMU NVMe latency under 1 IO thread is 35µs on a 2.3GHZ CPU and 23µs on a 4.0GHz CPU.

3.1 Scalability

Scalability is an important property of a flash emulator, especially with high internal parallelism of modern SSDs. Unfortunately, stock QEMU exhibits a scalability limitation. For example, as shown in Figure 2a, with QEMU NVMe (although it is more scalable than IDE), more IO threads still increases the average IO latency (*e.g.*, with 8 IO threads, the average IO latency already reaches 106µs). This is highly undesirable because typical read latency of modern SSDs can be below 100µs.

More scalable alternatives to NVMe are virtio and dataplane (dp) interfaces [3, 30] (`virtio`/`dp` vs. `NVMe` lines in Figure 2a). However, these interfaces are not as extensible as NVMe (which is more popular). Nevertheless, virtio and dp are also not scalable enough to emulate low flash latencies. For example, at 32 IO threads, their IO latencies already reach 185µs and 126µs, respectively.

Problems: Collectively, all of the scalability bottlenecks above are due to two reasons: **(1)** QEMU uses a traditional trap-and-emulate method to emulate IOs. The

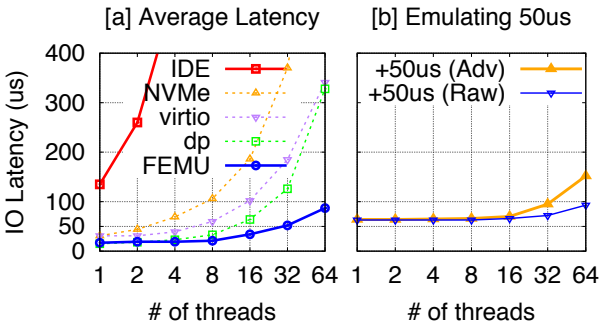


Figure 2: **QEMU Scalability.** *The figure shows the scalability of QEMU's IDE, NVMe, virtio, and dataplane (dp) interface implementations, as well as FEMU. The x-axis represents the number of concurrent IO threads running at the user level. Each thread performs random 4KB read IOs. The y-axis shows the user-perceived average IO latency. For Figure (a), the IDE and NVMe lines representing VSSIM and LightNVM's QEMU respectively are discussed in §2.5; virtio, dp, and FEMU lines in §3.1. For Figure (b), the "+50µs (Raw)" line is discussed in §3.2.1; the "+50µs (Adv)" line in "Result 3" part of §3.2.3.*

Guest OS' NVMe driver "rings the doorbell [5]" to the device (QEMU in our case) that some IOs are in the device queue. This "doorbell" is an MMIO operation that will cause an expensive VM-exit ("world switch" [39]) from the Guest OS to QEMU. A similar operation must also be done upon IO completion. **(2)** QEMU uses asynchronous IOs (AIO) to perform the actual read/write (byte transfer) to the backing image file. This AIO component is needed to avoid QEMU being blocked by slow IOs (*e.g.*, on a disk image). However, the AIO overhead becomes significant when the storage backend is a RAM-backed image.

Our solutions: To address these problems, we leverage the fact that FEMU purpose is for research prototyping, thus we perform the following modifications:

(1) We transform QEMU from an interrupt- to a polling-based design and disable the doorbell writes in the Guest OS (just 1 LOC commented out in the Linux NVMe driver). We create a dedicated thread in QEMU to continuously poll the status of the device queue (a shared memory mapped between the Guest OS and QEMU). This way, the Guest OS still "passes" control to QEMU but without the expensive VM exits. We emphasize that FEMU can still work without the changes in the Guest OS as we report later. This optimization can be treated as an optional feature, but the 1 LOC modification is extremely simple to make in many different kernels.

(2) We do not use virtual image file (in order to skip the AIO subcomponent). Rather, we create our own RAM-backed storage in QEMU's heap space (with configurable size `malloc()`). We then modify QEMU's DMA emulation logic to transfer data from/to our heap-

backed storage, transparent to the Guest OS (*i.e.*, the Guest OS is not aware of this change).

Results: The bold `FEMU` line in Figure 2a shows the scalability achieved. In between 1-32 IO threads, FEMU can keep IO latency stable in less than 52µs, and even below 90µs at 64 IO threads. If the single-line Guest-OS optimization is not applied (the removal of VM-exit), the average latency is 189µs and 264µs for 32 and 64 threads, respectively (not shown in the graph). Thus, we recommend applying the single-line change in the Guest OS to remove expensive VM exits.

The remaining scalability bottleneck now only comes from QEMU's *single-thread* "event loop" [4, 15], which performs the main IO routine such as dequeueing the device queue, triggering DMA emulations, and sending end-IO completions to the Guest OS. Recent works addressed these limitations (with major changes) [10, 23], but have not been streamlined into QEMU's main distribution. We will explore the possibility of integrating other solutions in future development of FEMU.

3.2 Accuracy

We now discuss the accuracy challenges. We first describe our delay mechanism (§3.2.1), followed by our basic and advanced delay models (§3.2.2-3.2.3).

3.2.1 Delay Emulation

When an IO arrives, FEMU will issue the DMA read/write command, then label the IO with an emulated completion time (T_{endio}) and add the IO to our "end-io queue," sorted based on IO completion time. FEMU dedicates an "end-io thread" that continuously takes an IO from the head of the queue and sends an end-io interrupt to the Guest OS, once the IO's emulated completion time has passed current time (T_{endio}>T_{now}).

The "+50us (Raw)" line in Figure 2b shows a simple (and stable) result where we add a delay of 50µs to *every* IO (T_{endio}=T_{entry}+50µs). Note that the end-to-end IO time is more than 50µs because of the Guest OS overhead (roughly 20µs). Important to say that FEMU also does not introduce severe latency tail. In the experiment above, 99% of all the IOs are stable at 70µs. Only 0.01% (99.99^{th} percentile) of the IOs exhibit latency tail of more than 105µs, which already exists in stock QEMU. For example, in VSSIM, the 99^{th}-percentile latency is already over 150µs.

3.2.2 Basic Delay Model

The challenge now is to compute the end-io time (T_{endio}) for every IO accurately. We begin with a basic delay model by marking every plane and channel with their next free time (T_{free}). For example, if a page write arrives to currently-free channel #1 and plane #2, then we will advance the channel's next free time

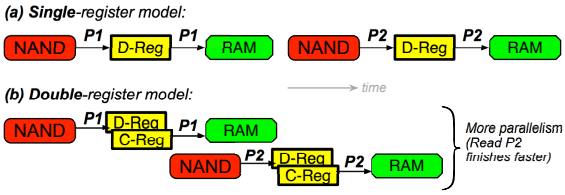


Figure 3: **Single- vs. double-register model.** *(a) In a single-register model, a plane only has one data register (D-Reg). Read of page P2 cannot start until P1 finishes using the register (i.e., the transfer to the controller's RAM completes). (b) In a double-register model, after P1 is read to the data register, it is copied quickly to the cache register (D-Reg to C-Reg). As the data register is free, read of P2 can begin (in parallel with P1's transfer to the RAM), hence finishes faster.*

($T_{freeOfChannel1}$=T_{now}+$T_{transfer}$, where $T_{transfer}$ is a configurable page transfer time over a channel) and the plane's next free time ($T_{freeOfPlane2}$+=T_{write}, where T_{write} is a configurable write/programming time of a NAND page). Thus, the end-io time of this write operation will be T_{endio}=$T_{freeOfPlane2}$.

Now, let us say a page read to the same plane arrives while the write is ongoing. Here, we will advance $T_{freeOfPlane2}$ by T_{read}, where T_{read} is a configurable read time of a NAND page, and $T_{freeOfChannel1}$ by $T_{transfer}$. This read's end-io time will be T_{endio}=$T_{freeOfChannel1}$ (as this is a read operation, not a write IO).

In summary, this basic *queueing* model represents a *single-register* and *uniform page latency* model. That is, every plane only has a single page register, hence cannot serve multiple IOs in parallel (*i.e.*, a plane's T_{free} represents IO serialization in that plane) and the NAND page read, write, and transfer times (T_{read}, T_{write} and $T_{transfer}$) are all *single* values. We also note that GC logic can be easily added to this basic model; a GC is essentially a series of reads/writes (and erases, T_{erase}) that will also advance plane's and channel's T_{free}.

3.2.3 Advanced "OC" Delay Model

While the model above is sufficient for basic comparative research (*e.g.*, comparing different FTL/GC schemes, some researchers might want to emulate the detailed intricacies of modern hardware. Below, we show how we extend our model and achieve a more accurate delay emulation of OpenChannel SSD ("**OC**" for short).

The OC's NAND hardware has the following intricacies. First, OC uses *double-register* planes; every plane is built with two registers (data+cache registers), hence a NAND page read/write in a plane can overlap with a data transfer via the channel to the plane (*i.e.*, more parallelism). Figure 3 contrasts the single- vs. double-register models where the completion time of the second IO to page P2 is faster in the double-register model.

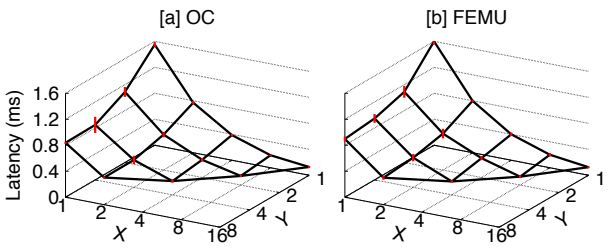


Figure 4: **OpenChannel SSD (OC) vs. FEMU.** *X: # of channels, Y: # of planes per channel. The figures are described in the "Result 1" segment of Section 3.2.3.*

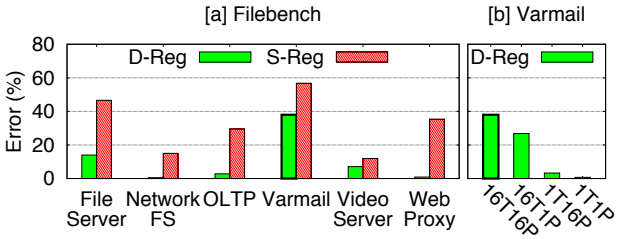


Figure 5: **Filebench on OpenChannel SSD (OC) vs. FEMU.** *The figures are described in the "Result 2" segment of Section 3.2.3. The y-axis shows the latency difference (error) of the benchmark results on OC vs. FEMU ($Error=(Lat_{femu}-Lat_{oc})/Lat_{oc}$). D-Reg and S-Reg represent the advanced and basic model respectively. The two bars with bold edge in Figures (a) and (b) are the same experiment and configuration (varmail with 16 threads on 16 planes).*

Second, OC uses a *non-uniform* page latency model; that is, pages that are mapped to upper bits of MLC cells ("upper" pages) incur higher latencies than those mapped to lower bits ("Lower" pages); for example 48/64µs for lower/upper-page read and 900/2400µs for lower/upper-page write. Making it more complex, the 512 pages in each NAND block are not mapped in a uniformly interleaving manner as in "`LuLuLuLu...`", but rather in a specific way, "`LLLLLLuLLuLLuu...`", where pages #0-6 and #8-9 are mapped to Lower pages, pages #7 and #10 to upper pages, and the rest ("...") have a repeating pattern of "`LLuu`".

Results: By incorporating this detailed model, FEMU can act as an accurate drop-in replacement of OC, which we demonstrate with the following results.

Result 1: Figure 4 compares the IO latencies on OC vs. FEMU. The workload is 16 IO threads performing random reads uniformly spread throughout the storage space. We map the storage space to different configurations. For example, x=1 and y=1 implies that OC and FEMU are configured with only 1 channel and 1 plane/channel, thus as a result, the average latency is high (z>1550µs) as all the 16 concurrent reads are contending for the same plane and channel. The result for x=16 and y=1 implies that we use 16 channels with 1 plane/channel (a total of 16 planes). Here, the concurrent reads are absorbed in parallel by all the planes and channels, hence a faster average read latency (z<130µs). Overall, Figures 4a and 4b exhibit a highly similar pattern, showing the success of our queuing delay emulation. The latency difference (error) is only between 0.8-11.6%; $Error=(Lat_{femu}-Lat_{oc})/Lat_{oc}$.

Result 2: Figure 5a shows the results from running several macrobenchmarks with six filebench personalities, with 16 IO threads of concurrent reads/writes on 16 planes across 4 channels. The figure only shows the latency difference ($Error$) which contrasts the accuracy of our basic and advanced delay models. With the basic model, the resulting latencies are highly inaccurate (12-57%), but with the advanced model, the error drops to only 0.5-38%, which are 1.5-40× more accurate across the six benchmarks.

We believe that these errors are reasonable as we deal with delay emulation of tens of µs granularity. We leave further optimization for future work; we might have missed other OC intricacies that should be incorporated into our advanced model (as explained at the end of §2.4, OC only exposes channels and chips, but other details are not exposed by the vendor). Nevertheless, we investigate further the residual errors, as shown in Figure 5b. Here, we use the `varmail` personality but we vary the #IO threads [T] and #planes [P]. For example, in the 16 threads on 16 planes configuration (x="16T16P" in Figure 5b, which is the same configuration used in experiments in Figure 5a), the error is 38%. However, the error decreases in less complex configurations (*e.g.*, 0.7% error with single thread on single plane). Thus, higher errors come from more complex configurations (*e.g.*, more IO threads and more planes), which we explain next.

Result 3: We find that using an advanced model requires more CPU computation, and this compute overhead will backlog with higher thread count. To show this, Figure 2b compares the simple +50µs delay emulation in our raw implementation (§3.2.1) vs. advanced model. Here, both cases simply add +50µs, but the advanced model must traverse many `if-else` statements (to check register, plane, and channel next free time), hence the compute overhead. Further scalability optimizations, as discussed at the end of §3.1 can help.

3.3 Usability and Extensibility

Being a *software*-based emulation platform, FEMU can be extended in many different ways. We now describe existing features/usabilities of FEMU, briefly showcase successful extensions used in our recent work [14, 43] as well as possible future work that FEMU features enable.

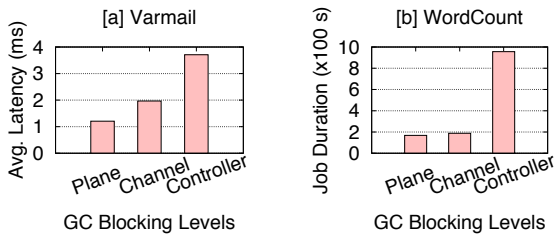


Figure 6: **Use examples.** *Figure 6a is described in the "FTL and GC schemes" segment of Section 3.3. Figure 6b is discussed in the "Distributed SSDs" segment of Section 3.3.*

• **FTL and GC schemes:** In default mode, our FTL employs a dynamic mapping and a channel-blocking GC as used in other simulators [9, 16]. One of our projects uses FEMU to compare different GC schemes: controller, channel, and plane blocking [43]. In controller-blocking GC, a GC operation "locks down" the controller, preventing any foreground IOs to be served (as in OpenSSD [7]). In channel-blocking GC, only channels involved in GC page movement are blocked (as in SSDSim [16]). In plane-blocking GC, the most efficient one, page movement only flows within a plane without using any channel (*i.e.*, "copyback" [2]). Sample results are shown in Figure 6a. Beyond our work, recent works also show the benefits of SSD partitioning for performance isolation [11, 17, 22, 27, 37], which are done on either a simulator or a hardware platform. More partitioning schemes can also be explored with FEMU.

• **White-box vs. black-box mode:** FEMU can be used as **(1)** a white-box device such as OpenChannel SSD where the device exposes physical page addresses and the FTL is managed by the OS such as in Linux LightNVM or **(2)** a black-box device such as commodity SSDs where the FTL resides inside FEMU and only logical addresses are exposed to the OS.

• **Multi-device support for flash-array research:** FEMU is configurable to appear as multiple devices to the Guest OS. For example, if FEMU exposes 4 SSDs, inside FEMU there will be 4 separate NVMe instances and FTL structures (with no overlapping channels) managed in a single QEMU instance. Previous emulators (VSSIM and LightNVM's QEMU) do not support this.

• **Extensible OS-SSD NVMe commands:** As FEMU supports NVMe, new OS-to-SSD commands can be added (*e.g.*, for host-aware SSD management or split-level architecture [31]). For example, currently in LightNVM, a GC operation reads valid pages from OC to the host DRAM and then writes them back to OC. This wastes host-SSD PCIe bandwidth; LightNVM foreground throughput drops by 50% under a GC. Our conversation with LightNVM developers suggests that one can add a new "`pageMove fromAddr toAddr`" NVMe command from the OS to FEMU/OC such that the data movement does not cross the PCIe interface. As mentioned earlier, split-level architecture is trending [12, 20, 29, 40, 44] and our NVMe-powered FEMU can be extended to support more commands such as transactions, deduplication, and multi-stream.

• **Page-level latency variability:** As discussed before (§3.2), FEMU supports page-level latency variability. Among SSD engineers, it is known that "not all chips are equal." High quality chips are mixed with lesser quality chips as long as the overall quality passes the standard. Bad chips can induce more error rates that require longer, repeated reads with different voltages. FEMU can also be extended to emulate such delays.

• **Distributed SSDs:** Multiple instances of FEMU can be easily deployed across multiple machines (as simple as running Linux hypervisor KVMs), which promotes more large-scale SSD research. For example, we are also able to evaluate the performance of Hadoop's word-count workload on a cluster of machines running FEMU, but with different GC schemes as shown in Figure 6b. Since HDFS uses large IOs, which will eventually be striped across many channels/planes, there is a smaller performance gap between channel and plane blocking. We hope FEMU can spur more work that modifies the SSD layer to speed up distributed computing frameworks (*e.g.*, distributed graph processing frameworks).

• **Page-level fault injection:** Beyond performance-related research, flash reliability research [26, 33] can leverage FEMU as well (*e.g.*, by injecting page-level corruptions and faults and observing how the high-level software stack reacts).

• **Limitations:** FEMU is DRAM-backed, hence cannot emulate large-capacity SSDs. Furthermore, for crash consistency research, FEMU users must manually emulate "soft" crashes as hard reboots will wipe out the data in the DRAM. Also, as mentioned before (§3.2), there is room for improving accuracy.

4 Conclusion & Acknowledgments

As modern SSD internals are becoming more complex, their implications to the entire storage stack should be investigated. In this context, we believe FEMU is a fitting research platform. We hope that our cheap and extensible FEMU can speed up future SSD research.

We thank Sam H. Noh, our shepherd, and the anonymous reviewers for their tremendous feedback. This material was supported by funding from NSF (grant Nos. CNS-1526304 and CNS-1405959).

References

[1] https://github.com/ucare-uchicago/femu.

[2] Using COPYBACK Operations to Maintain Data Integrity in NAND Flash Devices. https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2941_idm_copyback.pdf, 2008.

[3] Towards Multi-threaded Device Emulation in QEMU. KVM Forum, 2014.

[4] Improving the QEMU Event Loop. KVM Forum, 2015.

[5] NVMe Specification 1.3. http://www.nvmexpress.org, 2017.

[6] Open-Channel Solid State Drives. http://lightnvm.io, 2017.

[7] The OpenSSD Project. http://openssd.io, 2017.

[8] Violin Memory. All Flash Array Architecture, 2017.

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

[10] Muli Ben-Yehuda, Michael Factor, Eran Rom, Avishay Traeger, Eran Borovik, and Ben-Ami Yassour. Adding Advanced Storage Controller Functionality via Low-Overhead Virtualization. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

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

[12] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. Software Orchestrated Flash Array. In *The 7th Annual International Systems and Storage Conference (SYSTOR)*, 2014.

[13] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[14] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[15] Nadav Har'El, Nadav, Gordon, Abel, Landau, Alex, Ben-Yehuda, Muli, Traeger, Avishay, Ladelsky, and Razya. Efficient and Scalable Paravirtual I/O System. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

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

[17] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[18] Xavier Jimenez and David Novo. Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST)*, 2014.

[19] William K. Josephson, Lars A. Bongo, David Flynn, Fusion-io, and Kai Li. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[20] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed Solid-State Drive. In *the 6th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[21] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

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

[23] Tae Yong Kim, Dong Hyun Kang, Dongwoo Lee, and Young Ik Eom. Improving Performance by Bridging the Semantic Gap between Multi-queue SSD and I/O Virtualization Framework. In *Proceedings of the 31st IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2015.

[24] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[25] Sungjin Lee, Ming Liu, Sang Woo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed Flash. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[26] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2015.

[27] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

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

[29] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[30] Rusty Russell. virtio: Towards a De-Facto Standard for Virtual I/O Devices. In *ACM SIGOPS Operating Systems Review (OSR)*, 2008.

[31] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 2012 EuroSys Conference (EuroSys)*, 2012.

[32] Mohit Saxena, Yiying Zhang, Michael M. Swift, Andrea C. Arpaci Dusseau, and Remzi H. Arpaci Dusseau. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

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

[34] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[35] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: A Deep Integration of Device and Application for Flash Based Key-Value Caching. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[36] Liang Shi, Kaijie Wu, Mengying Zhao, Chun Jason Xue, Duo Liu, and Edwin H.-M. Sha. Retention Trimming for Lifetime Improvement of Flash Memory Storage Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35(1), January 2016.

[37] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[38] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a Flash with ioSnap. In *Proceedings of the 2014 EuroSys Conference (EuroSys)*, 2014.

[39] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2001.

[40] Animesh Trivedi, Nikolas Ioannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Ioannis Koltsidas, Kornilios Kourtis, and Thomas R. Gross. FlashNet: Flash/Network Stack Co-design. In *The 10th Annual International Systems and Storage Conference (SYSTOR)*, 2017.

[41] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the 2014 EuroSys Conference (EuroSys)*, 2014.

[42] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[43] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[44] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. AutoStream: Automatic Stream Management for Multi-streamed SSDs. In *The 10th Annual International Systems and Storage Conference (SYSTOR)*, 2017.

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

[46] Jianquan Zhang, Dan Feng, Jianlin Gao, Wei Tong, Jingning Liu, Yu Hua, Yang Gao, Caihua Fang, Wen Xia, Feiling Fu, and Yaqing Li. Application-Aware and Software-Defined SSD Scheme for Tencent Large-Scale Storage System. In *Proceedings of 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[47] 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 Symposium on File and Storage Technologies (FAST)*, 2012.

Spiffy: Enabling File-System Aware Storage Applications

Kuei Sun, Daniel Fryer, Joseph Chu, Matthew Lakier, Angela Demke Brown and Ashvin Goel
University of Toronto

Abstract

Many file-system applications such as defragmentation tools, file system checkers or data recovery tools, operate at the storage layer. Today, developers of these storage applications require detailed knowledge of the file-system format, which takes a significant amount of time to learn, often by trial and error, due to insufficient documentation or specification of the format. Furthermore, these applications perform ad-hoc processing of the file-system metadata, leading to bugs and vulnerabilities.

We propose Spiffy, an annotation language for specifying the on-disk format of a file system. File-system developers annotate the data structures of a file system, and we use these annotations to generate a library that allows identifying, parsing and traversing file-system metadata, providing support for both offline and online storage applications. This approach simplifies the development of storage applications that work across different file systems because it reduces the amount of file-system specific code that needs to be written.

We have written annotations for the Linux Ext4, Btrfs and F2FS file systems, and developed several applications for these file systems, including a type-specific metadata corruptor, a file system converter, and an online storage layer cache that preferentially caches files for certain users. Our experiments show that applications that use the library to access file system metadata can achieve good performance and are robust against file system corruption errors.

1 Introduction

There are many file-system aware storage applications that bypass the virtual file system interface and operate directly on the file system image. These applications require a detailed understanding of the format of a file system, including the ability to identify, parse and traverse file system structures. These applications can operate in an offline or online context, as shown in Table 1. Examples of offline tools include a file system checker that traverses the file system image to check the consistency of its metadata [17], and a data recovery tool that helps recover deleted files [4].

Online storage applications need to understand the file-system semantics of blocks as they are accessed at runtime (e.g., whether the block contains data or metadata, whether it belongs to a specific type of file, etc.).

Storage Applications	Category	Purpose
Differentiated services [18]	online	performance
Defragmentation tool	either	
File system checker [13]	either	reliability
Data recovery tool [4]	offline	
IO shepherding [12]	online	
Runtime verification [8]	online	
File system conversion tool	offline	administrative
Partition editor [11]	offline	
Type-specific corruption [2]	offline	debugging
Metadata dump tool	offline	

Table 1: Example file-system aware storage applications. Offline applications have exclusive access to the file system; online applications operate on an in-use file system.

These applications improve the performance or reliability of a storage system by performing file-system specific processing at the storage layer. For example, differentiated storage services [18] improve performance by preferentially caching blocks that contain file-system metadata or the data of small files. I/O shepherding [12] improves reliability by using file structure information to implement checksumming and replication. Similarly, Recon [8] improves reliability by verifying the consistency of file-system metadata at the storage layer.

Today, developers of these storage applications perform ad-hoc processing of file system metadata because most file systems do not provide the requisite library code. Even when such library code exists, its interface may not be usable by all storage applications. For example, the `libext2fs` library only supports offline interpretation of a Linux Ext3/4 file system partition; it does not support online use. Furthermore, the libraries of different file systems, even when they exist, do not provide similar interfaces. As a result, these storage applications have to be developed from scratch, or significantly rewritten for each file system, impeding the adoption of new file systems or new file-system functionality.

To make matters worse, many file systems do not provide detailed and up-to-date documentation of their metadata format. The ad-hoc processing performed by these storage applications is thus error-prone and can lead to system instability, security vulnerability, and data corruption [3]. For example, `fsck` can sometimes further corrupt a file system [33]. Some storage applications reduce the amount of file-system specific code in their im-

plementation by modifying their target file system and operating system [18, 12]. This approach only works for specific file systems, and can introduce its own bugs. It also requires custom system software, which may be impractical in virtual machine and cloud environments.

Our aim is to reduce the burden of developing file-system aware storage applications. To do so, we enable file system developers to specify the format of their file system using a domain-specific language so that the file system metadata can be parsed, traversed and updated correctly. We introduce Spiffy,[1] a language for annotating file system data structures defined in the C language. Spiffy allows file system developers to unambiguously specify the *physical* layout of the file system. The annotations handle low level details such as the encoding of specific fields, and the pointer relationships between file system structures. We compile the annotated sources to generate a Spiffy library that provides interfaces for type-safe parsing, traversal and update of file system metadata. The library allows a developer to write actions for different file system metadata structures, invoking file-system specific or generic code as needed, for their offline or online application. We support online applications that need to read metadata, such as differentiated storage services [18], but not ones that need to modify metadata such as online defragmentation.

The generic interfaces provided by the library simplify the development of applications that work across different file systems. Consider an application that shows file-system fragmentation by plotting a histogram of the size of free extents in the file system. This application needs to traverse the file system to find and parse structures that represent free space, and then collect the extent information. With Spiffy, the application code for finding and parsing structures is similar for different file systems. File-system specific actions are only needed for collecting the extent information from the free space structures (e.g., bitmaps for Ext4 and free space extents for Btrfs).

The complexity of modern file systems [16] raises several challenges for our specification-based approach. Many aspects of file system structures and their relationships are not captured by their declarations in header files. First, an on-disk pointer in a file-system structure may be implicitly specified, e.g., as an integer, as shown below. The naming convention suggests that this field is a pointer, but that fact cannot be deduced from the structure definition because it is embedded in file system code.

```
struct foo {
  __le32 bar_block_ptr;
};
```

Second, the interpretation of file system structures can depend on other structures. For example, the size of an inode structure in a Linux Ext3/4 file system is stored in a field within the super block that must be accessed to correctly interpret an inode block. Similarly, many structures are variable sized, with the size information being stored in other structures. Third, the semantics of metadata fields may be context-sensitive. For example, pointers inside an inode structure can refer to either directory blocks or data blocks, depending on the type of the inode. Fourth, the placement of structures on disk may be implicit in the code that operates on them (e.g., an instance of structure B optionally follows structure A) and some structures may not be declared at all (e.g., treating a buffer as an array of integers). Finally, metadata interpretation must be performed efficiently, but it is impractical to load all file-system metadata into memory for large file systems. These challenges are not addressed by existing specification tools, as discussed in Section 7.

In Spiffy, the key to specifying the relationships between file system structures is a pointer annotation that specifies that a field holds an address to a data structure on physical storage. Pointers have an address space type that indicates how the address should be mapped to the physical location. In the `struct foo` example above, this annotation would help clarify that `bar_block_ptr` holds an address to a structure of type `bar`, and its address space type is a (little-endian) block pointer. We expose cross-structure dependencies by using a name resolution mechanism that allows annotations to name the necessary structures unambiguously. We handle context-sensitive fields and structures by providing support for conditional types and conditionally inherited structures. We also provide support for specifying implicit fields that are computed at runtime. Last, annotations can specify the granularity at which the structures should be accessed from storage, allowing efficient data access and reducing the memory footprint of the applications.

Together, these Spiffy features have allowed us to properly annotate three widely deployed file systems, 1) Ext4, an update-in-place file system, 2) Btrfs, a copy-on-write file system, and 3) F2FS, a log-structured file system [15]. We have implemented five applications that are designed to work across file systems: a file system dump tool, a file system corruption tool, a free space display tool, a file system converter, and a storage layer service that preferentially caches data for specific users.

2 Bugs in File-System Applications

We motivate this work by presenting various bugs caused by incorrect parsing of file-system metadata in storage applications (outlined in Table 2). Some of these bugs cause crashes, while others may result in file system corruption. For each bug, we discuss the root cause.

1. An extra memory allocation caused uninitialized bytes

[1] **Sp**ecifying and **I**nterpreting the **F**ormat of **F**iles**y**stems

	Tool	FS	Bug Title	Closed
1	libparted	Fat32	#22266: jump instruction and boot code corrupted with random bytes after fat is resized	2016-05
2	ntfsprogs	NTFS	Bug 723343 - Negative Number of Free Clusters in NTFS Not Properly Interpreted	2014-02
3	e2fsck	Ext4	#781110 e2fsprogs: e2fsck does not detect corruption	2016-05
4	e2fsck	Ext4	#760275 e2fsprogs: e2fsck corrupts Hurd filesystems	2015-05
5	btrfsck	Btrfs	Bug 104141 - Malformed input causing crash / floating point exception in btrfsck	2015-10

Table 2: Bugs due to incorrect parsing of file system formats.

to be written to the boot jump field of Fat32 file systems during resizing. Since Windows depends on the correctness of this field, the bug rendered the file system unrecognizable by the operating system.

2. NTFS has a complex specification for the size of the MFT record. If the value is positive, it is interpreted as the number of clusters per record. Otherwise, the size of the record is $2^{|value|}$ bytes (e.g., -10 would mean that the record size is 1024 bytes). The developers of ntfsprogs were unaware of this detail, and so the GParted partition editing tool would fail when attempting to resize an NTFS partition.
3. The e2fsck file system checker failed to detect corrupted directory entries if the size field of the entries was set to zero, which resulted in no repair being performed. Ironically, other programs, such as debugfs, ls, and the file system itself, could correctly detect the corruption.
4. Ext2/3/4 inodes contain union fields for storing operating system (OS) specific metadata. A sanity check was omitted in e2fsck prior to accessing this field, and repairs were always performed assuming that the creator OS is Linux. Consequently, the file system becomes corrupt for Hurd and possibly other OSs.
5. A fuzzer [34] was able to craft corrupted super blocks that would crash the Btrfsck tool. In response, Btrfs developers added 15 extra checks (for a total of 17 checks) to the super block parsing code.

The common theme among all these bugs is that: 1) they are simple errors that occur because they require a detailed understanding of the file system format; 2) they can cause serious data loss or corruption; and 3) most of these bugs were fixed in less than 5 lines of code. Our domain-specific language allows generating libraries that can sanitize file system metadata by checking various structural constraints before it is accessed in memory. In the presence of corrupted metadata, our libraries generate error codes, rather than crashing the tools or propagating the corruption further. Section 3.1 discusses how our approach can help prevent or detect these bugs.

3 Approach

Our annotation language enables type-safe interpretation of file system structures, in both offline and online contexts. Type safety ensures that parsing and serialization of file system structures will detect data corruption that leads to type violations, thus reducing the chance of corruption propagation, and avoiding crash failures.

Ideally, data structure types and their relationships could be extracted from file system source code. Although the C header files of a file system contain the structural definitions for various metadata types, they are incomplete descriptions of the file system format because information is often hidden within the file system code. Our annotations augment the C language, helping specify parts of a file system's format that cannot be easily expressed in C.

After a file system developer annotates his or her file system's data structures, we use a compiler to parse the annotated structures and to generate a library that provides file-system specific interpretation routines. The library supports traversal and selective retrieval of metadata structures through type introspection. These facilities allow writing generic or file-system specific actions on specific file system metadata structures. For example, the application may wish to operate on the directory entries of a file system. Instead of attempting to parse the entire file system and find all directory entries, which requires significant file-system specific code, a developer using Spiffy would use generic type introspection code to find and operate on all directory entries. However, since the directory entry format may not be the same across file systems, the application may require file-system specific actions on the directory entry structures.

Our annotation-based approach has several advantages. First, it provides a concise and clear documentation of the file system's format. Second, our generated libraries enable rapid prototyping of file-system aware storage applications. The libraries provide a uniform API, easing the development of applications that work across file systems so that the programmer can focus on the logic and not the format of the file systems. Third, our approach requires minimal changes to the file system source code (the annotations are only in the C header files and are backwards compatible with existing binary code), reducing the chance of introducing file system bugs. In contrast, differentiated storage services [18] needed to modify the file system and the kernel's storage stack to enable I/O classification. With our approach, this application can be implemented by using introspec-

```
struct ext4_dir_entry {
  __le32 inode;             /* Inode number */
  __le16 rec_len; /* Directory entry length */
  __u16  name_len;           /* Name length */
  char   name[EXT4_NAME_LEN];  /* File name */
};
```

Figure 1: Ext4 directory entry structure definition.

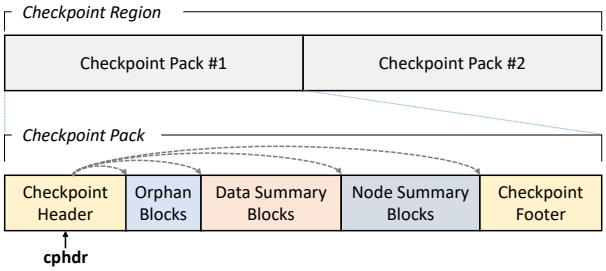


Figure 2: Each F2FS checkpoint pack contains a header followed by a variable number of orphan blocks.

tion at the block layer for an unmodified file system, or at the hypervisor for an existing virtual machine. Finally, file system formats are known to be stable over time, so there is minimal cost for maintaining annotations.

3.1 Designing Annotations

The design of our annotation language for specifying the format of file system structures was motivated by several key concepts.

File System Pointers File system pointers connect the metadata structures in a file system, but they are not well specified in C data structure definitions, as explained in Section 1. The difference between a file system pointer and an in-memory pointer is that the content of an in-memory pointer is always interpreted as the in-memory address of the pointed-to data, but interpreting the address contained by a file system pointer may involve multiple layers of translation. The most common type of file system pointer is a block pointer, where the address maps to a physical block location that contains a contiguous data structure. However, file system structures may also be laid out discontiguously. For example, the journal of an Ext4 file system is a logically contiguous structure that can be stored on disk non-contiguously, as a file. Similarly, Btrfs maps logical addresses to physical addresses for supporting RAID configurations.

Our design incorporates this requirement by associating an *address space* with each file system pointer. Each address space specifies a mapping of its addresses to physical locations. In the case of the Ext4 journal, we use the inode number, which uniquely identifies files in Unix file systems, as an address in the file address space.

Cross-Structure Dependencies File system structures often depend on other structures. For example, the length of a directory entry's `name` in Ext4 is stored in a field called `name_len`, as shown in Figure 1. However, this data structure definition does not provide the linkage between the two fields.[2] Structures may depend on fields in other structures as well. For example, several fields of the super block are frequently accessed to determine the block size, the features that are enabled in the file system, etc. To support these dependencies, we need to name these structures. For example, the expression `sb.s_inode_size` helps determine the size of an inode object, where `sb` is the name assigned to the super block.

The naming mechanism must ensure that a name refers to the correct structure. For example, the F2FS file system contains two checkpoint packs for ensuring file system consistency, as shown in Figure 2. The number of orphan blocks in a F2FS checkpoint pack is determined by a field inside the checkpoint header. Our naming mechanism must ensure that when this field is accessed, it refers to the header structure in the correct checkpoint pack.

Spiffy uses a path-based name resolution mechanism, based on the observation that every file system structure is accessed along a path of pointers starting from the super block. In the simplest case, the automatic `self` variable is used to reference the fields of the same structure. Otherwise, a name lookup is performed in the reverse order of the path that was used to access the data structure. For example, in Figure 2, when we need to reference the checkpoint header (`cphdr` in the figure) while parsing the orphan block, the name resolution mechanism can unambiguously determine that it is referring to its parent checkpoint header. This strategy also makes it easy to use reference counting to ensure that a referenced structure is valid in memory when it needs to be accessed.

Context-Sensitive Types File system metadata are frequently context-sensitive. A pointer may reference different types of metadata, or a structure may have optional fields, based on a field value. For example, the type of a journal block in Ext4 depends on a common field called `h_blocktype`. If the field's value is 3, then it is the journal super block that contains many additional fields that can be parsed. However, if its value is 2, then it is a commit block that contains no other fields. We need to be able to handle such context-sensitive structures and pointers. We use a *when* expression, evaluated at runtime, to support such context-sensitive types. These conditional expressions also allow us to specify when different fields of a union are valid, which enables Spiffy to enforce a strict access discipline at runtime, and would prevent Bug #4 from Section 2.

Computed Fields Sometimes file systems compute a value from one or more fields and use it to locate structures. For example, the block group descriptor table in

[2] Confusingly, `name` has a fixed size in the definition.

Base Class	Member Function	Description
Spiffy File System Library		
Entity	`int process_fields(Visitor & v)`	allows *v* to visit all fields of this object
	`int process_pointers(Visitor & v)`	allows *v* to visit all pointer fields of this object
	`int process_by_type(int t, Visitor & v)`	allows *v* to visit all structures of type *t*
Pointer	`Entity * fetch()`	retrieves the pointed-to container from disk
Container	`int save(bool alloc=true)`	serializes and then persists the container, may assign a new address to the container
FileSystem	`FileSystem(IO & io)`	instantiates a new file system object
	`Entity * fetch_super()`	retrieves the super block from disk
	`Entity * create_container(int type, Path & p)`	creates a new container of metadata *type*
	`Entity * parse_by_type(int type, Path & p, Address & addr, const char * buf, size_t len)`	parses the buffer as metadata *type*, using *p* to resolve cross structure dependencies
File System Developer		
IO	`int read(Address & addr, char * & buf)`	reads from an address space specified by *addr*
	`int write(Address & addr, const char * buf)`	writes to an address space specified by *addr*
	`int alloc(Address & addr, int type)`	allocates an on-disk address for metadata *type*
Application Programmer		
Visitor	`int visit(Entity * e)`	visits an entity and possibly processes it

Table 3: Spiffy C++ Library API.

Ext4 is implicitly the block(s) that immediately follows the super block. However, the exact address of the descriptor blocks depends on the block size, which is specified in the super block. We annotate this information as an implicit field of the super block that is computed at runtime. This approach allows the field to be dereferenced like a normal pointer, allowing traversal of the file system without requiring any changes to the underlying format. A computed field annotation can also be used to specify the size calculation for an NTFS MFT record, avoiding Bug #2 from Section 2.

Metadata Granularity Existing file systems assume that the underlying storage media is a block device and access data in block units. Data structures can exist within such blocks or they can span contiguous physical blocks. Some data structures that span blocks are read in their entirety. For example, the Btrfs B-tree nodes are (by default) 16KB, or 4 blocks, and these blocks are read from disk together. In other cases, the data structure is read in portions. For example, an Ext4 inode table contains a group of inode blocks. The file system does not load the entire table in memory because it can be very large. Instead, it only loads the portions that are needed.

We define an *access unit* for file system structures so that the compiler can generate efficient code for traversing the file system. We call the unit of disk access a *container*. The container size is typically the file system block size but it may span multiple blocks, as in the Btrfs example. A structure that is placed inside a container is called an *object*. Finally, structures that span containers are called *extents*. We load extents on demand, when their containers are accessed.

Constraint Checking The values of metadata fields within or across different objects often have constraints. For example, an Ext4 extent header always begins with the magic number `0xF30A` to help detect corrupt blocks. Similarly, the `name_len` field of an Ext4 directory entry should be less than the `rec_len` field. Such constraints can be specified for each structure so that they can be checked to ensure correctness when parsing the structure. The use of constraint annotations could have helped prevent Bug #1, and detect Bugs #3 and #5 from Section 2.

The set of valid addresses for a metadata container may also have a *placement constraint*. For example, F2FS NAT blocks can only be placed inside the NAT area, which is specified in the F2FS super block. By annotating the placement constraint of a metadata container, Spiffy can verify that the address assigned to newly allocated metadata is within the correct bounds before the metadata is persisted to disk.

3.2 The Spiffy API

Table 3 shows a subset of the API for building Spiffy applications. The API consists of three sets of functions. The first set are automatically generated by Spiffy based on the annotated file system data structures. The second set need to be implemented by file system developers and are reusable across different applications. The last set are written by the application programmer for implementing application and file-system specific logic.

The Spiffy library uses the visitor pattern [9], allowing a programmer to customize the operations performed on each file system metadata type by implementing the `visit` function of the abstract base class `Visitor`.

The `Entity` base class provides a common interface for all metadata structures and their fields. The `process_pointers` function invokes the `visit` function of an application-defined Visitor class on each

```
struct Address {
  int      aspc;   /* address space type */
  long     id;     /* id of the address */
  unsigned offset; /* offset from id */
  unsigned size;   /* size of object */
};
```

Figure 3: Address structure to locate container on disk.

pointer within the entity. The `process_by_type` function allows visiting a specific type of structure that is reachable from the entity. Unlike the other process functions, `process_by_type` will automatically follow pointers. For example, invoking `process_by_type` on the super block with the inode structure as an argument results in visiting all inodes in the file system.

Every container (and extent) has an address associated with it that allows accessing the container from disk. Figure 3 shows the format of an address, consisting of an address space, an identifier and an offset within the address space, and the size of the container. The offset field is used when a container belongs to an extent.

The `Pointer` class stores the address of a container (or an extent), and its `fetch` function reads the pointed-to container from disk. Figure 4 shows the generated code for the `fetch` function for a pointer to a container named `IBlock` (inode block). The file-system developer implements an `IO` class with a `read` function for each address space defined for the file system. When the `IBlock` is constructed, it invokes the constructors of its fields, thus creating all the objects (e.g., inodes) within the container. The constructors for inodes, in turn, invoke the constructors of block pointers in the inodes, which initialize a part of the address (address space, size and offset) of the block pointers based on the annotations. Then the container is parsed, which initializes the container fields in a nested manner, including setting the `id` component of the address of all the block pointers in the inodes contained in the `IBlock`.

The `Path` object is associated with every entity and contains the list of structures that are needed to resolve cross-structure dependencies during parsing or serializing the container. It is set up based on the sequence of constructor calls, with each constructor adding the current object to the path passed to it.

The `save` function serializes a container by invoking nested serialization on its fields. Then, it invokes the `alloc` function for newly created metadata, or when existing metadata has to be reallocated (e.g., copy-on-write allocator). The allocator finds a new address for the container and updates any metadata that tracks allocation (e.g., the Ext4 block bitmap). If the address passes placement constraint checks, the buffer is written to disk.

The `create_container` function constructs empty containers of a given type. The application developer can then fill the container with data and invoke `save` to allocate and write the newly created container to disk.

```
Entity * IBlockPtr::fetch() {
  IBlock * ib;
  Address & addr = this->address;
  char * buf = new char[addr.size];
  this->fs.io.read(addr, buf);
  ib = new IBlock(this->fs, addr, this->path);
  ib->parse(buf, addr.size);
  return ib;
}
```

Figure 4: Example of a generated `fetch` function. `IBlockPtr` is a subclass of `Pointer`.

3.3 Building Applications

Figure 5 shows a sample application built using the Spiffy API. This application prints the type of each metadata block in an Ext4 file system in depth-first order. The `Ext4IO` class implements the block and the file address space, as described in Section 5. The program starts by invoking `fetch_super`, which fetches the super block from a known location on disk and parses it. Then it uses two mutually recursive visitors, `EntVisitor` and `PtrVisitor`, to traverse the file system.

The `EntVisitor::visit` function takes an entity as input, prints its name, and then invokes `process_pointers`, which calls the `PtrVisitor::visit` function for every pointer in the entity. The `PtrVisitor::visit` function invokes `fetch`, which fetches the pointed-to entity from disk, and invokes `EntVisitor::visit` on it.

3.4 Limitations

The correctness of Spiffy applications depends on correctly written annotations. Therefore, if and when file system format changes do occur, the specifications will need to be updated. Spiffy applications will also need to update all file-system specific code that is affected by the format changes. These changes will likely only affect code that directly operates on the updated metadata structures, since the Spiffy library will provide safe traversal and parsing of any intermediate structures.

Currently, we have implemented an online application at the storage layer (metadata caching, see Section 5) that reads file system metadata, but does not modify it. We are exploring modifying file system metadata using Spiffy at the storage layer (which requires hooks into the file system code, e.g., for transactions and allocation [12]), and at the file system level (which enables more powerful applications).

Unlike typical file-system applications that operate at the VFS layer and are file-system independent, Spiffy applications operate directly on file-system specific struc-

```
EntVisitor ev;
PtrVisitor pv;
int PtrVisitor::visit(Entity & e) {
  Entity * tmp = ((Pointer &)e).fetch();
  if (tmp != nullptr) {
    ev.visit(*tmp);
    tmp->destroy();
  }
  return 0;
}
int EntVisitor::visit(Entity & e) {
  cout << e.get_name() << endl;
  return e.process_pointers(pv);
}
void main(void) {
  Ext4IO io("/dev/sdb1");
  Ext4 fs(io);
  Entity * sup;
  if ((sup = fs.fetch_super()) != nullptr) {
    ev.visit(*sup);
    sup->destroy();
  }
}
```

Figure 5: Code for traversing and printing the types of all the metadata blocks in an Ext4 file system.

tures and are thus file-system dependent. Since file systems share common abstractions (e.g. files, directories, inodes), it may be possible to carefully abstract the functionality that is shared between implementations, reducing file-system dependence even further.

4 File System Applications

We have written five file-system aware storage applications using the Spiffy framework: a dump tool, a free space reporting tool, a type-specific metadata corruptor, a file system conversion tool, and a prioritized block layer cache. The first four applications operate offline, while the last one is an online application.

File System Dump Tool The file system dump tool parses all the metadata in a file system image and exports the result in an XML format, using file system traversal code similar to the example in Figure 5. In addition to `process_pointers`, the entity class provides a `process_fields` method that allows iterating over all fields (not just pointer fields) of the class. The dump tool can be configured to prevent structures such as unallocated inode structures from being exported.

Type-Specific Corruption Tool This tool is a variant of the dump tool that injects file-system corruption in a type-specific manner [2], allowing us to test the robustness of file systems and their tools. When we decide to corrupt a field, we cannot simply modify its in-memory value, since serialization is type-safe. For example, the serializer will refuse to serialize a corrupted value that violates its type constraints. Instead, corruption is performed after a block is serialized but before it is written.

Free Space Tool This tool shows file-system fragmentation by plotting a histogram of the size of free extents. The tool retrieves the metadata structures that store free space information and processes them (e.g., block bitmaps for Ext4, extent items for Btrfs, and segment information table (SIT) for F2FS). This logic is implemented using `process_by_type` (see Table 3) and a custom visit function that processes all the retrieved metadata structures. Code to traverse the file system and parse intermediate structures is provided by our library.

File System Conversion Tool Converting an existing file system into a file system of another type is a time-consuming process, involving copying files to another disk, reformatting the disk, and then copying the files back to the new file system. In-place file system conversion that updates file system metadata without moving most file data can speed up the conversion dramatically. While some such conversion tools exist,[3] they are hard to implement correctly and not generally available.

We have designed an in-place file system conversion tool using the Spiffy framework. Such a conversion tool requires detailed knowledge of the source and the destination file systems, and is thus a challenging application for our approach. In-place conversion involves several steps. First, the file and directory related metadata, such as inodes, extent mappings, and directory entries of the source file system, are parsed into a standard format. Second, the free space in the source file system is tracked. Third, if any source file data occupies blocks that are statically allocated in the destination file system, then those blocks are reallocated to the free space, and the conversion aborted if sufficient free space is not available. Finally, the metadata for the destination file system is created and written to disk. In our current tool, a power failure during the last step would corrupt the source file system. We plan to add failure atomicity in the future.

Our tool currently converts extent-based Ext4 file systems to log-structured F2FS file systems. The source file system is read using a custom set of visitors that efficiently traverse the file system and create in-memory copies of relevant metadata. For example, unused block groups can be skipped while processing block group descriptors. Next, we generate the free space list by reusing components from the free space tool, and then removing F2FS's static metadata area from the list. Then, Ext4 extents in the F2FS metadata area are relocated to the free space with their mappings updated. Finally, F2FS metadata is created from the in-memory copies and written to

[3] The convert utility converts FAT32 to NTFS [27], and updating to iOS 10.3 upgrades the file system from HFS+ to APFS [28]

disk, which involves allocation and pointer management, requiring significant file-system-specific logic.

Fortunately, various pieces of the code can be reused for different combinations of source and destination file system when adapting new file systems. As an example, only the code to copy Btrfs metadata from an existing file system and to list its free space is required to support the conversion from Btrfs to F2FS, since the in-memory data structures are generic across file systems that support VFS. If the file system does not support VFS, suitable default values can be used, which would be helpful for upgrading from a legacy file system such as FAT32.

Prioritized Block Layer Cache We have implemented a file-system aware block layer cache based on Bcache [20]. Our cache preferentially caches the files of certain priority users, identified by the `uid` of the file. This caching policy can dramatically improve workload performance by improving the cache hit rate for prioritized workloads, as shown in previous work [26]. Bcache uses an LRU replacement policy; in our implementation, blocks belonging to priority users are given a second chance and are only evicted if they return to the head of the LRU list without being referenced.

We use a runtime interpretation module, described in more detail in Section 5, to identify metadata blocks at the block layer without any modifications to the file system. We track the data extents that belong to file inodes containing the `uid` of a priority user, so that we can preferentially cache these extents. For Ext4, we use custom visit functions to parse inodes and determine the priority extent nodes. Similarly, we parse the priority extent nodes to determine the priority extent leaves, which contain the priority data extents.

For Btrfs, the inodes and their file extent items may not be placed close together (e.g., within the same B-tree leaf block), and so parsing an inode object will not provide information about its extents. Fortunately, the key of a file extent item is its associated inode number, making it easy to track the file extents of priority users.

5 Implementation

We implemented a compiler that parses Spiffy annotations. The compiler generates the file system's internal representation in a symbol table, containing the definitions of all the file system metadata, their annotations, their fields (including type and symbolic name), and each of their field's annotations. Next, it detects errors such as duplicate declarations or missing required arguments. Finally, the symbol table and compiler options are exported for use by the compiler's backend.

Spiffy's backend generates C++ code for a file-system specific metadata library using Jinja2 [22]. The library can be compiled as either a user space library or as part of a Linux kernel module. We linked our module, including our generated library, into the Linux kernel by porting some C++ standard containers to the kernel environment and integrating the GNU g++ compiler into the kernel build process, which required minor changes.

Every annotated structure is wrapped in a class that allows introspection. Each field in the wrapped class can refer to its name, type and size, and has a reference to the containing structure. The generated library performs various types of error-checking operations. For example, the parsing of offset fields ensures that objects do not cross container boundaries, and that all variable-sized structures fit within their containers. These checks are essential if an application aims to handle file system corruption. When parsing does fail, an error code is propagated to the caller of the `parse` or `serialize` function.

Address Spaces Annotation developers must implement the `IO` interface shown in Table 3. The Ext4 file address space implementation for the `Ext4IO` class (see Figure 5) requires fetching the file contents associated with an inode number. For Btrfs, we currently support the RAID address space for a single device, which only allows metadata mirroring (RAID-1). For F2FS, we support the NID address space, which maps a NID (node id) to a node block. The implementation involves a lookup to see if a valid mapping entry is in the journal. If not, the mapping is obtained from the node address table.

Runtime Interpretation Offline Spiffy applications use variants of the file-system traversal algorithm in Figure 5. Spiffy also supports online file-system aware storage applications via a kernel module that performs file system interpretation at the block layer of the Linux kernel using the generated libraries. These storage applications are typically difficult to write and error prone, since manual parsing code is needed for each block type. However, our implementation only requires a small amount of bootstrap code to support any annotated file system. The rest of the code is file-system independent.

In offline applications, the fetch function reads data from disk and parses the structure. The type of the structure is known from the pointer that is passed to the fetch function. In contrast, for online interpretation, the file system performs the read, and the application just needs to parse it. The `parse_by_type` function in Table 3 allows parsing of arbitrary buffers and constructing the corresponding containers, without the need for an `IO` object to read data from disk. However, it needs to know the type of the block before parsing is possible. Our runtime interpretation depends on the fact that a pointer to a metadata block must be read before the pointed-to block is read. When a pointer is found during the parsing of a block, the module tracks the type of the pointed-to block so that its type is known when it is read.

Our module exports several functions, including `interpret_read` and `interpret_write`, that need to be placed in the I/O path to perform runtime interpretation. These functions operate on locked block buffers. The module maintains a mapping between block numbers and their types. After intercepting a completed read request, it checks whether a mapping exists, and if so, it is a metadata block and it gets parsed. Next, `process_pointers` is invoked with a visitor that adds (or updates) all the pointers that are found in the block into the mapping table. If a parsed block will be referenced later (e.g., super block), we make a copy so that it is available during subsequent parsing of structures that depend on the value of its fields (e.g., parsing the Ext4 inode block requires knowing the size of an inode, which is in the super block). The local copy is atomically replaced when a new version of the block is written to disk.

When the I/O operation is a write, the module needs to determine the type of the written block. A statically allocated block can be immediately parsed because its type will not change. For example, most metadata blocks in Ext4 are statically allocated. However, in Btrfs, the super block is the only statically allocated metadata block. For dynamically allocated blocks, the block must first be labeled as unknown and its contents cached, since its type may either be unknown or have changed. Interpretation for this block is deferred until it is referenced by a block that is subsequently accessed (either read or written), and whose type is known. At that point, the module will interpret all unknown blocks that are referenced.

Since most dynamically-typed blocks are data blocks, they should be discarded immediately to reduce memory overhead. For the Btrfs file system, this is relatively easy because metadata blocks are self-identifying. For Ext4, these blocks need to be temporarily buffered until they can be interpreted. However, we use a heuristic for Ext4 to quickly identify dynamically-typed blocks that are definitely not metadata, to reduce the memory overhead of deferred interpretation. The block is first parsed as if it were a dynamically allocated block (e.g., a directory block or extent metadata block), and if the parsing results in an error, then the block is assumed to be data and discarded. This heuristic could be used in other file systems as well because most file systems have a small number of dynamically allocated metadata block types, or their blocks are self-identifying.

The module currently relies on the file system to issue `trim` operations to detect deallocation of blocks so that stale entries can be removed from the mapping table. Since file systems do not guarantee correct implementation of `trim`, the module additionally flushes out entries for dynamically allocated blocks that have not been accessed recently. This works for a caching application, but may lead to mis-classification for other runtime applications. Accurate classification can be implemented by keeping the previous versions of blocks and comparing the versions at transaction commit time. However, it comes with a higher memory overhead [8].

File System	Line Count	Annotated	Structures
Ext4	491	113	15+10+4
Btrfs	556	151	27+4+1
F2FS	462	127	14+16+5

Table 4: File system structure annotation effort.

6 Evaluation

In this section, we discuss the effort required to annotate the structures of existing file systems, the effort required to write Spiffy applications, and the robustness of Spiffy libraries. We then evaluate the performance of our file-system conversion tool and the file-system aware block-layer caching mechanism.

6.1 Annotation Effort

Table 4 shows the effort required to correctly annotate the Ext4, Btrfs and F2FS file systems. The second column shows the number of lines of code of existing on-disk data structures in these file systems. The lines of code count was obtained using `cloc` [6] to eliminate comments and empty lines. The third column shows the number of annotation lines. This number is less than one-third of the total line count for all the file systems.

The last column is listed as $A+B+C$, with A showing no modification to the data structure (other than adding annotations), B showing the number of data structures that were added, and C showing the number of data structures that needed to be modified. Structure declarations needed to be added or modified for three reasons:

1. We break down structures that benefit from being declared as conditionally inherited types. For example, `btrfs_file_extent_item` is split into two parts: the header and an optional footer, depending on whether it contains inline data or extent information.
2. Simple structures such as Ext4 extent metadata blocks, are not declared in the original source code. However, for annotation purposes, they need to be explicitly declared. All of the added structures in Ext4 belong to this category.
3. Some data structures with a complex or backward-compatible format require modifications to enable proper annotation. For example, Ext4 inode retains its Ext3 definition in the official header file even though the `i_block` field now contains extent tree information rather than block pointers. We redefined the Ext4 inode structure and replaced `i_block` with the extent header followed by four extent entries.

6.2 Developer Effort

Dump Tool: The file system dump tool includes a file-system independent XML writer module, written in 565 lines of code. The main function for each file system is written in 40 to 50 lines of code. The dump tool is helpful for debugging issues with real file systems. In addition, an expert can verify that the annotations are correct when the output of the dump tool matches the expected contents of the file system. Therefore, this tool has become an integral part of our development process.

Type-Specific Corruptor: This tool is written in 455 lines of code, with less than 30 lines of code required for the main function of each file system. The structure that the user wants to corrupt is specified via the command line and the tool uses `process_by_type` to find it, without the need for file-system specific code.

Free Space Tool: The file system free space tool has 271 lines of file-system independent code. File-system specific parts require 76 lines for Ext4, 77 lines for Btrfs, and 194 lines for F2FS. F2FS requires more code due to the complex format of its block allocation information.

Conversion Tool: The Spiffy file system conversion tool framework is written in 504 lines of code. The code for reading Ext4 takes 218 lines, the code to convert to the F2FS file system requires 1760 lines, and the file-system developer code for F2FS, which is reused in other applications such as the dump tool, consists of 383 lines. We also wrote a manual converter tool that uses the `libext2fs` [30] library to copy Ext4 metadata from the source file system, and manually writes raw data to create an F2FS file system. The manual converter has 223 lines of Ext4 code, and 2260 lines for the F2FS code. While the two converters have similar number of lines of code, the Spiffy converter has several other benefits. For the source file system, the manual converter takes advantage of the `libext2fs` library. Writing the code to convert from a different source file system would require significant effort, and would require much more code for a file system such as ZFS that lacks a similar user-level library. On the destination side, the Spiffy converter requires many file-system specific lines of code to manually initialize each newly created object. However, Spiffy checks constraints on objects and uses the `create_container` and `save` functions to create and serialize objects in a type-safe manner, while the manual converter writes raw data, which is error-prone, leading to the types of bugs discussed in Section 2.

Prioritized Cache: The original Bcache code consisted of 10518 lines of code. To implement prioritized caching we added 289 lines to this code, which invoke our generic runtime metadata interpretation framework, consisting of 2158 lines of code. This framework provides hooks to specify file-system specific policies. Our Ext4-specific policy requires 111 lines of code, and the Btrfs-specific policy requires 134 lines of code. Currently, we have not implemented prioritized caching for F2FS, which would require tracking NAT entries, similar to how we track inode numbers for Btrfs to find file extents.

6.3 Corruption Experiments

We use our type-specific corruption tool to evaluate the robustness of Spiffy generated libraries. The experiment fills a 128MB file system image with 12,000 files and some directories, then clobbers a chosen field in a specific metadata structure (e.g., one of the inode structures) to create a corrupted file system image. We corrupt each field in each type of metadata structure three times, twice to a random value and once to zero.

The Spiffy dump tool was able to generate correctly formatted XML files in the face of arbitrary single-field corruptions for all of these images. When corruption is detected during the parsing of a container or a pointer fetch (i.e., pointer address is out-of-bound or fails a placement constraint), an error is printed and the program stops the traversal.

Table 5 describes the crashes we found when we ran existing tools on the same corrupted images. For dumpe2fs (dump tool for Ext4) v1.42.13, we found a single crash when the `s_creator_os` field of the super block is corrupted. For dump.f2fs v1.6.1-1, we observed 5 instances of segmentation faults. Three of the crashes were due to corruption in the super block, and one crash each was detected for the summary block and inode structures. We were unable to trigger any crash-related bugs in btrfs-debug-tree v4.4.

These results are not unexpected since F2FS is a relatively young file system. Btrfs uses metadata checksumming to detect corruption, and thus requires corruption to be injected before checksum generation to fully test the robustness of its dump tool. Lastly, dumpe2fs does not traverse the full file system metadata, and so does not encounter most of the metadata corruption. Our Spiffy dump tool is both more complete and more robust than dumpe2fs, without requiring significant testing effort.

We also tried an extensive set of random corruption experiments, and none of the existing tools crashed, showing that our type-specific corruptor is a useful tool for testing the robustness of these applications.

6.4 File System Conversion Performance

We compare the time it takes to perform copy-based conversion, versus using the Spiffy-based and the manually written in-place file-system conversion tools. The results are shown in Table 6. The experiments are run on an Intel 510 Series SATA SSD. We create the file set using Filebench 1.5-a3 [32] in an Ext4 partition on the SSD,

Tool Name	Structure	Field	Description
dumpe2fs	super block	`s_creator_os`	index out of bound error during OS name lookup
dump.f2fs	super block	`log_blocks_per_seg`	index out of bound error while building nat bitmap
	super block	`segment_count_main`	null pointer dereference after calloc fails
	super block	`cp_blkaddr`	double free error during error handling (no valid checkpoint)
	summary block	`n_nats`	index out of bound error during nid lookup
	inode	`i_namelen`	index out of bound error when adding null character to end of name

Table 5: List of segmentation faults found during type-specific corruption experiments.

# files	Copy Converter	Manual Conv.	Spiffy Conv.
20000	188.2 ± 3.7s	6.6 ± 0.5s	7.0 ± 0.2s
1000	192.7 ± 2.3s	3.3 ± 0.1s	3.8 ± 0.0s
100	195.1 ± 0.2s	3.3 ± 0.1s	3.7 ± 0.1s

Table 6: Time required for each technique to convert from Ext4 to F2FS for different number of files.

and then convert the partition to F2FS. The 20K file set uses the `msnfs` file size distribution with the largest file size up to 1GB. The rest of the file sets have progressively fewer small files. All file sets have a total size of 16GB. For the copy converter, we run `tar -aR` at the root of the SSD partition and save the tar file on a separate local disk. We then reformat the SSD partition and extract the file set back into the partition.

The copy converter requires transferring two full copies of the file set, and so it takes 30x to 50x longer than using the conversion tools, which only need to move data blocks out of F2FS's static metadata area and then create the corresponding F2FS metadata. Both conversion tools take more time with larger file sets since they need to handle the conversion of more file system metadata. The library-assisted conversion tool performs reasonably compared to its manually-written counterpart, with at most a 16.7% overhead for the added type-safety protection that the library offers.

6.5 Prioritized Cache Performance

We measure the performance of our prioritized block layer cache (see Section 4), and compare it against LRU caching with one or two instances of the same workload.

Our experimental setup includes a client machine connected to a storage server over a 10Gb Ethernet using the iSCSI protocol. The storage server runs Linux 3.11.2 and has 4 Intel Processor E7-4830 CPUs for a total of 32 cores, 256GB of memory and a software RAID-6 volume consisting of 13 Hitachi HDS721010 SATA2 7200 RPM disks. The client machine runs Linux 4.4.0 with Intel Processor E5-2650, and an Intel 510 Series SATA SSD that is used for client-side caching. To mimic the memory-to-cache ratio of real-world storage servers, we limit the memory on the client to 4GB and use 8GB of the SSD for write-back caching. The RAID partition is formatted with either the Ext4 or Btrfs file system and is used as the primary storage device. To avoid any

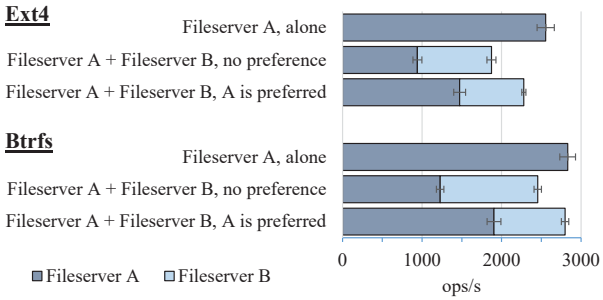


Figure 6: Throughput of prioritized caching over LRU caching with one or two file servers for Ext4 and Btrfs.

scheduling related effects, the NOOP I/O scheduler is used in all cases for both the caching and primary device.

We use a pair of identical Filebench fileserver workloads to simulate a shared hosting scenario with two users where one requires higher storage performance than the other. We generate a total file set size of 8GB with an average file size of 128KB, for each workload. The fileserver personality performs a series of create, write, append, read and delete of random files throughout the experiment. Filebench reports performance metrics every 60 seconds over a period of 90 minutes. Performance initially fluctuates as the cache fills, therefore we present the average throughput over the last 60 minutes of the experiment, after performance stabilizes.

Figure 6 shows the average throughput for each of the experiments in operations per second. The error bars show 95% confidence intervals. First, we establish the baseline performance of a single fileserver instance running alone, which has a cache hit ratio of 64% and 54% for Ext4 and Btrfs, respectively. Next, we run two instances of fileserver to observe the effect of cache contention. We see a drastic reduction in cache hit ratio to 23% and 24% for Ext4 and Btrfs, respectively. Both fileservers have similar performance, which is between 2.3x and 2.7x less than when running alone. When we apply preferential caching to the files used by fileserver A, however, its throughput improves by 60% over non-prioritized LRU caching when running concurrently with fileserver B, with the overall cache hit ratio improving to 46% and 53% for Ext4 and Btrfs, respectively. Prioritized caching also improves the aggregate throughput of the system by 14% to 22%. Giving priority to one of the two jobs implicitly reduces cache contention.

These results show that storage applications using our generated library can provide reasonable performance improvements without changing the file system code.

7 Related Work

A large body of work has focused on storage-layer applications that perform file-system specific processing for improving performance or reliability. Semantically-smart disks [24] used probing to gather detailed knowledge of file system behavior, allowing functionality or performance to be enhanced transparently at the block layer. The probing was designed for Ext4-like file systems and would likely require changes for copy-on-write and log-structured file systems. Spiffy annotations avoid the need for probing, helping provide accurate block type information based on runtime interpretation.

I/O shepherding [12] improves reliability by using file structure information to implement checksumming and replication. Block type information is provided to the storage layer I/O shepherd by modifying the file system and the buffer-cache code. Our approach enables I/O shepherding without requiring these changes. Also, unlike I/O shepherding, Spiffy allows interpreting block contents, enabling more powerful policies, such as caching the files of specific users.

A type-safe disk extends the disk interface by exposing primitives for block allocation and pointer relationships [23], which helps enforce invariants such as preventing access to unallocated blocks, but this interface requires extensive file system modifications. We believe that our runtime interpretation approach allows enforcing such type-safety invariants on existing file systems.

Serialization of structured data has been explored through interface languages such as ASN.1 [25] and Protocol Buffers [31], which allow programmers to define their data structures so that marshaling routines can be generated for them. However, the binary serialization format for the structures is specified by the protocol and not under the control of the programmer. As a result, these languages cannot be used to interpret the existing binary format of a file system.

Data description languages such as Hammer [21] and PADS [7] allow fine-grained byte-level data formats to be specified. However, they have limited support for non-sequential processing, and thus their parsers cannot interpret file system I/O, where a graph traversal is required rather than a sequential scan. Furthermore, with online interpretation, this traversal is performed on a small part of the graph, and not on the entire data.

Nail [3] shares many goals with our work. Its grammar provides the ability to specify arbitrarily computed fields. It also supports non-linear parsing, but its scope is limited to a single packet or file, and so it does not support references to external objects. Our annotation language overcomes this limitation by explicitly annotating pointers, which defines how file system metadata reference each other. We also provide support for address spaces, so that address values can be mapped to user-specified physical locations on disk.

Several projects have explored C extensions for expressing additional semantic information [19, 35, 29]. CCured [19] enables type and memory safety, and the Deputy Type System [35] prevents out-of-bound array errors. Both projects annotate source code, perform static analysis, and add runtime checks, but they are designed for in-memory structures.

Formal specification approaches for file systems [1, 5] require building a new file system from scratch, while our work focuses on building tools for existing file systems. Chen et al. [5] use logical address spaces as abstractions for writing higher-level file system specifications. This idea inspired our use of an address space type for specifying pointers. Another method for specifying pointers is by defining paths that enable traversing the metadata tree to locate a metadata object, such as finding the inode structure from an inode number [14, 10]. These approaches focus on the correctness of file-system operations at the virtual file system layer, whereas our goal is to specify the physical structures of file systems.

8 Conclusion

Spiffy is an annotation language for specifying the on-disk file system data structures. File system developers annotate their data structures using Spiffy, which enables generating a library that allows parsing and traversing file system data structures correctly.

We have shown the generality of our approach by annotating three vastly different file systems. The annotated file system code serves as detailed documentation for the metadata structures and the relationships between them. File-system aware storage applications can use the Spiffy libraries to improve their resilience against parsing bugs, and to reduce the overall programming effort needed for supporting file-system specific logic in these applications. Our evaluation suggests that applications using the generated libraries perform reasonably well. We believe our approach will enable interesting applications that require an understanding of storage structures.

Acknowledgements

We thank the anonymous reviewers and our shepherd, André Brinkmann, for their valuable feedback. We specially thank Michael Stumm, Ding Yuan, Mike Qin, and Peter Goodman for their insightful suggestions. This work was supported by NSERC Discovery.

References

[1] AMANI, S., RYZHYK, L., AND MURRAY, T. Towards a fully verified file system, 2012. EuroSys Doctoral Workshop 2012.

[2] BAIRAVASUNDARAM, L. N., RUNGTA, M., AGRAWA, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SWIFT, M. M. Analyzing the effects of disk-pointer corruption. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)* (2008), IEEE, pp. 502–511.

[3] BANGERT, J., AND ZELDOVICH, N. Nail: A practical tool for parsing and generating data formats. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 615–628.

[4] BUCKEYE, B., AND LISTON, K. Recovering deleted files in linux. `http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/SA/v11/i04/a9.htm`, 2006.

[5] CHEN, H., ZIEGLER, D., CHAJED, T., CHLIPALA, A., KAASHOEK, M. F., AND ZELDOVICH, N. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 18–37.

[6] DANIAL, A. Cloc–count lines of code. *Open source* (2009). `http://cloc.sourceforge.net/`.

[7] FISHER, K., AND WALKER, D. The pads project: an overview. In *Proceedings of the 14th International Conference on Database Theory* (2011), ACM, pp. 11–17.

[8] FRYER, D., SUN, K., MAHMOOD, R., CHENG, T., BENJAMIN, S., GOEL, A., AND BROWN, A. D. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage 8*, 4 (Dec. 2012), 15:1–15:29.

[9] GAMMA, E. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[10] GARDNER, P., NTZIK, G., AND WRIGHT, A. Local reasoning for the posix file system. In *European Symposium on Programming Languages and Systems* (2014), Springer, pp. 169–188.

[11] GEDAK, C. *Manage Partitions with GParted How-to*. Packt Publishing Ltd, 2012.

[12] GUNAWI, H. S., PRABHAKARAN, V., KRISHNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Improving file system reliability with I/O shepherding. In *Proc. of the Symposium on Operating Systems Principles (SOSP)* (2007), pp. 293–306.

[13] GUNAWI, H. S., RAJIMWALE, A., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. SQCK: A declarative file system checker. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Dec. 2008).

[14] HESSELINK, W. H., AND LALI, M. I. Formalizing a hierarchical file system. *Electronic Notes in Theoretical Computer Science 259* (2009), 67–85.

[15] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 273–286.

[16] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A study of Linux file system evolution. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2013).

[17] MA, A., DRAGGA, C., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. ffsck: The fast file system checker. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2013).

[18] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated storage services. In *Proc. of the Symposium on Operating Systems Principles (SOSP)* (2011), pp. 57–70.

[19] NECULA, G. C., MCPEAK, S., AND WEIMER, W. Ccured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2002), POPL '02, ACM, pp. 128–139.

[20] OVERSTREET, K. Linux bcache, Aug. 2016. `https://bcache.evilpiepirate.org/`.

[21] PATTERSON, M., AND HIRSCH, D. Hammer parser generator, march 2014. `https://github.com/UpstandingHackers/hammer`.

[22] RONACHER, A. Jinja2 documentation, 2011.

[23] SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. Type-safe disks. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 15–28.

[24] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)* (2003), pp. 73–88.

[25] STEEDMAN, D. *Abstract syntax notation one (ASN. 1): the tutorial and reference*. Technology appraisals, 1993.

[26] STEFANOVICI, I., THERESKA, E., O'SHEA, G., SCHROEDER, B., BALLANI, H., KARAGIANNIS, T., ROWSTRON, A., AND TALPEY, T. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), ACM, pp. 174–181.

[27] TECHNET, M. How to convert fat disks to ntfs. `https://technet.microsoft.com/en-us/library/bb456984.aspx`.

[28] TOM WARREN. Apple is upgrading millions of iOS devices to a new modern file system today. `https://www.theverge.com/2017/3/27/15076244/apple-file-system-apfs-ios-10-3-features`. Accessed: 2017-03-27.

[29] TORVALDS, L., TRIPLETT, J., AND LI, C. Sparse–a semantic parser for c. *see http://sparse.wiki.kernel.org* (2007).

[30] TS'O, T. E2fsprogs: Ext2/3/4 filesystem utilities. `http://e2fsprogs.sourceforge.net/`, 2017.

[31] VARDA, K. Protocol buffers: Google's data interchange format. *Google Open Source Blog, Available at least as early as Jul* (2008).

[32] WILSON, A. The new and improved filebench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies* (2008). `https://github.com/filebench/filebench/`.

[33] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems (TOCS) 24*, 4 (2006), 393–423.

[34] ZALEWSKI, M. American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2016.

[35] ZHOU, F., CONDIT, J., ANDERSON, Z., BAGRAK, I., ENNALS, R., HARREN, M., NECULA, G., AND BREWER, E. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 45–60.

Towards Robust File System Checkers

Om Rameshwar Gatla[†], Muhammad Hameed[†], Mai Zheng[†],
Viacheslav Dubeyko[‡], Adam Manzanares[‡], Filip Blagojevic[‡], Cyril Guyot[‡], Robert Mateescu[‡]

[†]*New Mexico State University* [‡]*Western Digital Research*

Abstract

File systems may become corrupted for many reasons despite various protection techniques. Therefore, most file systems come with a checker to recover the file system to a consistent state. However, existing checkers are commonly assumed to be able to complete the repair without interruption, which may not be true in practice.

In this work, we demonstrate via fault injection experiments that checkers of widely used file systems may leave the file system in an uncorrectable state if the repair procedure is interrupted unexpectedly. To address the problem, we first fix the ordering issue in the undo logging of `e2fsck`, and then build a general logging library (i.e., `rfsck-lib`) for strengthening checkers. To demonstrate the practicality, we integrate `rfsck-lib` with existing checkers and create two new checkers: (1) `rfsck-ext`, a robust checker for Ext-family file systems, and (2) `rfsck-xfs`, a robust checker for XFS file system, both of which require only tens of lines of modification to the original versions. Both `rfsck-ext` and `rfsck-xfs` are resilient to faults in our experiments. Also, both checkers incur reasonable performance overhead (i.e., up to 12%) comparing to the original unreliable versions. Moreover, `rfsck-ext` outperforms the patched `e2fsck` by up to nine times while achieving the same level of robustness.

1 Introduction

Achieving data integrity is critical for computer systems ranging from a single desktop to large-scale distributed storage clusters [21]. In order to make sense of the ever increasing amount of data stored, it is common to use local (e.g., Ext4 [4], XFS [70], F2FS [49]) and multi-node file systems (e.g., HDFS [66], Ceph [74], Lustre [9]) to organize the data on top of storage devices. Although file systems are designed to maintain the data integrity [36, 38, 45, 60, 72, 75], situations arise when the file system metadata needs to be checked for integrity. Such situations may be caused by power outages, server crashes, latent sector errors, software bugs, etc [19, 20, 31, 51, 54].

File system checkers, such as `e2fsck` for Ext-family file systems [3], serve as the last line of defense to recover a corrupted file system back to a healthy state [54]. They contain intimate knowledge of file system metadata structures, and are commonly assumed to be able to complete the repair *without interruption*.

Unfortunately, the same issues that lead to file system inconsistencies (e.g., power outages or crashes), can also occur during file system repair. One real-world example happened at the High Performance Computing Center in Texas [17]. In this accident, multiple Lustre file systems suffered severe data loss after power outages: the first outage triggered the Lustre checker (`lfsck` [6]) after the cluster was restarted, while another outage interrupted `lfsck` and led to the downtime and data loss. Because Lustre is built on top of a variant of Ext4 (`ldiskfs` [9]), and `lfsck` relies on `e2fsck` to fix local inconsistencies on each node, the checking and repairing is complicated (e.g., requiring several days [17]). As of today, it is still unclear which step of `lfsck`/`e2fsck` caused the uncorrectable corruptions. With the trend of increasing the storage capacity and scaling to more and more nodes, checking and repairing file systems will likely become more time-consuming and thus more vulnerable to faults. Such accidents and observation motivate us to remove the assumption that file system checkers can always finish normally without interruption.

Previous research has demonstrated that file system checkers themselves are error-prone [27, 42]. File system specific approaches have also been developed that use higher level languages to elegantly describe file system repair tasks [42]. In addition, efforts have also been made to speed up the repair procedure, which leads to a smaller window of potential data loss due to an interruption [54]. Although these efforts improve file system checkers, they do not address the fundamental issue of improving the resilience of checkers in the face of *unex-*

pected interruptions.

In this work, we first demonstrate that the checkers of widely used file systems (i.e., `e2fsck` [3] and `xfs_repair` [14]) may leave the file system in an uncorrectable state if the repair procedure is unexpectedly interrupted. We collect corrupted file system images from file system developers and additionally generate test images to trigger the repair procedure. Moreover, we develop `rfsck-test`, an automatic fault injection tool, to systematically inject faults during the repair, and thus manifest the vulnerabilities.

To address the problem exposed in our study, we analyze the undo logging feature of `e2fsck` in depth, and identify an ordering issue which jeopardizes its effectiveness. We fix the issue and create a patched version called `e2fsck-patch` which is truly resilient to faults.

However, we find that `e2fsck-patch` is inherently suboptimal as it requires extensive sync operations. To address the limitation, and to improve the checkers of other file systems, we design and implement `rfsck-lib`, a general logging library with a simple interface. Based on the similarities among checkers, `rfsck-lib` decouples the logging from the repairing, and provides an interface to log the repairing writes in fine granularity.

To demonstrate the practicality, we integrate `rfsck-lib` with existing checkers and create two new checkers: (1) `rfsck-ext`, a robust checker for Ext-family file systems, which adds 50 lines of code (LoC) to `e2fsck`; and (2) `rfsck-xfs`, a robust checker for XFS file system, which adds 15 LoC to `xfs_repair`.[1] Both `rfsck-ext` and `rfsck-xfs` are resilient to faults in our experiments. Also, both checkers incur reasonable performance overhead (i.e., up to 12%) compared to the original unreliable versions. Moreover, `rfsck-ext` outperforms `e2fsck-patch` by up to nine times while achieving the same level of fault resilience.

The rest of the paper is organized as follows. First, we introduce the background of file system checkers (§2). Next, we describe `rfsck-test` and study `e2fsck` and `xfs_repair` (§3). We analyze the ordering issue of the undo logging of `e2fsck` in §4. Then, we introduce `rfsck-lib` and integrate it with existing checkers (§5). We evaluate `rfsck-ext` and `rfsck-xfs` in §6, and discuss several issues in §7. Finally, we discuss related work (§8) and conclude (§9).

2 Background

Most file systems employ checkers to check and repair inconsistencies. The checkers are usually file system specific, and they examine different consistency rules depending on the metadata structures. We use two representative checkers as concrete examples to illustrate the complexity as well as the potential vulnerabilities of checkers in this section.

2.1 Workflow of `e2fsck`

`e2fsck` is the checker of the widely used Ext-family file systems. It first replays the journal (in case of Ext3 and Ext4) and then restarts itself. Next, `e2fsck` runs the following five passes in order:

Pass-1: Scan the file system and check inodes. `e2fsck` scans the entire volume and stores information of all inodes into a set of bitmaps. In addition, it performs four sub-passes to generate a list of duplicate blocks and their owners, check the integrity of extent trees, etc.

Pass-2: Check directory structure. Based on the bitmap information, `e2fsck` iterates through all directory inodes and checks a set of rules for each directory.

Pass-3: Check directory connectivity. `e2fsck` first checks if a root directory is available; if not, a new root directory is created and is marked "done". Then it traverses the directory tree, checks the reachability of each directory inode, breaks directory loops, etc.

Pass-4: Check reference counts. `e2fsck` iterates over all inodes to validate the inode link counts. Also, it checks the connectivity of the extended attribute blocks and reconnects them if necessary.

Pass-5: Recalculate checksums and flush updates. Finally, `e2fsck` checks the repaired in-memory data structures against on-disk data structures and flushes necessary updates to the disk.

2.2 Workflow of `xfs_repair`

`xfs_repair` is the checker of the popular XFS file system.[2] Similar to `e2fsck`, `xfs_repair` fixes inconsistencies in seven passes (or phases), including: **Pass-1**, superblock verification; **Pass-2**, replay logs, validate maps and the root inode; **Pass-3**, check inodes in each allocation group; **Pass-4**, check duplicate block allocations; **Pass-5**, rebuild the allocation group structure and superblock; **Pass-6**, check inode connectivity; **Pass-7**, verify and correct link counts.

Unlike `e2fsck` which is single-threaded, `xfs_repair` employs multi-threading in passes 2, 3, 6 and 7 to improve the performance. Nevertheless, we can see that both checkers are complicated and may be vulnerable to faults. For example, later passes may depend on previous passes, and there is no atomicity guarantee for related updates. We describe our method for systematically exposing the vulnerabilities in §3.

[1] The prototypes of `rfsck-test`, `e2fsck-patch`, `rfsck-lib`, `rfsck-ext`, and `rfsck-xfs` are publicly available [10].

[2] There is another utility called `xfs_check` [14], which checks the consistency without repairing; we do not evaluate it in this work as it is impossible for the read-only utility to introduce additional corruption.

2.3 The Logging Support of Checkers

Some file system developers have envisioned the potential need of reverting the changes done to the file system. For example, the "undo io manager" has been added to the utilities of Ext-family file systems since 2007 [3, 15]. It can save the content of the location being overwritten to an undo log before committing the overwrite.

However, due to the degraded performance as well as the log format issues [2, 16], the undo feature has not been integrated into `e2fsck` until recently. Starting from v1.42.12, `e2fsck` includes a "-z" option to allow the user to specify the path of the log file and enable logging [3]. When enabled, `e2fsck` maintains an undo log during the checking and repairing, and writes an undo block to the log before updating any block of the image. If `e2fsck` fails unexpectedly, the undo log can be replayed via `e2undo` [3] to revert the undesired changes.

Given the undo logging, one might expect that an interrupted `e2fsck` will not cause any issue. As we will see in the next section, however, this is not true.

3 Are the Existing Checkers Resilient to Faults?

In this section, we first describe our method for analyzing the fault resilience of file system checkers (§3.1 - §3.3), and then present our findings on `e2fsck` (§3.4) and `xfs_repair` (§3.5) .

3.1 Generating Corrupted Test Images

File system checkers are designed to repair corrupted file systems, so the first step of testing checkers is to generate a set of corrupted file system images to trigger the target checker. We call this set of images as *test images*.

To generate test images, we use two methods. First, some file system developers may provide test images to perform regression testing of their checkers, which usually cover the most representative corruption scenarios as envisioned by the developers [3]. We collect such default test images to trigger the target checker if they are available. Additionally, we create test images by ourselves using the debug tools provided by the file system developers (e.g., `debugfs` [3] and `xfs_db` [14]). These tools allow "trashing" specific metadata structures with random bits, which may cover corruption scenarios beyond the default test images.

In both cases, the test images are generated as regular files instead of real physical disks, which makes the testing more efficient.

3.2 Interrupting Checkers

Generating corrupted test images solves only one part of the problem. Another challenge in evaluating the fault resilience is how to interrupt checkers in a systematic and controllable way. To this end, we emulate the effect of faults using software.

To make the emulation precise and reasonable, we follow the "clean power fault" model [80], which assumes that there is a minimal atomic unit of write operations (e.g., 512B or 4KB). Under this model, the size of data written to the on-disk file system is always an integer multiple of the minimal atomic block. A fault can occur at any point during the repair procedure of the checker; once a fault happens, all atomic blocks committed before the fault are durable without corruption, and all blocks after the fault have no effect on the media.

Apparently, this is an idealized model under power outages or system crashes. More severe damage (e.g., reordering or corruption of committed blocks) may happen in practice [61, 73, 77, 81, 82]. However, such clear model can serve as a conservative lower bound of the failure impact. In other words, file system checkers must be able to handle this fault model gracefully before addressing more aggressive fault models.

Based on the fault model, we build a fault injection tool called `rfsck-test` using a customized driver [8], which has two modes of operation as follows:

Basic mode: This is used for testing a checker *without* logging support. In this mode, the target checker writes to the test image and generates I/O commands through the customized driver. `rfsck-test` records the I/O commands generated during the execution of the checker in a command history file, and replays a prefix of the command history (i.e., partial commands) to a copy of the initial test image, which effectively generates the effect of an interrupted checker on the test image. For each command history, we exhaustively replay all possible prefixes, and thus generate a set of interrupted images which correspond to injecting faults at different points during the execution of the checker.

Advanced mode: This is used for testing a checker *with* logging support. In this mode, the target checker writes to the test image as well as its log file. `rfsck-test` records the commands sent to both the image and the log in the command history. During the replay, `rfsck-test` selects a prefix of the command history, and replays the partial commands either to a copy of the initial test image or to a copy of the initial log, depending on the original destination of the commands. In this way, `rfsck-test` generates the effect of an interrupted checker on both the test image and the log. Moreover, `rfsck-test` replays the log to the test image, which is necessary for the logging to take effect.

3.3 Summary of Testing Framework

Putting it all together, we summarize our framework for testing the fault resilience of checkers with and without

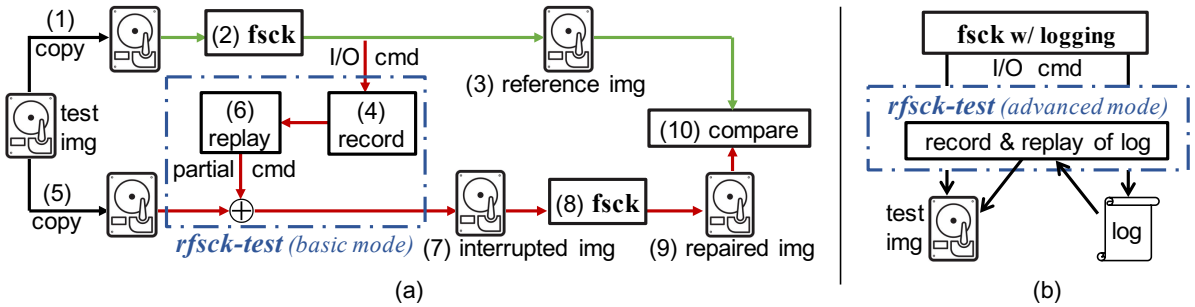


Figure 1: **(a) Testing the fault resilience of a file system checker (fsck) without logging support.** *There are ten steps: (1) make a copy of the test image which contains a corrupted file system; (2) run fsck on the test image copy; (3) store the image generated in step 2 as the reference image; (4) record the I/O commands generated during the fsck; (5) make another copy of the test image; (6) replay partial commands to emulate the effect of an interrupted fsck; (7) store the image generated in step 6 as the interrupted image; (8) run fsck on the interrupted image; (9) store the image generated in step 8 as the repaired image; (10) compare the repaired image with the reference image to identify mismatches.* **(b) Testing fsck with logging support.** *The workflow is similar except that rfsck-test interrupts the I/O commands sent to both the test image and the log, and the log is replayed between steps 7 and 8.*

logging support as follows:

Testing checkers *without* logging support: As shown in Figure 1a, there are ten steps: (1) we make a copy of the test image which contains a corrupted file system; (2) the target checker (i.e., `fsck`) is executed to check and repair the original corruption on the copy of the test image; (3) after `fsck` finishes normally in the previous step, the resulting image is stored as the *reference image*;[3] (4) during the checking and repairing of `fsck`, the fault injection tool `rfsck-test` operates in the basic mode, which records the I/O commands generated by `fsck` in a command history file; (5) we make another copy of the original test image; (6) `rfsck-test` replays partial commands recorded in step 4 to the new copy of the test image, which emulates the effect of an interrupted `fsck`; (7) the image generated in step 6 is stored as the *interrupted image*; (8) `fsck` is executed again on the interrupted image to fix any repairable issues; (9) the image generated in step 8 is stored as the *repaired image*; (10) finally, we compare the file system on the repaired image with that on the reference image to identify any mismatches.

The comparison in step 10 is first performed via the `diff` command. If a mismatch is reported, we further verify it manually. Note that in step 8 we have run `fsck` without interruption, so a mismatch implies that there is some corruption which cannot be recovered by `fsck`.

Testing checkers *with* logging support: The workflow of testing a checker with logging support is similar. As shown in Figure 1b, `rfsck-test` operates in the advanced mode, which records the I/O commands sent to both the test image and the log and emulates the effect of interruption on both places. Also, between steps 7 and 8, the (interrupted) log is replayed to the test image to make the logging take effect. The other steps are the same.

[3]It is possible that a checker may not be able to fully repair a corrupted file system even without interruption [27, 42]. So we simply use the result of an uninterrupted repair as a criterion in this work.

3.4 Case Study I: `e2fsck`

In this section, we apply the testing framework to study `e2fsck`. As discussed in §2.3, `e2fsck` has recently added the undo logging support. For clarity, we name the original version without undo logging as `e2fsck`, and the version with undo logging as `e2fsck-undo`.

To trigger the checker, we collect 175 Ext4 test images from `e2fsprogs` v1.43.1 [3] as inputs. The sizes of these images range from 8MB to 128MB, and the file system block size is 1KB. To emulate faults on storage systems with different atomic units, we inject faults at two granularities: 512B and 4KB. In other words, we interrupt `e2fsck`/ `e2fsck-undo` after every 512B or 4KB of an I/O transfer command. Since the file system block is 1KB, we do not break file system blocks when injecting faults at the 4KB granularity.

First, we study `e2fsck` using the method in Figure 1a. As described in §3.3, for each fault injected (i.e., each interruption) we run `e2fsck` again and generate one repaired image. Because the repair procedure usually requires updating multiple file system blocks, it can often be interrupted at multiple points depending on the fault injection granularity. Therefore, we usually generate multiple repaired images from one test image.

For example, to fix the test image "`f_dup`" (block claimed by two inodes), `e2fsck` needs to update 16KB in total. At the fault injection granularity of 512B, we generate 32 interrupted images (and consequently 32 repaired images). The last fault is injected after all 16KB blocks, which leads to a repaired image equivalent to the

Fault injection granularity	# of Ext4 test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
512 B	175	25,062	34	240
4 KB	175	3,192	17	37

Table 1: **Counts of images in testing e2fsck at two fault injection granularities.** *This table shows the number of repaired images (3rd column) generated from the 175 Ext4 test images when injecting faults at 512B/4KB granularities; the last two columns show the number of test images and repaired images reporting corruption respectively.*

Corruption Type	test images		repaired images	
	512 B	4 KB	512 B	4 KB
cannot mount	20	1	41	3
data corruption	9	5	107	10
misplacement	9	11	82	23
others	1	1	10	1

Table 2: **Classification of corruption.** *This table shows the number of test images and repaired images reporting different corruptions at two fault injection granularities.*

reference image without interruption. Similarly, at the 4KB granularity, we generate 4 repaired images.

For every test image, we generate a number of repaired images and compare each of them with the corresponding reference image. If the comparison reports a mismatch, it implies that the repaired image contains uncorrectable corruption. We count the number of repaired images reporting such corruption. Moreover, if at least one repaired image contains uncorrectable corruption, we mark the test image as reporting corruption, too.

Table 1 summarizes the counts of images in testing `e2fsck` at the two fault injection granularities. The total number of repaired images generated from the 175 Ext4 test images is shown in the third column. We can see that at the 512B granularity there are more repaired images (25,062) because the repairing procedure is interrupted more frequently, while at the 4KB granularity only 3,192 repaired images are generated. Also, more test images report corruption at the 512B granularity (34 $>$ 17). This is because the repair commands are broken into smaller pieces, and thus it is more challenging to maintain consistency when interrupted.

Table 2 further classifies the corruption into four types and shows the number of test images and repaired images reporting each type. Among the four types, *data corruption* (i.e., a file's content is corrupted) and *misplacement* (i.e., a file is either in the"lost+found" folder or completely missing) are the common ones. The most severe corruption is *cannot mount* (i.e., the whole file system volume becomes not mountable). Such corruption has been observed at both fault injection granularities. In other words, interrupting `e2fsck` may lead to an unmountable image, even when a fault cannot break the superblock because the 4KB fault granularity is larger than the 1KB superblock.

Fault injection granularity	# of images reporting corruption	
	e2fsck	e2fsck-undo
512 B	34	34
4 KB	17	15

Table 3: **Comparison of e2fsck and e2fsck-undo.** *This table compares the number of test images reporting corruption under e2fsck and e2fsck-undo.*

Next, to see if the undo logging can avoid the corruption, we use the method in Figure 1b to study `e2fsck-undo`. We focus on the test images which report corruption when testing `e2fsck` (i.e., the 34 and 17 test images in Table 1).

Table 3 compares the number of test images reporting corruption under `e2fsck` and `e2fsck-undo`. Surprisingly, we observe a similar amount of corruption. For example, all 34 images which report corruption when testing `e2fsck` at the 512B granularity still report corruption under `e2fsck-undo`. We defer the analysis of the root cause to §4.

3.5 Case Study II: `xfs_repair`

We have also applied the testing framework to study `xfs_repair`. Since `xfs_repair` does not support logging, only the method in Figure 1a is used.

To generate test images, we create 20 clean XFS images first. Each image is 100MB, and the file system block size is 1KB (same as the Ext4 test images). We use the `blocktrash` command of `xfs_db` [14] to flip 2 random bits on the metadata area of each image. In this way, we generate 20 corrupted XFS test images in total.

Table 4 summarizes the total number of repaired images generated from the XFS test images at two fault injection granularities. We use 3 test images to inject faults at the 512B granularity, and 17 images for the 4KB granularity. Similar to the Ext4 case, the smaller granularity (i.e., 512B) leads to more repaired images (i.e., 3 test images lead to 1,127 repaired images). The table also shows the number of test images and repaired images reporting corruption. We can see that there are uncorrectable corruptions under both granularities, same as the Ext4 case.

Fault injection injection	# of XFS test images	# of repaired images generated	# of images reporting corruption	
			test images	repaired images
512 B	3	1,127	2	443
4 KB	17	1,409	12	737

Table 4: **Counts of images in testing xfs_repair at two fault injection granularities.** *This table shows the number of repaired images (3rd column) generated from the XFS test images when injecting faults at 512B/4KB granularities; the last two columns show the number of test images and repaired images reporting corruption respectively.*

```
1.  /*open undo log*/
2.  undo_open(...){
3.      open(...); /*no O_SYNC*/
4.  }
5.  ...
6.  /*fix 1st inconsistency*/
7.  undo_write_blk64(...){
8.      /*write to undo log asynchronously*/
9.      undo_write_tdb(...){
10.         ...
11.         pwrite(...); /*no fsync()*/
12.     }
13.     /*write to fs image asynchronously*/
14.     io_channel_write_blk64(...){...}
15. }
16. /*fix 2nd, 3rd, ... inconsistencies*/
17. ...
18.
19. /*sync buffered writes to fs image*/
20. ext2fs_flush(...){...}
21. /*close undo log*/
22. undo_close(...){...}
```

Figure 2: **Workflow of the undo logging in e2fsck-undo.** *The writes to the log (line 9 -12) and the writes to the file system image (line 14) are asynchronous, and there is no ordering guarantee between the writes.*

4 Why Does the Existing Undo Logging Not Work?

The study in §3 shows that even the checkers of some most popular file systems are not resilient to faults. This is consistent with other studies on the catastrophic failures of real-world systems [41, 44], which find that the recovery procedures themselves are often imperfect, and sometimes "the cure is worse than the disease" [44].

One way to handle the faults and provide crash consistency is write-ahead logging (WAL) [58], which has been widely used in databases [12] and journaling file systems [72] for transactional recovery. While it is perhaps not surprising that file system checkers without crash consistency support (e.g., `e2fsck` and `xfs_repair`) may introduce additional corruptions upon interruption, it is counterintuitive that `e2fsck-undo`, which has the undo logging support, still cannot prevent cascading damage.

To understand the root cause, we analyze the source code of `e2fsck-undo` as well as the runtime traces (e.g., system calls and I/O commands), and have found that there is no ordering guarantee between the writes to the undo log and the writes to the image being fixed, which essentially invalidates the WAL mechanism.

To better illustrate the issue, Figure 2 shows a simplified workflow of the undo logging in `e2fsck-undo`. At the beginning of checking (line 2-4), the undo log file is opened without the `O_SYNC` flag. To fix an inconsistency, `e2fsck-undo` first gets the original content of the block being repaired (not shown) and then writes it as an undo block to the log *asynchronously* (line 9-12). After the write to the log, it updates the file system image asynchronously (line 14). The same pattern (i.e., locate the block that needs to be repaired, copy the old content to the log, and update the file system image) is repeated for fixing every inconsistency. At the end, `e2fsck-undo` flushes all buffered writes of the image to the persistent storage (line 20) and closes the undo log (line 22).

While the extensive asynchronous writes (line 6-17) is good for performance, it is problematic from the WAL's perspective. All asynchronous writes are buffered in memory. Since the dirty pages may be flushed by kernel threads due to memory pressure or timer expiry (e.g., `dirty_writeback_centisecs`), or by the internal flushing routine of the host file system, there is no strict ordering guarantee among the buffered writes. In other words, for every single fix, the writes to the log and the writes to the file system image may reach the persistent storage in an arbitrary order. Consequently, when `e2fsck-undo` is interrupted, the file system image may have been modified without the appropriate undo blocks recorded. Because the WAL mechanism works only if a log block reaches the persistent storage *before* the updated data block it describes, the lack of ordering guarantee between the writes to the log and the writes to the image invalidates the WAL mechanism. As a result, the existing undo logging does not work as expected.

5 Robust File System Checkers

In this section, we describe our method to address the problem exposed in §3 and §4.

First, we fix the ordering issue of `e2fsck-undo` by enforcing necessary sync operations. For clarity, we name the version with our patch as `e2fsck-patch`.

Next, we observe that although `e2fsck-patch` may provide the desired robustness, it inherently requires extensive synchronized I/O, which may hurt the perfor-

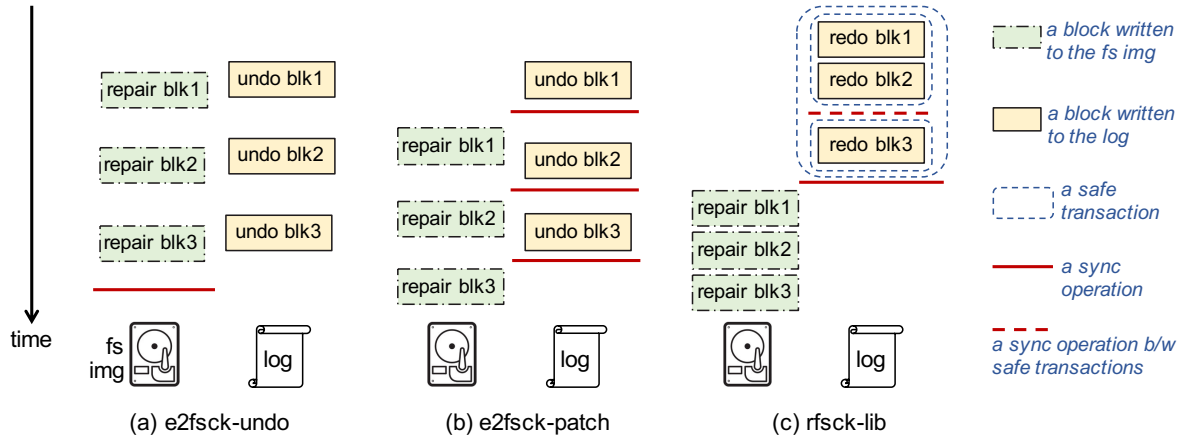


Figure 3: **Comparison of different logging schemes.** *This figure compares different logging schemes using a sequence of blocks written to the file system image (i.e., "fs img") and the log: (a) e2fsck-undo is the logging scheme of e2fsck, which does not have the necessary ordering guarantee between the writes to the log and the writes to the file system image; (b) e2fsck-patch guarantees the correct ordering between each undo block (e.g., "undo blk1") and the corresponding repair block (e.g., "repair blk1") by enforcing a sync operation (i.e., the red line) after each write of an undo block; (c) rfsck-lib uses redo logging to eliminate the frequent sync required in e2fsck-patch, and only syncs after a safe transaction which includes a set of blocks constituting a consistent update.*

mance severely. To address the limitation, and to provide a generic solution to the checkers of other file systems, we design and implement `rfsck-lib`, a general logging library with a simple interface. Different from `e2fsck-patch` which interleaves the writes to the log (i.e., *log writes*) and the writes to the image being repaired (i.e., *repair writes*), `rfsck-lib` makes use of the similarities among checkers to decouple the logging from the repairing of the file system, and provides fine-grained control of logging.

To demonstrate the practicality, we use `rfsck-lib` to strengthen existing checkers and create two new checkers: (1) `rfsck-ext`, a robust checker for Ext-series file systems, and (2) `rfsck-xfs`, a robust checker for XFS file system, both of which require only a few lines of modification to the original versions.

5.1 Goals

While there are many desired objectives, `rfsck-lib` is designed to meet the three key goals as follows:

Robustness: Unlike existing checkers which may introduce uncorrectable corruptions when interrupted, we expect checkers integrated with `rfsck-lib` to be resilient to faults. We believe such robustness should be of prime concern for file system practitioners besides the heavily studied performance issue [54].

Performance: Guaranteeing the robustness may come at the cost of performance, because it almost inevitably requires additional operations. However, the performance overhead should be reduced to minimum, which is particularly important for production environments.

Compatibility: We expect `rfsck-lib` to be compatible to existing file systems and checkers. For example, no change to the existing on-disk layouts or repair rules is needed. While such compatibility may sacrifice some flexibility and optimization opportunities, it directly enables improving the robustness of many widely used systems in practice.

5.2 `e2fsck-patch`: Fixing the Ordering Issue in e2fsck-undo

As discussed in §4, `e2fsck-undo` does not guarantee the necessary ordering between log writes and repair writes. Figure 3a illustrates the scenario using a sequence of writes. In this example, three blocks are written to the file system image (i.e., "fs img") to repair inconsistencies (i.e., "repair blk1" to "repair blk3"). Meanwhile, three blocks are written to the undo log (i.e., "undo blk1" to "undo blk3") to save the original content of the blocks being overwritten, for the purpose of undoing changes in case the repair fails. Because all blocks are written asynchronously, the repair blocks may reach the persistent storage *before* the corresponding undo blocks, which essentially invalidate the undo logging scheme. Although there is a sync operation at the end to the file system image (i.e., the red solid line), it cannot prevent the previous buffered blocks from reaching the persistent storage out of the desired order.

A naive way to solve the issue is to use a synchronous write for each block. However, this is overkill. As long as an undo block (e.g., "undo blk1") becomes persistent , it is unnecessary for the corresponding repair block (e.g.,

"repair blk1") to be written synchronously. Therefore, we only enforce synchronized I/O for the undo log file.

Specifically, we add the `O_SYNC` flag when opening the undo log file, which is equivalent to adding an `fsync` call after each write to the log [7]. As shown in Figure 3b, the simple patch guarantees that a repair block is always written *after* the corresponding undo block becomes persistent. On the other hand, all repair blocks are still written asynchronously. In this way, `e2fsck-patch` fixes `e2fsck-undo` with minimum modification.

5.3 `rfsck-lib`: A General Library for Strengthening File System Checkers

While the logging scheme of `e2fsck-patch` may improve the fault resilience, it has two limitations. First, the log writes and the repair writes are interleaved. Consequently, it requires extensive synchronized I/O to maintain the correct ordering (e.g., three sync operations are required in Figure 3b), which may incur severe performance overhead. Second, as part of `e2fsck`, the logging feature is closely tied to Ext-family file systems, and thus it cannot benefit other file system checkers directly. We address the limitations by building a general logging library called `rfsck-lib`.

5.3.1 Similarities Among File System Checkers

Different file systems usually vary a lot in terms of on-disk layouts and consistency rules. However, there are similarities among different checkers, which makes designing a general and efficient solution possible.

First of all, as a user-level utility, file system checkers always repair corrupted images through a limited number of system calls, which are irrelevant to file systems' internal structures and consistency rules. Moreover, based on our survey on popular file system checkers (e.g., `e2fsck`, `xfs_repair`, `fsck.f2fs`), we find that they always use write system calls (e.g., `pwrite` and its variants) instead of other memory-based system calls (e.g., `mmap`, `msync`). Therefore, only a few writes may cause potential cascading corruptions under faults. In other words, by focusing on the writes, we may improve different checkers.

Second, there is natural locality in checkers. Similar to the cylinder groups of FFS [56], many modern file systems have a layout consisting of relatively independent areas with an identical structure (e.g., block groups of Ext4 [4], allocation groups of XFS [70], and cubes of IceFS [52]). Among others, such common design enables co-locating related files to mitigate file system aging [33, 68] while isolating unrelated files. From the checker's perspective, most consistency rules within each area may be checked locally without referencing other areas. Also, each type of metadata usually has its unique structure and consistency rules (e.g., the `rec_len` of each directory entry in an Ext4 inode should be within a range). These local consistency rules may be checked independently without cross-checking other metadata.

Due to the locality, checkers usually consist of relatively self-contained components. For example, `e2fsck` includes five passes for checking different sets of consistency rules (§2.1). Similarly, `xfs_repair` includes seven passes, and it forks multiple threads to check multiple allocation groups separately (§2.2). Such locality exists even without changing the file system layout or reordering the checking of consistency rules [54]. Therefore, it is possible to split an existing checker into several pieces and isolate their impact under faults.

Based on the observations above, we describe `rfsck-lib`'s design in the following subsections.

5.3.2 Basic Redo Logging

A corrupted file system image is repaired by a checker through a set of repair writes. If the checker finishes without interruption, the set of writes turn the image back to a consistent state. On the other hand, if the checker is interrupted, only a subset of writes changes the image, and the resulting state may become uncorrectable. Therefore, the key of preventing uncorrectable states is to maintain the atomicity of the checker's writes. To this end, `rfsck-lib` redirects the checker's writes to a log first, and then repairs the image based on the log. Essentially, it implements a redo logging scheme [58].

As shown in Figure 3c, all repair writes are issued to the redo log first (i.e., "redo blk1" to "redo blk3"). After the write of the last redo block (i.e., "redo blk3"), a sync operation (i.e., the red solid line) is issued to make the redo blocks persistent. After the sync operation returns, the image is repaired (i.e., "repair blk1" to "repair blk3") based on the redo log. Compared with `e2fsck-patch` in Figure 3b, the log writes and the repair writes are separated, and the required number of sync operations is reduced from three to one. Such improvement in terms of sync overhead can be more significant if more blocks on the image need to be repaired.

5.3.3 Fine-grained Logging with Safe Transactions

While the basic redo logging scheme reduces the ordering constraint to minimum, there is one limitation: if a fault happens before the final sync operation finishes, all checking and repairing effort may be lost. In some complicated cases where the checker may take hours to finish [54], the waste is undesirable. On the other hand, a checker may be split into relatively independent pieces due to the locality (§5.3.1). Therefore, `rfsck-lib` extends the basic redo logging with safe transactions.

A *safe transaction* is a set of repair writes which will not lead to uncorrectable inconsistencies if they are written to the file system image atomically. In the simplest case, the whole checker (i.e., the complete set of all re-

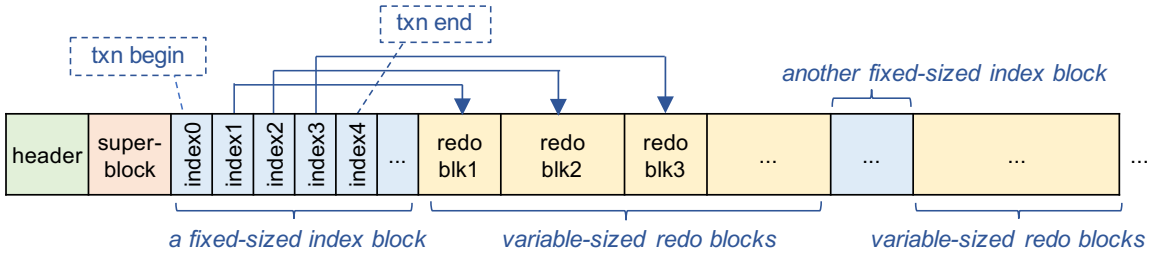


Figure 4: **The log format of rfsck-lib.** *The log includes a header, a superblock, fixed-sized index blocks, and variable-sized redo blocks. Each index block includes a fixed number of indexes. Each index can either describe the beginning/end of a transaction (i.e.,"txn begin"/"txn end"), or describe one variable-sized redo block. "index0" to "index4" describe one safe transaction with three redo blocks (i.e.,"redo blk1" to "redo blk3") in this example.*

pair writes) is one safe transaction. At a finer granularity, each pass of the checker (or the check of each allocation group) may be considered as one safe transaction. While a later pass may depend on the result of a previous pass, the previous pass is executed without any dependency on the later passes. Therefore, by guaranteeing the atomicity of each pass as well as the ordering among pass-based safe transactions, the repair writes may be committed in several batches without introducing uncorrectable inconsistencies. In the extreme case, the checking and repairing of each individual consistency rule may be considered as one safe transaction.

Figure 3c illustrates the safe transactions. In the simplest case, all three redo blocks (i.e., "redo blk1" to "redo blk3") constitute one safe transaction, and only one sync operation (i.e., the red solid line) is needed, same as the basic redo logging (§5.3.2). At a finer granularity, the first two redo blocks (i.e., "redo blk1" and "redo blk2") may constitute one safe transaction (e.g., updating an inode and the corresponding bitmap), and the third block itself (i.e., "redo blk3") may be another safe transaction (e.g., updating another inode). Another sync operation (i.e., the red dash line) is issued between the two transactions to guarantee the correct ordering. If a crash happens between the two sync operations, the first safe transaction (i.e., "redo blk1" and "redo blk2") is still valid. In this case, instead of re-calculating the rules and regenerating the blocks, the checker can directly replay the valid transaction from the log after restart.

In summary, a checker may be logged as one or more safe transactions. Compared to the basic redo logging, such fine-grained control avoids losing all recovery effort before the fault. On the other hand, maintaining the atomicity as well as the ordering requires additional sync operations. So there is a tradeoff between the transaction granularity and the transaction overhead. Since different systems may have different preferences, `rfsck-lib` simply provides an interface to define safe transactions, without restricting the number of transactions.

5.3.4 Log Format

To support the redo logging with safe transactions, `rfsck-lib` uses a special log format extended from `e2fsck-undo`. As shown in Figure 4, the log includes a header, a superblock, fixed-sized index blocks, and variable-sized redo blocks.

The header starts with a magic number to distinguish the log from other files. Besides, it includes the offsets of the superblock and the first index block, the total number of index blocks, a flag showing whether the log has been replayed, and a checksum of the header itself.

The superblock is copied from the file system image to be repaired, which is used to match the log with the image to avoid replaying an irrelevant log to the image.

The index block includes a fixed number of indexes. Each index can describe the beginning of a transaction (i.e., "txn begin"), the end of a transaction (i.e., "txn end"), or one variable-sized redo block. Therefore, a group of indexes can describe one safe transaction together. For example, in Figure 4 five indexes (i.e., "index0" to "index4") describe one safe transaction with three redo blocks (i.e.,"redo blk1" to "redo blk3").

As shown in Table 5, an index has 16 bytes consisting of three fields. To describe one redo block, the first field (i.e., `uint32_t cksum`) stores a checksum of the redo block, the second field (i.e., `uint32_t size`) stores its size, and the third field (i.e., `uint64_t fs_lba`) stores its logical block address (LBA) in the file system image.

To describe "txn begin" or "txn end", the first field of the index is repurposed to store a transaction ID instead of a checksum, which marks the boundary of indexes belonging to the same transaction. The second field (`size`) is set to zero. Since a valid redo block must have a non-zero size, `rfsck-lib` can differentiate "txn begin" or "txn end" indexes from those describing redo blocks even if a transaction ID happens to collide with a checksum. In addition, the "txn begin" index uses the third field to denote whether the transaction has been replayed or not, and the "txn end" index uses the third field to store a checksum of all indexes in the transaction.

Field	Description
uint32_t cksum	checksum of the redo block
uint32_t size	size of the redo block
uint64_t fs_lba	LBA in the file system image

Table 5: **The structure of an index**.

Function	Description
rfsck_get_sb	get the superblock
rfsck_open	create a redo log
rfsck_txn_begin	begin a safe transaction
rfsck_write	write a redo block
rfsck_txn_end	end of a safe transaction
rfsck_replay	replay the redo log
rfsck_close	close the redo log

Table 6: **The interface of rfsck-lib**. *rfsck_get_sb is a wrapper function for invoking file-system-specific procedure to get the superblock, while the others are file-system agnostic.*

For each write of the checker, rfsck-lib creates an index in the index block and then append the content of the write to the area after the index block as a redo block. Since the writes may have different sizes, the redo blocks may vary in size as well. However, since all other metadata blocks (i.e., header, superblock, index blocks) have known fixed sizes, the offset of a redo block in the log can be identified by accumulating the sizes of all previous blocks. In other words, there is no need to maintain the offsets of redo blocks in the log.

When an index block becomes full, another index block is allocated after the previous redo blocks (which are described by the previous index block). In this way, rfsck-lib can support various numbers of writes and transactions.

5.3.5 Interface

To enable easy integration with existing checkers, rfsck-lib provides a simple interface. As shown in Table 6, there are seven function calls in total. The first function (rfsck_get_sb) is a wrapper for invoking a file-system-specific procedure to get the superblock, which is written to the second part of the log (Figure 4). Since all checkers need to read the superblock anyway, rfsck_get_sb can wrap around the existing procedure.

rfsck_open is used to create a log file at a given path at the beginning of the checker procedure. Internally, rfsck-lib initializes the metadata blocks of the log.

rfsck_txn_begin is used to denote the beginning of a safe transaction, which creates a "txn begin" index in the log. Similarly, rfsck_txn_end denotes the end of a transaction, which generates a "txn end" index and sync all updates to the log. All writes between rfsck_txn_begin and rfsck_txn_end are replaced with rfsck_write, which creates a redo block and the corresponding index in the log.

rfsck_replay is used to replay logged transactions to the file system image. Besides, similar to the e2undo utility [3], the replay functionality is also implemented as an independent utility called rfsck-redo, which can replay an existing (potentially incomplete) log to a file system image (e.g., after the checker is interrupted). rfsck-redo first verifies if the log belongs to the image (based on the superblock). If yes, rfsck-redo further verifies the integrity of the log based on metadata and then replays valid transactions. Note that the additional verifications are only needed when the log is replayed by rfsck-redo. The rfsck_replay function can skip these verifications as it is invoked directly after the logging by the (uninterrupted) checker.

Finally, rfsck_close is used at the end of the checker to release all resources used by rfsck-lib and exist.

5.3.6 Limitations

The current prototype of rfsck-lib is not thread-safe. Therefore, if a checker is multi-threaded (e.g., xfs_repair), using rfsck-lib may require additional attention to avoid race conditions on logging. However, as we will demonstrate in §5.4 and §6, rfsck-lib can still be applied to strengthen xfs_repair.

In addition, rfsck-lib only provides an interface, which requires manual modification of the source code. Since the modification is simple, we expect the manual effort to be acceptable. Also, it is possible to use compiler infrastructures [11, 13] to automate the code instrumentation, which we leave as future work.

5.4 Integration with Existing Checkers

Strengthening an existing checker using rfsck-lib is straightforward given the simple interface (§5.3.5). To demonstrate the practicality, we first integrate rfsck-lib with e2fsck, and create a robust checker for Ext-family file systems (i.e., rfsck-ext).

There are potential writes to the file system image in each pass of e2fsck (including the first scanning pass), so we create a safe transaction for each pass. Moreover, within Pass-1 and Pass-2 (§2.1), there are a few places where e2fsck explicitly flushes the writes to the image and restarts scanning from the beginning (via goto statement). In other words, the restarted scanning (and subsequent passes) requires the previous writes to be visible on the image. In this case, we insert additional rfsck_txn_end and rfsck_replay before the goto statement to guarantee that previous writes are visible on the image for re-scanning. We add a "-R" option to allow the user to specify the log path via command line. In total, we add 50 LoC to e2fsck.

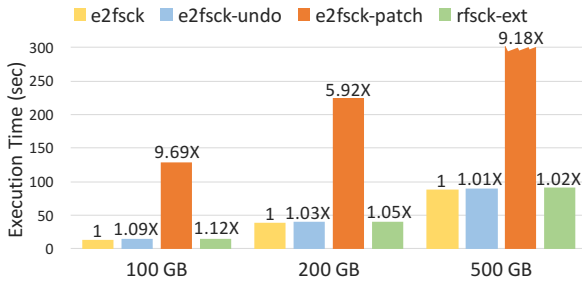


Figure 5: **Performance comparison of e2fsck, e2fsck-undo, e2fsck-patch, and rfsck-ext.** *This figure compares the execution time of e2fsck, e2fsck-undo, e2fsck-patch, and rfsck-ext. The y-axis shows the execution time in seconds. The x-axis shows file system sizes. The number above each bar indicates the normalized time (relative to e2fsck). Note: e2fsck and e2fsck-undo are not resilient to faults.*

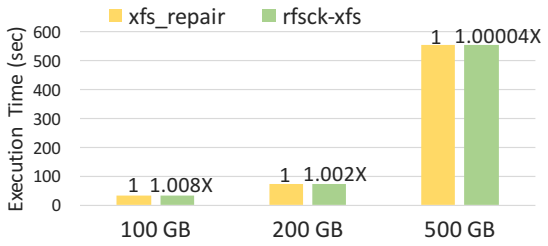


Figure 6: **Performance comparison of xfs_repair and rfsck-xfs.** *This figure compares the execution time of xfs_repair and rfsck-xfs. The y-axis shows the execution time in seconds. The x-axis shows file system sizes. The number above each bar indicates the normalized time (relative to xfs_repair). Note: xfs_repair is not resilient to faults.*

Similarly, we also integrate `rfsck-lib` with `xfs_repair`, and create a robust checker for XFS file system (i.e., `rfsck-xfs`). As mentioned in (§2.2), one feature of `xfs_repair` is multi-threading: it forks multiple threads to repair multiple allocation groups in parallel. The threads update in-memory structures concurrently, and the main thread writes all updates to the image at the end. Although it is possible to encapsulate each repair thread into one safe transaction, doing so requires additional concurrency control. To minimize the modification, we simply treat the whole repair procedure as one transaction. Since all writes are issued by the main thread, there is no race condition for `rfsck-lib`. We also add a "-R" command line option. In total, we add 15 LoC to `xfs_repair`.

6 Evaluation

In this section, we evaluate `rfsck-ext` and `rfsck-xfs` in terms of robustness (§6.1) and performance (§6.2).

Our experiments were conducted on a machine with a Intel Xeon 3.00GHz CPU, 8GB main memory, and two WD5000AAKS hard disks. The operating system is Ubuntu 16.04 LTS with kernel v4.4. To evaluate the robustness, we used the test images reporting corruption under `e2fsck-undo` (§3.4) and `xfs_repair` (§3.5). To evaluate the performance, we created another set of images with practical sizes, and measured the execution time of `e2fsck`, `e2fsck-undo`, `e2fsck-patch`, `rfsck-ext`, `xfs_repair`, and `rfsck-xfs`. For each checker, we report the average time of three runs.

In general, we demonstrate that both `rfsck-ext` and `rfsck-xfs` can survive fault injection experiments. Also, both checkers incur reasonable performance overhead (i.e., up to 12%) compared to the original unreliable versions. Moreover, `rfsck-ext` outperforms `e2fsck-patch` by up to 9 times while achieving the same level of robustness.

6.1 Robustness

As discussed in §3, when injecting faults at the 4KB granularity, 17 Ext4 test images report corruptions under `e2fsck` (Table 1), and 12 XFS test images report corruptions under `xfs_repair` (Table 4). We use these test images to trigger `rfsck-ext` and `rfsck-xfs`, respectively. Since both checkers have the logging support, we use the method in Figure 1b to evaluate them.

For `rfsck-ext`, all 17 test images report no corruptions. Similarly, for `rfsck-xfs`, all 12 test images report no corruptions. This result verifies that `rfsck-lib` can help improve the fault resilience of existing checkers.

6.2 Performance

The test images used in §3 are created as regular files, and they are small in size (i.e., 8MB to 128MB). Therefore, they are unsuitable for evaluating the execution time of checkers. So we create another set of Ext4 and XFS test images with practical sizes (i.e., 100G, 200GB, 500GB) on real hard disks. We first fill up the entire file system by running `fs_mark` [5] for five times. Each time `fs_mark` fills up 20% of the capacity by creating directories and files with a certain size. The file size is a random value between 4KB to 1MB, which is relatively small in order to maximize the number of inodes used. After filling up the entire file system, we inject 2 random bit corruptions to the metadata using either `debugfs` [1] (for Ext4) or `blocktrash` [14] (for XFS). We measure the execution time of checkers on corrupted images, and verify that the repair results of `rfsck-ext` and `rfsck-xfs` are the same as that of the original checkers.

Figure 5 compares the execution time of `e2fsck`, `e2fsck-undo`, `e2fsck-patch`, and `rfsck-ext` on different images. For each size of image, the bars represent the execution time in seconds (y-axis). Also, the number

above each bar shows the normalized execution time (relative to e2fsck). We can see that rfsck-ext incurs up to 12% overhead, while e2fsck-patch may incur more than 8 times overhead due to extensive sync operations.

Also, we can see that as the size of file system increases, the overhead of rfsck-ext decreases. This is because the execution time of rfsck-ext is largely dominated by the scanning in Pass-1 (§2.1) which is proportional to the file system size, similar to e2fsck [54].

Similarly, Figure 6 compares the execution time of xfs_repair and rfsck-xfs. We can see that rfsck-xfs incurs up to 0.8% overhead, and the overhead also decreases as the file system size increases.

Note that our aging method is relatively simple compared to other aging techniques [33, 68]. Also, the 2-random-bit corruption may not necessarily lead to extensive repair operations of checkers. Therefore, the execution time measured here may not reflect the complexity of checking and repairing real-world file systems (which may take hours [35, 34, 54, 69]). We leave generating more representative file systems as future work.

7 Discussion

Co-designing file systems and checkers. Recent work has demonstrated the benefits of co-designing file systems and checkers. For example, by co-designing rext3 and ffsck, ffsck may be 10 times faster than e2fsck [54]. In contrast, rfsck-lib is designed to be file system agnostic, which makes it directly applicable to existing systems. We believe checkers may be improved further in terms of both reliability and performance by co-designing, and we leave it as future work.

Other reliability techniques. There are other techniques which may mitigate the impact of an inconsistent file system image or the loss of an entire image (e.g., replication [39]). However, maintaining the consistency of local file systems and improving the checkers is still important for many reasons. For example, a consistent local file system is the building block of large-scale file systems, and the local checker may be the foundation of higher-level recovery procedures (e.g., lfsck [6]). Therefore, our work is orthogonal to these other efforts.

Robustness. We evaluate the robustness of checkers based on fault injection experiments in this work. The test images we use are limited, and may not cover all corruption scenarios or trigger all code paths of the checkers. There are other techniques (e.g., symbolic execution and formal verification) which might provide more coverage, and we leave it as future work.

8 Related Work

Reliability of file system checkers. Gunawi *et al.* [42] find that the Ext2 checker may create inconsistent or even insecure repairs; they then propose a more elegant checker (i.e., SQCK) based on a declarative query language. Carreira *et al.* [27] propose a tool (i.e., SWIFT) to test checkers using a mix of symbolic and concrete execution; they tested five popular checkers and found bugs in all of them. Ma *et al.* [54] change the structure of Ext3 and co-design the checker, which enables faster checking and thus narrows the window of vulnerability. Generally, these studies consider the behavior of checkers during normal executions (i.e., no interruption). Complimentarily, we study checkers under faults.

Reliability of file systems. Great efforts have been put towards improving the reliability of file systems [23, 29, 32, 36, 43, 51, 55, 57, 62, 67, 77, 79]. For example, Prabhakaran *et al.* [62] analyze the failure policies of four file systems and propose the IRON file system which implements a family of novel recovery techniques. Fryer *et al.* [36] transform global consistency rules to local consistency invariants and enable fast runtime checking. CrashMonkey [55] provides a framework to automatically test the crash consistency of file systems. Overall, these research help understand and improve the reliability of file systems, which may reduce the need for checkers. However, despite these efforts, checkers remain a necessary component for most file systems.

Reliability of storage devices. In terms of storage devices, research efforts are also abundant [19, 20, 30, 47, 63, 64]. For example, Bairavasundaram *et al.* [19, 20] analyze the data corruption and latent sector errors in production systems containing a total of 1.53 million HDDs. Besides HDDs, more recent work has been focused on flash memory and solid state drives (SSDs) [18, 22, 24, 25, 26, 28, 37, 40, 46, 48, 50, 53, 59, 65, 71, 73, 76, 78, 81, 82]. These studies provide valuable insights for understanding file system corruptions caused by hardware.

9 Conclusion

We have studied the behavior of file system checkers under faults. We find that running the checker after an interrupted repair may not return the file system to a valid state. To address the issue, we have built a general logging library which can help strengthen existing checkers with little modification. We hope our work will raise the awareness of reliability vulnerabilities in storage systems, and facilitate building truly fault-resilient systems.

10 Acknowledgements

We thank the anonymous reviewers and Keith Smith (our shepherd) for their insightful feedback. We also thank Linux practitioners including Theodore Ts'o and Ric Wheeler for the invaluable discussion. This work was supported in part by NSF under grants CNS-1566554 and CCF-1717630. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] debugfs. http://man7.org/linux/man-pages/man8/debugfs.8.html.

[2] Discussion with Theodore Ts'o at Linux FAST Summit'17. https://www.usenix.org/conference/linuxfastsummit17.

[3] E2fsprogs: Ext2/3/4 Filesystem Utilities. http://e2fsprogs.sourceforge.net/.

[4] Ext4 File System. https://ext4.wiki.kernel.org/index.php/Main_Page.

[5] fs_mark: Benchmark file creation. https://github.com/josefbacik/fs_mark.

[6] LFSCK: an online file system checker for Lustre. https://github.com/Xyratex/lustre-stable/blob/master/Documentation/lfsck.txt.

[7] Linux Programmer's Manual: O_SYNC flag for open. http://man7.org/linux/man-pages/man2/open.2.html.

[8] Linux SCSI target framework (tgt). http://stgt.sourceforge.net/.

[9] Lustre File System. http://opensfs.org/lustre/.

[10] Prototypes of rfsck-test, e2fsck-patch, refsck-lib, refsck-ext, rfsck-xfs. https://www.cs.nmsu.edu/~mzheng/lab/lab.html.

[11] ROSE Compiler Infrastructure. http://rosecompiler.org/.

[12] SQLite documents. http://www.sqlite.org/docs.html.

[13] The LLVM Compiler Infrastructure. https://llvm.org/.

[14] XFS File System Utilities. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Storage_Administration_Guide/xfsothers.html.

[15] [PATCH 1/3] e2fsprogs: Add undo I/O manager. http://lists.openwall.net/linux-ext4/2007/07/25/2, 2007.

[16] [PATCH 16/31] e2undo: ditch tdb file, write everything to a flat file. http://lists.openwall.net/linux-ext4/2015/01/08/1, 2015.

[17] High Performance Computing Center (HPCC) Power Outage Event. Email Annoucement by HPCC, Monday, January 11, 2016 at 8:50:17 AM CST. https://www.cs.nmsu.edu/~mzheng/docs/failures/2016-hpcc-outage.pdf, 2016.

[18] Nitin Agarwal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance, 2008.

[19] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage*, 4(3):8:1–8:28, November 2008.

[20] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*, pages 289–300, 2007.

[21] Luiz Andre Barroso and Urs Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[22] Hanmant P Belgal, Nick Righos, Ivan Kalastirsky, Jeff J Peterson, Robert Shiner, and Neal Mielke. A new reliability model for post-cycling charge retention of flash memories. In *Proceedings of the 40th Annual Reliability Physics Symposium*, pages 7–20. IEEE, 2002.

[23] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, 51(4):83–98, 2016.

[24] Adam Brand, Ken Wu, Sam Pan, and David Chin. Novel read disturb failure mechanism induced by FLASH cycling. In *Proceedings of the 31st Annual Reliability Physics Symposium*, pages 127–132. IEEE, 1993.

[25] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*, pages 521–526, 2012.

[26] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Osman Unsal, Adrian Cristal, and Ken Mai. Neighbor-cell assisted error correction for MLC NAND flash memories. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 491–504. ACM, 2014.

[27] João Carlos Menezes Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable testing of file system checkers. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, pages 239–252, 2012.

[28] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the ACM Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*, 2009.

[29] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 18–37. ACM, 2015.

[30] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[31] 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.

[32] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*, February 2012.

[33] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 45–58, 2017.

[34] GParted Forum. e2fsck is taking forever. http://gparted-forum.surf4.info/viewtopic.php?id=13613, 2009.

[35] JaguarPC Forum. How long does it take FSCK to run?! http://forums.jaguarpc.com/hosting-talk-chit-chat/14217-how-long-does-take-fsck-run.html, 2006.

[36] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th Conference on File and Storage Technologies (FAST'12)*, February 2012.

[37] Ryan Gabrys, Eitan Yaakobi, Laura M. Grupp, Steven Swanson, and Lara Dolecek. Tackling intracell variability in TLC flash through tensor product codes. In *Proceedings of IEEE International Symposium of Information Theory*, pages 1000–1004, 2012.

[38] Gregory R Ganger, Marshall Kirk McKusick, Craig AN Soules, and Yale N Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS'00)*, 18(2):127–153, 2000.

[39] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 29–43, 2003.

[40] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*, pages 24–33, 2009.

[41] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of ACM Symposium on Cloud Computing (SoCC'16)*, pages 1–16, 2016.

[42] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sqck: A declarative file system checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, pages 131–146, 2008.

[43] Haryadi S Gunawi, Cindy Rubio-González, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Ben Liblit. Eio: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, volume 8, pages 1–16, 2008.

[44] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure Recovery: When the Cure Is Worse Than the Disease. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS'13)*, 2013.

[45] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*, pages 15–26, 2012.

[46] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: improving nand flash lifetime by balancing page endurance. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 47–59, 2014.

[47] 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 13th USENIX Conference on File and Storage Technologies (FAST'08)*, volume 8, pages 1–15, 2008.

[48] H Kurata, K Otsuga, A Kotabe, S Kajiyama, T Osabe, Y Sasago, S Narumi, K Tokami, S Kamohara, and O Tsuchiya. The impact of random telegraph signals on the scaling of multilevel flash memories. In *Proceedings of the 2006 Symposium on VLSI Circuits*, pages 112–113. IEEE, 2006.

[49] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 273–286, 2015.

[50] Jiangpeng Li, Kai Zhao, Xuebin Zhang, Jun Ma, Ming Zhao, and Tong Zhang. How much can data compressibility help to improve nand flash memory lifetime? In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 227–240, 2015.

[51] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 31–44, 2013.

[52] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 81–96, 2014.

[53] Youyou Lu, Jiwu Shu, Weimin Zheng, et al. 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)*, volume 13, 2013.

[54] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The fast file system checker. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 1–15, 2013.

[55] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A Framework to Automatically Test File-System Crash Consistency. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)*, 2017.

[56] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *Proceedings of the ACM Transactions on Computer Systems (TOCS'84)*, 2(3):181–197, August 1984.

[57] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 361–377. ACM, 2015.

[58] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transactions on Database Systems (TODS'92)*, 1992.

[59] T. Ong, A. Frazio, N. Mielke, S. Pan, N. Righos, G. Atwood, and S. Lai. Erratic Erase In ETOX/sup TM/ Flash Memory Array. In *Proceedings of the Symposium on VLSI Technology (VLSI'93)*, 1993.

[60] Lluis Pamies-Juarez, Filip Blagojević, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandić. Opening the chrysalis: On the real repair performance of MSR codes. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 81–94, 2016.

[61] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, October 2014.

[62] 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, October 2005.

[63] Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Coerced cache eviction and discreet mode journaling: Dealing with misbehaving disks. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN'11)*, pages 518–529. IEEE, 2011.

[64] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[65] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 67–80, February 2016.

[66] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, pages 1–10. IEEE, 2010.

[67] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-button verification of file systems via crash refinement. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[68] Keith A. Smith and Margo I. Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'97)*, pages 203–213, 1997.

[69] V. Svanberg. Fsck takes too long on multiply-claimed blocks. http://old.nabble.com/Fsck-takes-too-long-on-multiply-claimed-blocks-td21972943.html, 2009.

[70] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the 1996 USENIX Annual Technical Conference (USENIX ATC'96)*, volume 15, 1996.

[71] Huang-Wei Tseng, Laura M. Grupp, and Steven Swanson. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference (DAC'11)*, 2011.

[72] Stephen C. Tweedie. Journaling the linux ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo*, 1998.

[73] Simeng Wang, Jinrui Cao, Danny V Murillo, Yiliang Shi, and Mai Zheng. Emulating Realistic Flash Device Errors with High Fidelity. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS'16)*. IEEE, 2016.

[74] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, 2006.

[75] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. A tale of two erasure codes in HDFS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 213–226, 2015.

[76] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. Write once, get 50% free: Saving ssd erase costs using wom codes. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, pages 257–271, 2015.

[77] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 131–146, November 2006.

[78] 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.

[79] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*, pages 29–42, 2010.

[80] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 449–464, 2014.

[81] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of SSDs under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[82] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W Zhao, and Elizabeth S. Yang. Reliability Analysis of SSDs under Power Fault. In *Proceedings of the ACM Transactions on Computer Systems (TOCS'16)*, 2016.

The Full Path to Full-Path Indexing

Yang Zhan, Alex Conway[†], Yizheng Jiao, Eric Knorr[†], Michael A. Bender[‡],
Martin Farach-Colton[†], William Jannen[¶], Rob Johnson[*], Donald E. Porter, Jun Yuan[‡]
The University of North Carolina at Chapel Hill, [†]Rutgers University,
[‡]Stony Brook University, [¶]Williams College, [*]VMware Research

Abstract

Full-path indexing can improve I/O efficiency for workloads that operate on data organized using traditional, hierarchical directories, because data is placed on persistent storage in scan order. Prior results indicate, however, that renames in a local file system with full-path indexing are prohibitively expensive.

This paper shows how to use full-path indexing in a file system to realize fast directory scans, writes, and renames. The paper introduces a range-rename mechanism for efficient key-space changes in a write-optimized dictionary. This mechanism is encapsulated in the key-value API and simplifies the overall file system design.

We implemented this mechanism in BetrFS, an in-kernel, local file system for Linux. This new version, BetrFS 0.4, performs recursive greps 1.5x faster and random writes 1.2x faster than BetrFS 0.3, but renames are competitive with indirection-based file systems for a range of sizes. BetrFS 0.4 outperforms BetrFS 0.3, as well as traditional file systems, such as ext4, XFS, and ZFS, across a variety of workloads.

1 Introduction

Today's general-purpose file systems do not fully utilize the bandwidth of the underlying hardware. For example, ext4 can write large files at near disk bandwidth but typically creates small files at less than 3% of disk bandwidth. Similarly, ext4 can read large files at near disk bandwidth, but scanning directories with many small files has performance that ages over time. For instance, a git version-control workload can degrade ext4 scan performance by up to 15× [14, 55].

At the heart of this issue is how data is organized, or ***indexed***, on disk. The most common design pattern for modern file systems is to use a form of indirection, such as inodes, between the name of a file in a directory and its physical placement on disk. Indirection simplifies implementation of some metadata operations, such as renames or file creates, but the contents of the file system can end up scattered over the disk in the worst case. Cylinder groups and other best-effort heuristics [32] are designed to mitigate this scattering.

Full-path indexing is an alternative to indirection, known to have good performance on nearly all operations. File systems that use full-path indexing store data and metadata in depth-first-search order, that is, lexicographic order by the full-path names of files and directories. With this design, scans of any directory subtree (e.g., `ls -R` or `grep -r`) should run at near disk bandwidth. The challenge is maintaining full-path order as the file system changes. Prior work [15, 22, 23] has shown that the combination of write-optimization [5–7, 9–11, 36, 43, 44] with full-path indexing can realize efficient implementations of many file system updates, such as creating or removing files, but a few operations still have prohibitively high overheads.

The Achilles' heel of full-path indexing is the performance of renaming large files and directories. For instance, renaming a large directory changes the path to every file in the subtree rooted at that directory, which changes the depth-first search order. Competitive rename performance in a full-path indexed file system requires making these changes in an I/O-efficient manner.

The primary contribution of this paper is showing that one *can*, in fact, use full-path indexing in a file system without introducing unreasonable rename costs. A file system can use full-path indexing to improve directory locality—and still have efficient renames.

Previous full-path indexing. The first version of BetrFS [22, 23] (v0.1), explored full-path indexing. BetrFS uses a write-optimized dictionary to ensure fast updates of large and small data and metadata, as well as fast scans of files and directory-tree data and metadata. Specifically, BetrFS uses two B^ε-trees [7, 10] as persistent key-value stores, where the keys are full path names to files and the values are file contents and metadata, respectively. B^ε-trees organize data on disk such that logically contiguous key ranges can be accessed via large, sequential I/Os. B^ε-trees aggregate small updates into large, sequential I/Os, ensuring efficient writes.

This design established the promise of full-path indexing, when combined with B^ε-trees. Recursive greps run 3.8x faster than in the best standard file system. File creation runs 2.6x faster. Small, random writes to a file run 68.2x faster.

However, renaming a directory has predictably miserable performance [22, 23]. For example, renaming the Linux source tree, which must delete and reinsert all the data to preserve locality, takes 21.2s in BetrFS 0.1, as compared to 0.1s in btrfs.

Relative-path indexing. BetrFS 0.2 backed away from full-path indexing and introduced zoning [54, 55]. Zoning is a schema-level change that implements ***relative-***

path indexing. In relative-path indexing, each file or directory is indexed relative to a local neighborhood in the directory tree. See Section 2.2 for details.

Zoning strikes a "sweet spot" on the spectrum between indirection and full-path indexing. Large file and directory renames are comparable to indirection-based file systems, and a sequential scan is at least 2x faster than inode-based file systems but 1.5x slower than BetrFS 0.1.

There are, however, a number of significant, diffuse costs to relative-path indexing, which tax the performance of seemingly unrelated operations. For instance, two-thirds of the way through the TokuBench benchmark, BetrFS 0.2 shows a sudden, precipitous drop in cumulative throughput for small file creations, which can be attributed to the cost of maintaining zones.

Perhaps the most intangible cost of zoning is that it introduces complexity into the system and thus hides optimization opportunities. In a full-path file system, one can implement nearly all file system operations as simple point or range operations. Adding indirection breaks this simple mapping. Indirection generally causes file system operations to map onto more key/value operations and often introduces reads before writes. Because reads are slower than writes in a write-optimized file system, making writes depend upon reads forgoes some of the potential performance benefits of write-optimization.

Consider `rm -r`, for example. With full-path indexing, one can implement this operation with a single range-delete message, which incurs almost no latency and requires a single synchronous write to become persistent [54, 55]. Using a single range-delete message also unlocks optimizations internal to the key/value store, such as freeing a dead leaf without first reading it from disk. Adding indirection on some directories (as in BetrFS 0.2) requires a recursive delete to scatter reads and writes throughout the keyspace and disk (Table 3).

Our contributions. This paper presents a B^ε-tree variant, called a ***lifted B^ε-tree***, that can efficiently rename a range of lexicographically ordered keys, unlocking the benefits of full-path indexing. We demonstrate the benefits of a lifted B^ε-tree in combination with full-path indexing in a new version of BetrFS, version 0.4, which achieves:

- fast updates of data and metadata,
- fast scans of the data and metadata in directory subtrees and fast scans of files,
- fast renames, and
- fast subtree operations, such as recursive deletes.

We introduce a new key/value primitive called ***range rename***. Range renames are the keyspace analogue of directory renames. Given two strings, p_1 and p_2, range rename replaces prefix p_1 with prefix p_2 in all keys that have p_1 as a prefix. Range rename is an atomic modification to a contiguous range of keys, and the values are unchanged. Our main technical innovation is an efficient implementation of range rename in a B^ε-tree. Specifically, we reduce the cost from the size of the subtree to the height of the subtree.

Using range rename, BetrFS 0.4 returns to a simple schema for mapping file system operations onto key/value operations; this in turn consolidates all placement decisions and locality optimizations in one place. The result is simpler code with less ancillary metadata to maintain, leading to better performance on a range of seemingly unrelated operations.

The technical insight behind efficient B^ε-tree range rename is a method for performing large renames by direct manipulation of the B^ε-tree. Zoning shows us that small key ranges can be deleted and reinserted cheaply. For large key ranges, range rename is implemented by slicing the tree at the source and destination. Once the source subtree is isolated, a pointer swing moves the renamed section of keyspace to its destination. The asymptotic cost of such tree surgery is proportional to the height, rather than the size, of the tree.

Once the B^ε-tree has its new structure, another challenge is efficiently changing the pivots and keys to their new values. In a standard B^ε-tree, each node stores the full path keys; thus, a straightforward implementation of range rename must rewrite the entire subtree.

We present a method that reduces the work of updating keys by removing the redundancy in prefixes shared by many keys. This approach is called ***key lifting*** (§5). A lifted B^ε-tree encodes its pivots and keys such that the values of these strings are defined by the path taken to reach the node containing the string. Using this approach, the number of paths that need to be modified in a range rename also changes from being proportional to the size of the subtree to the depth of the subtree.

Our evaluation shows improvement across a range of workloads. For instance, BetrFS 0.4 performs recursive greps 1.5x faster and random writes 1.2x faster than BetrFS 0.3, but renames are competitive with standard, indirection-based file systems. As an example of simplicity unlocked by full path indexing, BetrFS 0.4 implements recursive deletion with a single range delete, significantly out-performing other file systems.

2 Background

This section presents background on BetrFS, relevant to the proposed changes to support efficient keyspace mutations. Additional aspects of the design are covered elsewhere [7, 22, 23, 54, 55].

2.1 B^ε-Tree Overview

The B^ε-tree is a ***write-optimized*** B-tree variant that implements the standard key/value store interface: in-

sert, delete, point query, and predecessor and successor queries (i.e., range queries). By write-optimized, we mean the insertions and deletions in B^ε-trees are orders of magnitude faster than in a B-tree, while point queries are just as fast as in a B-tree. Furthermore, range queries and sequential insertions and deletions in B^ε-trees can run at near disk bandwidth.

Because insertions are much faster than queries, the common read-modify-write pattern can become bottlenecked on the read. Therefore, B^ε-trees provide an ***upsert*** that logically encodes, but lazily applies a read-modify-write of a key-value pair. Thus, upserts are as fast as inserts.

Like B-trees, B^ε-trees store key/value pairs in nodes, where they are sorted by key order. Also like B-trees, interior nodes store pointers to children, and ***pivot keys*** delimit the range of keys in each child.

The main distinction between B^ε-trees and B-trees is that interior B^ε-tree nodes are augmented to include ***message buffers***. A B^ε-tree models all changes (inserts, deletes, upserts) as ***messages***. Insertions, deletions, and upserts are implemented by inserting messages into the buffers, starting with the root node. A key technique behind write-optimization is that messages can accumulate in a buffer, and are flushed down the tree in larger batches, which amortize the costs of rewriting a node. Most notably, this batching can improve the costs of small, random writes by orders of magnitude.

Since messages lazily propagate down the tree, queries may require traversing the entire root-to-leaf search path, checking for relevant messages in each buffer along the way. The newest target value is returned (after applying pending upsert messages, which encode key/value pair modifications).

In practice, B^ε-trees are often configured with large nodes (typically $\geq$4MiB) and fanouts (typically $\leq$16) to improve performance. Large nodes mean updates are applied in large batches, but large nodes also mean that many contiguous key/value pairs are read per I/O. Thus, range queries can run a near disk bandwidth, with at most one random I/O per large node.

The B^ε-tree implementation in BetrFS supports both ***point*** and ***range*** messages; range messages were introduced in v0.2 [54]. A point message is addressed to a single key, whereas a range message is applied to a contiguous range of keys. Thus far, range messages have only been used for deleting a range of contiguous keys with a single message. In our experience, range deletes give useful information about the keyspace that is hard to infer from a series of point deletions, such as dropping obviated insert and upsert messages.

The B^ε-tree used in BetrFS supports transactions and crash consistency as follows. The B^ε-tree internally uses a logical timestamp for each message and MVCC to implement transactions. Pending messages can be thought of as a history of recent modifications, and, at any point in the history, one can construct a consistent view of the data. Crash consistency is ensured using logical logging, i.e., by logging the inserts, deletes, etc, performed on the tree. Internal operations, such as node splits, flushes, etc, are not logged. Nodes are written to disk using copy-on-write. At a periodic checkpoint (every 5 seconds), all dirty nodes are written to disk and the log can be trimmed. Any unreachable nodes are then garbage collected and reused. Crash recovery starts from the last checkpoint, replays the logical redo log, and garbage collects any unreachable nodes; as long as an operation is in the logical log, it will be recoverable.

2.2 BetrFS Overview

BetrFS translates VFS-level operations into B^ε-tree operations. Across versions, BetrFS has explored schema designs that map VFS-level operations onto B^ε-tree operations as efficiently as possible.

All versions of BetrFS use two B^ε-trees: one for file data and one for file system metadata. The B^ε-tree implementation supports transactions, which we use internally for operations that require more than one message. BetrFS does not expose transactions to applications, which introduce some more complex issues around system call semantics [25, 35, 39, 46].

In BetrFS 0.1, the metadata B^ε-tree maps a full path onto the typical contents of a `stat` structure, including owner, modification time, and permission bits. The data B^ε-tree maps keys of the form (p, i), where p is the full path to a file and i is a block number within that file, to 4KB file blocks. Paths are sorted in a variant of depth-first traversal order.

This full-path schema means that entire sub-trees of the directory hierarchy are stored in logically contiguous ranges of the key space. For instance, this design enabled BetrFS 0.1 to perform very fast recursive directory traversals (e.g. `find` or recursive `grep`).

Unfortunately, with this schema, file and directory renames do not map easily onto key/value store operations. In BetrFS 0.1, file and directory renames were implemented by copying the file or directory from its old location to the new location. As a result, renames were orders of magnitude slower than conventional file systems.

BetrFS 0.2 improved rename performance by replacing the full-path indexing schema of BetrFS 0.1 with a ***relative-path*** indexing schema [54, 55]. The goal of relative path indexing is to get the rename performance of inode-based file systems and the recursive-directory-traversal performance of a full-path indexing file system.

BetrFS 0.2 accomplishes this by partitioning the directory hierarchy into a collection of connected regions called ***zones***. Each zone has a single root file or directory

and, if the root of a zone is a directory, it may contain sub-directories of that directory. Each zone is given a zone ID (analogous to an inode number).

Relative-path indexing made renames on BetrFS 0.2 almost as fast as inode-based file systems and recursive-directory traversals almost as fast as BetrFS 0.1.

However, our experience has been that relative-path indexing introduces a number of overheads and precludes other opportunities for mapping file-system-level operations onto B^ε-tree operations. For instance, must be split and merged to keep all zones within a target size range. These overheads became a first-order performance issue, for example, the Tokubench benchmark results for BetrFS 0.2.

Furthermore, relative-path indexing also has bad worst-case performance. It is possible to construct arrangements of nested directories that will each reside in their own zone. Reading a file in the deepest directory will require reading one zone per directory (each with its own I/O). Such a pathological worst case is not possible with full-path indexing in a B^ε-tree, and an important design goal for BetrFS is keeping a reasonable bound on the worst cases.

Finally, zones break the clean mapping of directory subtrees onto contiguous ranges of the key space, preventing us from using range-messages to implement bulk operations on entire subtrees of the directory hierarchy. For example, with full-path indexing, we can use range-delete messages not only to delete files, but entire subtrees of the directory hierarchy. We could also use range messages to perform a variety of other operations on subtrees of the directory hierarchy, such as recursive `chmod`, `chown`, and timestamp updates.

The goal of this paper is to show that, by making rename a first-class key/value store operation, we can use full-path indexing to produce a simpler, more efficient, and more flexible system end-to-end.

3 Overview

The goal of this section is to explain the performance considerations behind our data structure design, and to provide a high-level overview of that design.

Our high-level strategy is to simply copy small files and directories in order to preserve locality—i.e., copying a few-byte file is no more expensive than updating a pointer. Once a file or directory becomes sufficiently large, copying the data becomes expensive and of diminishing value (i.e., the cost of indirection is amortized over more data). Thus, most of what follows is focused on efficient rename of *large* files and directories, large meaning at least as large as a B^ε-tree node.

Since we index file and directory data and metadata by full path, a file or directory rename translates into a prefix replacement on a *contiguous* range of keys. For example, if we rename directory `/tmp/draft` to `/home/paper/final`, then we want to find all keys in the B^ε-tree that begin with `/tmp/draft` and replace that prefix with `/home/paper/final`. This involves both updating the key itself, and updating its location in the B^ε-tree so that future searches can find it.

Since the affected keys form a contiguous range in the B^ε-tree, we can move the keys to their new (logical) home without moving them physically. Rather, we can make a small number of pointer updates and other changes to the tree. We call this step ***tree surgery***. We then need to update all the keys to contain their new prefix, a process we call ***batched key update***.

In summary, the algorithm has two high-level steps:

Tree Surgery. We identify a subtree of the B^ε-tree that includes all keys in the range to be renamed (Figure 1). Any ***fringe*** nodes (i.e., on the left and right extremes of the subtree), which contain both related and unrelated keys, are split into two nodes: one containing only affected keys and another containing only un-affected keys. The number of fringe nodes will be at most logarithmic in the size of the sub-tree. At the end of the process, we will have a subtree that contains only keys in the range being moved. We then change the pivot keys and pointers to move the subtree to its new parent.

Batched Key Updates. Once a subtree has been logically renamed, full-path keys in the subtree will still reflect the original key range. We propose a B^ε-tree modification to make these key updates efficient. Specifically, we modify the B^ε-tree to factor out common prefixes from keys in a node, similar to prefix-encoded compression. We call this transformation ***key lifting***. This transformation does not lose any information—the common prefix of keys in a node can be inferred from the pivot keys along the path from the root to the node by concatenating the longest common prefix of enclosing pivots along the path. As a result of key lifting, once we perform tree surgery to isolate the range of keys affected by a rename, the prefix to be replaced in each key will already be removed from every key in the sub-tree. Furthermore, since the omitted prefixes are inferred from the sub-tree's parent pivots, moving the sub-tree to its new parent implicitly replaces the old prefix with the new one. Thus a large subtree can be left untouched on disk during a range rename. In the worst case, only a logarithmic number of nodes on the fringe of the subtree will have keys that need to be updated.

Buffered Messages and Concurrency. Our range move operation must also handle any pending messages targeting the affected keys. These messages may be buffered in any node along a search path from the root to one of the affected keys. Our solution leverages the fact that messages have a logical timestamp and are ap-

plied in logical order. Thus, it is sufficient to ensure that pending messages for a to-be-renamed subtree must be flushed into the subtree before we begin the tree surgery for a range rename.

Note that most of the work in tree surgery involves node splits and merges, which are part of normal B^ϵ-tree operation. Thus the tree remains a "valid" B^ϵ-tree during this phase of the range move. Only the pointer swaps need to be serialized with other operations. Thus this approach does not present a concurrency bottleneck.

The following two sections explain tree surgery and lifting in more detail.

4 Tree Surgery

This section describes our approach to renaming a directory or large file via changes within the B^ϵ-tree, such that most of the data is not physically moved on disk. Files that are smaller than 4MiB reside in at most two leaves. We therefore move them by copying and perform tree surgery only on larger files and, for simplicity of the prototype, directories of any size.

For the sake of discussion, we assume that a rename is moving a source file over an existing destination file; the process would work similarly (but avoid some work) in the case where the destination file does not exist. Our implementation respects POSIX restrictions for directories (i.e., you cannot rename over a non-empty destination directory), but our technique could easily support different directory rename semantics. In the case of renaming over a file, where a rename implicitly deletes the destination file, we use transactions in the B^ϵ-tree to insert both a range delete of the destination and a range rename of the source; these messages are applied atomically.

This section also operates primarily at the B^ϵ-tree level, not the directory namespace. Unless otherwise noted, pointers are pointers within the B^ϵ-tree.

In renaming a file, the goal is to capture a range of contiguous keys and logically move these key/value pairs to a different point in the tree. For anything large enough to warrant using this rename approach, some B^ϵ-tree nodes will exclusively store messages or key/value pairs for the source or destination, and some may include unrelated messages or key/value pairs before and after in sort order, corresponding to the left and right in the tree.

An important abstraction for tree surgery is the ***Lowest Common Ancestor***, or ***(LCA)***, of two keys: the B^ϵ-tree node lowest in the tree on the search path for both keys (and hence including all keys in between). During a rename, the source and destination will each have an LCA, and they may have the same LCA.

The first step in tree surgery is to find the source LCA and destination LCA. In the process of identifying the LCAs, we also flush any pending messages for the source or destination key range, so that they are buffered at or below the corresponding LCAs.

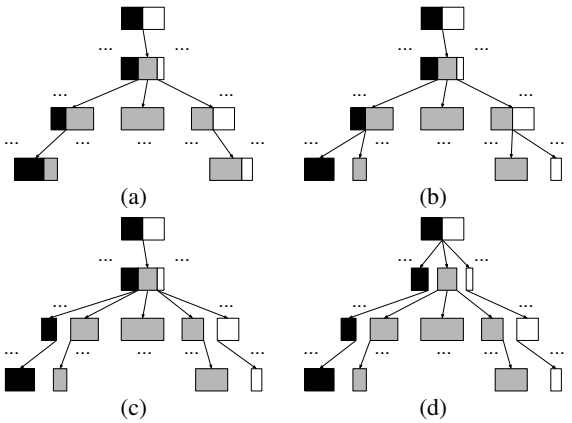


Figure 1: Slicing /gray between /black and /white.

Slicing. The second step is to slice out the source and destination from any shared nodes. The goal of slicing is to separate unrelated key-value pairs that are not being moved but are packed into the same B^ϵ-tree node as key-value pairs that are being moved. Slicing uses the same code used for standard B^ϵ-tree node splits, but slicing divides the node at the slicing key rather than picking a key in the middle of the node. As a result, slicing may result in nodes that temporarily violate constraints on target node size and fanout. However, these are performance, not correctness, constraints, so we can let queries continue concurrently, and we restore these invariants before completing the rename.

Figure 1 depicts slicing the sub-tree containing all gray keys from a tree with black, gray, and white keys. The top node is the parent of the LCA. Because messages have been flushed to the LCA, the parent of the LCA contains no messages related to gray. Slicing proceeds up the tree from the leaves, and only operates on the left and right fringe of the gray sub-tree. Essentially, each fringe node is split into two smaller B^ϵ-tree nodes (see steps b and c). All splits, as well as later transplanting and healing, happen when the nodes are pinned in memory. During surgery, they are dirtied and written at the next checkpoint. Eventually, the left and right edge of an exclusively-gray subtree (step d) is pinned, whereas interior, all-grey nodes may remain on disk.

Our implementation requires that the source and destination LCA be at the same height for the next step. Thus, if the LCAs are not at the same level of the tree, we slice up to an ancestor of the higher LCA. The goal of this choice is to maintain the invariant that all B^ϵ-tree leaves be at the same depth.

Transplant. Once the source and destination are both sliced, we then swap the pointers to each sub-tree in the respective parents of LCAs. We then insert a range-delete message at the source (which now points to a sub-

tree containing all the data in the file that used to reside at the destination of the rename). The B^ε-tree's builtin garbage collection will reclaim these nodes.

Healing. Our B^ε-tree implementation maintains the invariant that all internal nodes have between 4 and 16 children, which bounds the height of the tree. After the transplant completes, however, there may be a number of in-memory B^ε-tree nodes at the fringe around the source and destination that have fewer than 4 children.

We handle this situation by triggering a rebalancing within the tree. Specifically, if a node has only one child, the slicing process will merge it after completing the work of the rename.

Crash Consistency. In general, BetrFS ensures crash consistency by keeping a redo log of pending messages and applying messages to nodes copy-on-write. At periodic intervals, BetrFS ensures that there is a consistent checkpoint of the tree on disk. Crash recovery simply replays the redo log since the last checkpoint. Range rename works within this framework.

A range rename is *logically applied* and persisted as soon as the message is inserted into the root buffer and the redo log. If the system crashes after a range rename is logged, the recovery will see a prefix of the message history that includes the range rename, and it will be logically applied to any queries for the affected range.

Tree surgery occurs when a range rename message is flushed to a node that is likely an LCA. Until surgery completes, all fringe nodes, the LCAs, and the parents of LCAs are pinned in memory and dirtied. Upon completion, these nodes will be unpinned and written to disk, copy-on-write, no later than the next B^ε-tree checkpoint.

If the system crashes after tree surgery begins but before surgery completes, the recovery code will see a consistent checkpoint of the tree as it was before the tree surgery. The same is true if the system crashes after tree surgery but before the next checkpoint (as these post-surgery nodes will not be reachable from the checkpoint root). Because a B^ε-tree checkpoint flushes all dirty nodes, if the system crashes after a B^ε-tree checkpoint, all nodes affected by tree surgery will be on disk.

At the file system level, BetrFS has similar crash consistency semantics to metadata-only journaling in ext4. The B^ε-tree implementation itself implements full data journaling [54, 55], but BetrFS allows file writes to be buffered in the VFS, weakening this guarantee end-to-end. Specifically, file writes may be buffered in the VFS caches, and are only logged in the recovery journal once the VFS writes back a dirty page (e.g., upon an `fsync` or after a configurable period). Changes to the directory tree structure, such as a `rename` or `mkdir` are persisted to the log immediately. Thus, in the common pattern of writing to a temporary file and then renaming it, it is possible for the rename to appear in the log before the writes. In this situation and in the absence of a crash, the writes will eventually be logged with the correct, renamed key, as the in-memory inode will be up-to-date with the correct B^ε-tree key. If the system crashes, these writes can be lost; as with a metadata-journaled file system, the developer must issue an `fsync` before the `rename` to ensure the data is on disk.

Latency. A rename returns to the user once a log entry is in the journal and the root of the B^ε-trees are locked. At this point, the rename has been applied in the VFS to in-memory metadata, and as soon as the log is fsynced, the rename is durable.

We then hand off the rest of the rename work to two background threads to do the cutting and healing. The prototype in this paper only allows a backlog of one pending, large rename, since we believe that concurrent renames are relatively infrequent. The challenge in adding a rename work queue is ensuring consistency between the work queue and the state of the tree.

Atomicity and Transactions. The B^ε-tree in BetrFS implements multi-version concurrency control by augmenting messages with a logical timestamp. Messages updating a given key range are always applied in logical order. Multiple messages can share a timestamp, giving them transactional semantics.

To ensure atomicity for a range rename, we create an MVCC "hazard": read transactions "before" the rename must complete before the surgery can proceed. Tree nodes in BetrFS are locked with reader-writer locks. We write-lock tree nodes hand-over-hand, and left-to-right to identify the LCAs. Once the LCAs are locked, this serializes any new read or write transactions until the rename completes. The lock at the LCA creates a "barrier"—operations can complete "above" or "below" this lock in the tree, although the slicing will wait for concurrent transactions to complete before write-locking that node. Once the transplant completes, the write-locks on the parents above LCAs are released.

For simplicity, we also ensure that all messages in the affected key range(s) that logically occur before the range rename are flushed below the LCA before the range rename is applied. All messages that logically occur after the rename follow the new pointer path to the destination or source. This strategy ensures that, when each message is flushed and applied, it sees a point-in-time consistent view of the subtree.

Complexity. At most 4 slices are performed, each from the root to a leaf, dirtying nodes from the LCA along the slicing path. These nodes will need to be read, if not in cache, and written back to disk as part of the checkpointing process. Therefore the number of I/Os is at most proportional to the height of the B^ε-tree, which is logarithmic in the size of the tree.

5 Batched Key Updates

After tree-surgery completes, there will be a subtree where the keys are not coherent with the new location in the tree. As part of a rename, the prefixes of all keys in this subtree need to be updated. For example, suppose we execute `'mv /foo /bar'`. After surgery, any messages and key/value pairs for file `/foo/bas` will still have a key that starts with `/foo`. These keys need to be changed to begin with `/bar`. The particularly concerning case is when `/foo` is a very large subtree and has interior nodes that would otherwise be untouched by the tree surgery step; our goal is to leave these nodes untouched as part of rename, and thus reduce the cost of key changes from the size of the rename tree to the height of the rename tree.

We note that keys in our tree are highly redundant. Our solution reduces the work of changing keys by reducing the redundancy of how keys are encoded in the tree. Consider the prefix encoding for a sequence of strings. In this compression method, if two strings share a substantial longest common prefix (lcp), then that lcp is only stored once. We apply this idea to B$^\varepsilon$-trees. The lcp of all keys in a subtree is removed from the keys and stored in the subtree's parent node. We call this approach ***key lifting*** or simply ***lifting*** for short.

At a high level, our lifted B$^\varepsilon$-tree stores a node's common, lifted key prefix in the node's parent, alongside the parent's pointer to the child node. Child nodes only store differing key suffixes. This approach encodes the complete key in the path taken to reach a given node.

Lifting requires a schema-level invariant that keys with a common prefix are adjacent in the sort order. As a simple example, if one uses `memcmp` to compare keys (as BetrFS does), then lifting will be correct. This invariant ensures that, if there is a common prefix between any two pivot keys, all keys in that child will have the same prefix, which can be safely lifted. More formally:

Invariant 1 *Let T' be a subtree in a B$^\varepsilon$-tree with full-path indexing. Let p and q be the pivots that enclose T'. That is, if T' is not the first or last child of its parent, then p and q are the enclosing pivots in the parent of T'. If T' is the first child of its parent, then q is the first pivot and p is the left enclosing pivot of the parent of T'.*

Let s be the longest common prefix of p and q. Then all keys in T' begin with s.

Given this invariant, we can strip s from the beginning of every message or key/value pair in T', only storing the non-lifted suffix. Lifting is illustrated in Figure 2, where the common prefix in the first child is "/b/", which is removed from all keys in the node and its children (indicated with strikethrough text). The common prefix (indicated with purple) is stored in the parent. As one moves toward leaves, the common prefix typically becomes longer ("/b/4/" in Figure 2), and each level of the tree can lift the additional common prefix.

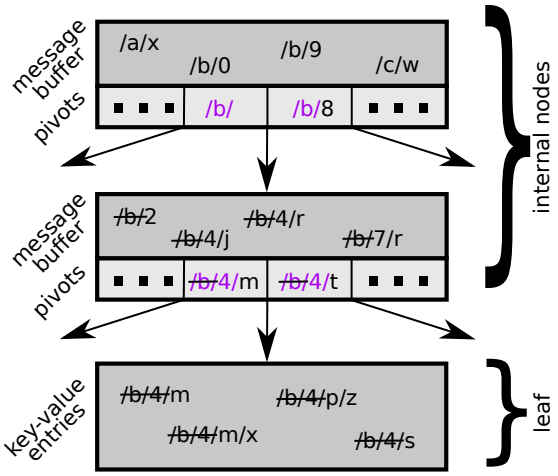


Figure 2: Example nodes in a lifted B$^\varepsilon$-tree. Since the middle node is bounded by two pivots with common prefix "/b/" (indicated by purple text), all keys in the middle node and its descendants must have this prefix in common. Thus this prefix can be omitted from all keys in the middle node (and all its descendants), as indicated by the strike-through text. Similarly, the bottom node (a leaf) is bounded by pivots with common prefix "/b/4/", so this prefix is omitted from all its keys.

Reads can reconstruct the full key by concatenating prefixes during a root-to-leaf traversal. In principle, one need not store the lifted prefix (s) in the tree, as it can be computed from the pivot keys. In our implementation, we do memoize the lifted prefix for efficiency.

As messages are flushed to a child, they are modified to remove the common prefix. Similarly, node splits and merges ensure that any common prefix between the pivot keys is lifted out. It is possible for all of the keys in T' to share a common prefix that is longer than s, but we only lift s because maintaining this amount of lifting hits a sweet spot: it is enough to guarantee fast key updates during renames, but it requires only local information at a parent and child during splits, merges, and insertions.

Lifting is completely transparent to the file system. From the file system's perspective, it is still indexing data with a key/value store that is keyed by full-path; the only difference from the file system's perspective is that the key/value store completes some operations faster.

Lifting and Renames. In the case of renames, lifting dramatically reduces the work to update keys. During a rename from a to b, we slice out a sub-tree containing exactly those keys that have a as a prefix. By the lifting invariant, the prefix a will be lifted out of the sub-tree, and the parent of the sub-tree will bound it between two pivots whose common prefix is a (or at least includes a—the pivots may have an even longer common pre-

fix). After we perform the pointer swing, the sub-tree will be bounded in its new parent by pivots that have b as a common prefix. Thus, by the lifting invariant, all future queries will interpret all the keys in the sub-tree has having b as a prefix. Thus, with lifting, the pointer swing implicitly performs the batch key-prefix replacement, completing the rename.

Complexity. During tree surgery, there is lifting work along all nodes that are sliced or merged. However, the number of such nodes is at most proportional to the height of the tree. Thus, the number of nodes that must be lifted after a rename is no more than the nodes that must be sliced during tree surgery, and proportional to the height of the tree.

6 Implementation Details

Simplifying key comparison. One small difference in the BetrFS 0.4 and BetrFS 0.3 key schemas is that BetrFS 0.4 adjusted the key format so that `memcmp` is sufficient for key comparison. We found that this change simplified the code, especially around lifting, and helped CPU utilization, as it is hard to compare bytes faster than a well-tuned `memcmp`.

Zone maintenance. A major source of overheads in BetrFS 0.3 is tracking metadata associated with zones. Each update involves updating significant in-memory bookkeeping; splitting and merging zones can also be a significant source of overhead (c.f., Figure 3). BetrFS 0.4 was able to delete zone maintenance code, consolidating this into the B^ε-tree's internal block management code.

Hard Links. BetrFS 0.4 does not support hard links. In future work, for large files, sharing sub-trees could also be used to implement hard links. For small files, zones could be reintroduced solely for hard links.

7 Evaluation

Our evaluation seeks to answer the following questions:

- (§7.1) Does full-path indexing in BetrFS 0.4 improve overall file system performance, aside from renames?
- (§7.2) Are rename costs acceptable in BetrFS 0.4?
- (§7.3) What other opportunities does full-path indexing in BetrFS 0.4 unlock?
- (§7.4) How does BetrFS 0.4 performance on application benchmarks compare to other file systems?

We compare BetrFS 0.4 with several file systems, including BetrFS 0.3 [14], Btrfs [42], ext4 [31], nilfs2 [34], XFS [47], and ZFS [8]. Each file system's block size is 4096 bytes. We use the versions of XFS, Btrfs, ext4 that are part of the 3.11.10 kernel, and ZFS 0.6.5.11, downloaded from `www.zfsonlinux.org`. We use default recommended file system settings unless otherwise noted. For ext4 (and BetrFS), we disabled lazy inode table and journal initialization, as these features accelerate file system creation but slow down some operations on a freshly-created file system; we believe this configuration yields more representative measurements of the file system in steady-state. Each experiment was run a minimum of 4 times. Error bars indicate minimum and maximum times over all runs. Similarly, error $\pm$ terms bound minimum and maximum times over all runs. Unless noted, all benchmarks are cold-cache tests and finish with a file-system sync. For BetrFS 0.3, we use the default zone size of 512 KiB.

In general, we expect BetrFS 0.3 to be the closest competitor to BetrFS 0.4, and focus on this comparison but include other file systems for context. Relative-path indexing is supposed to get most of the benefits of full-path indexing, with affordable renames; comparing BetrFS 0.4 with BetrFS 0.3 shows the cost of relative-path indexing and the benefit of full-path indexing.

All experimental results were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 500 GB, 7200 RPM SATA disk, with a 4096-byte block size. The system runs Ubuntu 14.04.5, 64-bit, with Linux kernel version 3.11.10. We boot from a USB stick with the root file system, isolating the file system under test to only the workload.

7.1 Non-Rename Microbenchmarks

Tokubench. The tokubench benchmark creates three million 200-byte files in a balanced directory tree, where no directory is allowed to have more than 128 children.

As Figure 3 shows, zone maintenance in BetrFS 0.3 causes a significant performance drop around 2 million files. This drop occurs all at once because, at that point in the benchmark, all the top-level directories are just under the zone size limit. As a result, the benchmark goes through a period where each new file causes its top-level directory to split into its own zone. If we continue the benchmark long enough, we would see this happen again when the second-level directories reach the zone-size limit. In experiments with very long runs of Tokubench, BetrFS 0.3 never recovers this performance.

With our new rename implementations, zone maintenance overheads are eliminated. As a result, BetrFS 0.4 has no sudden drop in performance. Only nilfs2 comes close to matching BetrFS 0.4 on this benchmark, in part because nilfs2 is a log-structured file system. BetrFS 0.4 has over 80$\times$ higher cumulative throughput than ext4 throughout the benchmark.

Recursive directory traversals. In these benchmarks, we run `find` and recursive `grep` on a copy of the Linux kernel 3.11.10 source tree. The times taken for these operations are given in Table 1. BetrFS 0.4 outperforms BetrFS 0.3 by about 5% on find and almost 30% on grep. In the case of grep, for instance, we found that roughly the same total number of bytes were read from disk in both

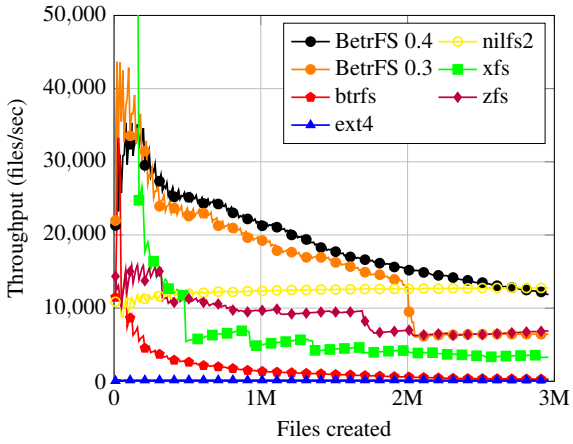


Figure 3: Cumulative file creation throughput during the Tokubench benchmark (higher is better). BetrFS 0.4 outperforms other file systems by orders of magnitude and avoids the performance drop that BetrFS 0.3 experience due to its zone-maintenance overhead.

File system	find (sec)	grep (sec)
BetrFS 0.4	0.233 ± 0.0	3.834 ± 0.2
BetrFS 0.3	0.247 ± 0.0	5.859 ± 0.1
btrfs	1.311 ± 0.1	8.068 ± 1.6
ext4	2.333 ± 0.1	42.526 ± 5.2
xfs	6.542 ± 0.4	58.040 ± 12.2
zfs	9.797 ± 0.9	346.904 ± 101.5
nilfs2	6.841 ± 0.1	8.399 ± 0.2

Table 1: Time to perform recursive directory traversals of the Linux 3.11.10 source tree (lower is better). BetrFS 0.4 is significantly faster than every other file system, demonstrating the locality benefits of full-path indexing.

versions of BetrFS, but that BetrFS 0.3 issued roughly 25% more I/O transactions. For this workload, we also saw higher disk utilization in BetrFS 0.4 (40 MB/s vs. 25 MB/s), with fewer worker threads needed to drive the I/O. Lifting also reduces the system time by 5% on grep, but the primary savings are on I/Os. In other words, this demonstrates the locality improvements of full-path indexing over relative-path indexing. BetrFS 0.4 is anywhere from 2 to almost 100 times faster than conventional file systems on these benchmarks.

Sequential IO. Figure 4 shows the throughput of sequential reads and writes of a 10GiB file (more than twice the size of the machine's RAM). All file systems measured, except ZFS, are above 100 MB/s, and the disk's raw read and write bandwidth is 132 MB/s.

Sequential reads in BetrFS 0.4 are essentially identical to those in BetrFS 0.3 and roughly competitive with other file systems. Both versions of BetrFS do not realize the full performance of the disk on sequential I/O, leaving up to 20% of the throughput compared to ext4 or Btrfs. This

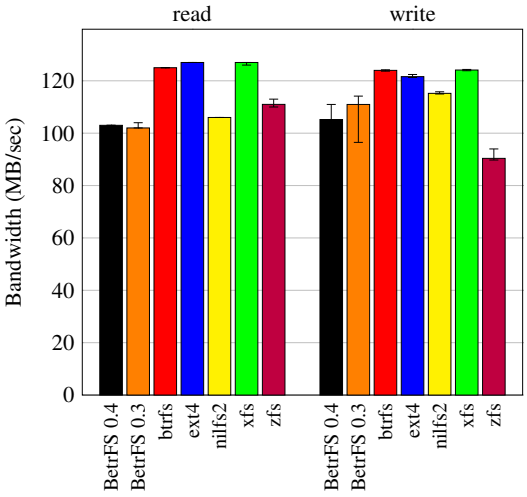


Figure 4: Sequential IO bandwidth (higher is better). BetrFS 0.4 performs sequential IO at over 100 MB/s but up to 19% slower than the fastest competitor. Lifting introduces some overheads on sequential writes.

is inherited from previous versions of BetrFS and does not appear to be significantly affected by range rename. Profiling indicates that there is not a single culprit for this loss but several cases where writeback of dirty blocks could be better tuned to keep the disk fully utilized. This issue has improved over time since version 0.1, but in small increments.

Writes in BetrFS 0.4 are about 5% slower than in BetrFS 0.3. Profiling indicates this is because node splitting incurs additional computational costs to re-lift a split child. We believe this can be addressed in future work by either better overlapping computation with I/O or integrating key compression with the on-disk layout, so that lifting a leaf involves less memory copying.

Random writes. Table 2 shows the execution time of a microbenchmark that issues 256K 4-byte overwrites at random offsets within a 10GiB file, followed by an `fsync`. This is 1 MiB of total data written, sized to run for at least several seconds on the fastest file system. BetrFS 0.4 performs small random writes approximately 400 to 650 times faster than conventional file systems and about 19% faster than BetrFS 0.3.

Summary. These benchmarks show that lifting and full-path indexing can improve performance over relative-path indexing for both reads and writes, from 5% up to 2×. The only case harmed is sequential writes. In short, lifting is generally more efficient than zone maintenance in BetrFS.

7.2 Rename Microbenchmarks

Rename performance as a function of file size. We evaluate rename performance by renaming files of different sizes and measuring the throughput. For each file

File system	random write (sec)	
BetrFS 0.4	4.9	± 0.3
BetrFS 0.3	5.9	± 0.1
btrfs	2147.5	± 7.4
ext4	2776.0	± 40.2
xfs	2835.7	± 7.9
zfs	3288.9	± 394.7
nilfs2	2013.1	± 19.1

Table 2: Time to perform 256K 4-byte random writes (1 MiB total writes, lower is better). BetrFS 0.4 is up to 600 times faster than other file systems on random writes.

size, we rename a file of this size 100 times within a directory and `fsync` the parent directory to ensure that the file is persisted on disk. We measure the average across 100 runs and report this as throughput, in Figure 5a.

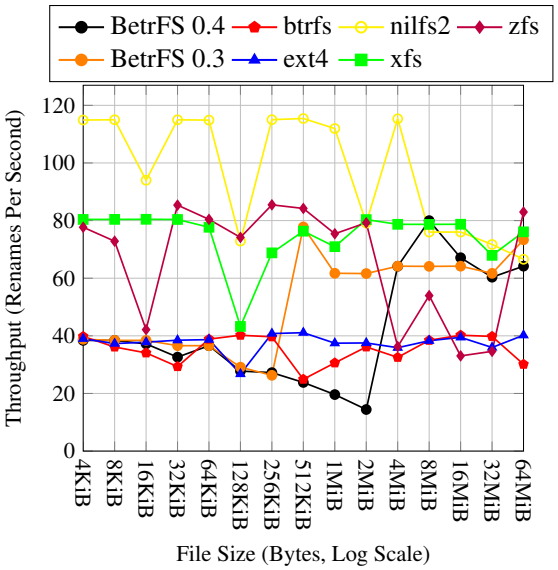
In both BetrFS 0.3 and BetrFS 0.4, there are two modes of operation. For smaller objects, both versions of BetrFS simply copy the data. At 512 KiB and 4 MiB, BetrFS 0.3 and BetrFS 0.4, respectively, switch modes—this is commensurate with the file matching the zone size limit and node size, respectively. For files above these sizes, both file systems see comparable throughput of simply doing a pointer swing.

More generally, the rename throughput of all of these file systems is somewhat noisy, but ranges from 30–120 renames per second, with nilfs2 being the fastest. Both variants of BetrFS are within this range, except when a rename approaches the node size in BetrFS 0.4.

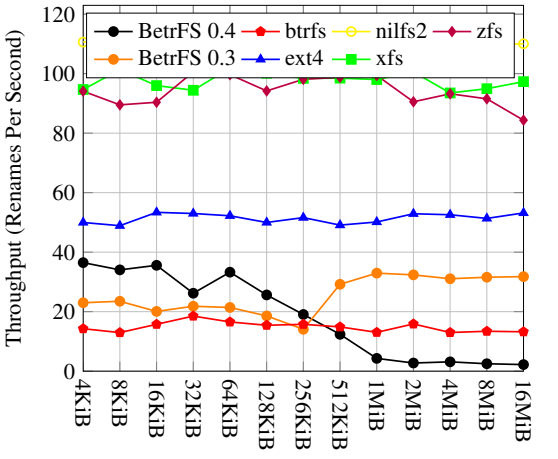
Figure 5b shows rename performance in a setup carefully designed to incur the worst-case tree surgery costs in BetrFS 0.4. In this experiment, we create two directories, each with 1000 files of the given size. The benchmark renames the interleaved files from the source directory to the destination directory, so that they are also interleaved with the files of the given size in the destination directory. Thus, when the interleaved files are 4MB or larger, every rename requires two slices at both the source and destination directories. We fsync after each rename.

Performance is roughly comparable to the previous experiment for small files. For large files, this experiment shows the worst-case costs of performing four slices. Further, all slices will operate on different leaves.

Although this benchmark demonstrates that rename performance has potential for improvement in some carefully constructed worst-case scenarios, the cost of renames in BetrFS 0.4 is nonetheless bounded to an average cost of 454ms. We also note that this line flattens, as the slicing overheads grow logarithmically in the size of the renamed file. In contrast, renames in BetrFS 0.1 were unboundedly expensive, easily getting into minutes; bounding this worst case is significant progress for



(a) Rename throughput as a function of file size. This experiment was performed in the base directory of an otherwise empty file system.



(b) Rename throughput as a function of file size. This experiment interleaves the renamed files with other files of the same size in both the source and destination directories.

Figure 5: Rename throughput

the design of full-path-indexed file systems.

7.3 Full-path performance opportunities

As a simple example of other opportunities for full-path indexing, consider deleting an entire directory (e.g., `rm -rf`). POSIX semantics require checking permission to delete all contents, bringing all associated metadata into memory. Other directory deletion semantics have been proposed. For example, HiStar allows an untrusted administrator to *delete* a user's directory but not *read* the contents [56].

We implemented a system call that uses range-delete messages to delete an entire directory sub-tree. This

File system	recursive delete (sec)	
BetrFS 0.4 (range delete)	0.053 ±	0.001
BetrFS 0.4	3.351 ±	0.5
BetrFS 0.3	2.711 ±	0.3
btrfs	2.762 ±	0.1
ext4	3.693 ±	2.2
xfs	7.971 ±	0.8
zfs	11.492 ±	0.1
nilfs2	9.472 ±	0.3

Table 3: Time to delete the Linux 3.11.10 source tree (lower is better). Full-path indexing in BetrFS 0.4 can remove a subtree in a single range delete, orders-of-magnitude faster than the recursive strategy of `rm -rf`.

system call therefore accomplishes the same goal as `rm -rf`, but it does not need to traverse the directory hierarchy or issue individual unlink/rmdir calls for each file and directory in the tree. The performance of this system call is compared to the performance of `rm -rf` on multiple file systems in Table 3. We delete the Linux 3.11.10 source tree using either our recursive-delete system call or by invoking `rm -rf`.

A recursive delete operation is orders of magnitude faster than a brute-force recursive delete on all file systems in this benchmark. This is admittedly an unfair benchmark, in that it foregoes POSIX semantics, but is meant to illustrate the *potential* of range updates in a full-path indexed system. With relative-path indexing, a range of keys cannot be deleted without first resolving the indirection underneath. With full-path indexing, one could directly apply a range delete to the directory, and garbage collect nodes that are rendered unreachable.

There is a regression in regular `rm -rf` performance for BetrFS 0.4, making it slower than Btrfs and BetrFS 0.3. A portion of this is attributable to additional overhead on un-lifting merged nodes (similar to the overheads added to sequential write for splitting); another portion seems to be exercising inefficiencies in flushing a large number of range messages, which is a relatively new feature in the BetrFS code base. We believe this can be mitigated with additional engineering. This experiment also illustrates how POSIX semantics, that require reads before writes, can sacrifice performance in a write-optimized storage system.

More generally, full-path indexing has the potential to improve many recursive directory operations, such as changing permissions or updating reference counts.

7.4 Macrobenchmark performance

Figure 6a shows the throughput of 4 threads on the Dovecot 2.2.13 mailserver. We initialize the mailserver with 10 folders, each contains 2500 messages, and use 4 threads, each performs 1000 operations with 50% reads and 50% updates (marks, moves, or deletes).

Figure 6b measures `rsync` performance. We copy the Linux 3.11.10 source tree from a source directory to a destination directory within the same partition and file system. With the `--in-place` option, `rsync` writes data directly to the destination file rather than creating a temporary file and updating via atomic rename.

Figure 6c reports the time to clone the Linux kernel source code repository [28] from a clone on the local system. The `git diff` workload reports the time to diff between the v4.14 and v4.07 Linux source tags.

Finally, Figure 6d reports the time to `tar` and un-`tar` the Linux 3.11.10 source code.

BetrFS 0.4 is either the fastest or a close second for 5 of the 7 application workloads. No other file system matches that breadth of performance.

BetrFS 0.4 represents a strict improvement over BetrFS 0.3 for these workloads. In particular, we attribute the improvement in the `rsync --in-place`, `git` and un-`tar` workloads to eliminating zone maintenance overheads. These results show that, although zoning represents a balance between full-path indexing and inode-style indirection, full path indexing can improve application workloads by 3-13% over zoning in BetrFS without incurring unreasonable rename costs.

8 Related Work

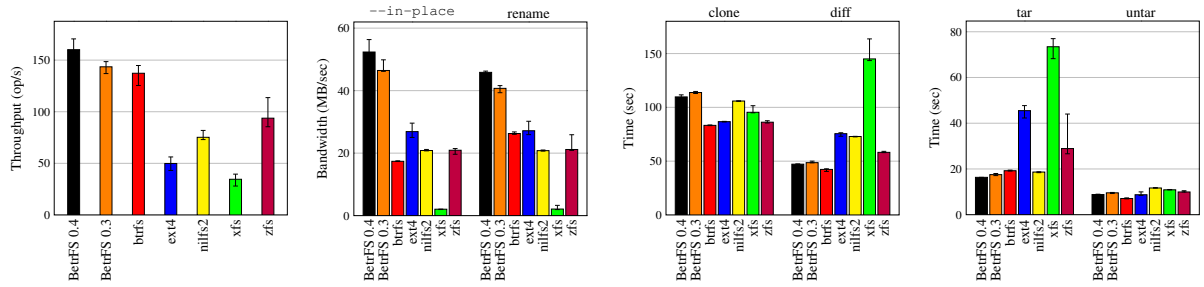
WODs. Write-Optimized Dictionaries, or WODs, including LSM-trees [36] and B^ε-trees [10], are widely used in key-value stores. For example, BigTable [12], Cassandra [26], LevelDB [20] and RocksDB [41] use LSM-trees; TokuDB [49] and Tucana [37] use B^ε-trees.

A number of projects have enhanced WODs, including in-memory component performance [4, 19, 44], write amplification [30, 53] and fragmentation [33]. Like the lifted B^ε-tree, the LSM-trie [53] also has a trie structure; the LSM-trie was applied to reducing write amplification during LSM compaction rather than fast key updates.

Several file systems are built on WODs through FUSE [17]. TableFS [40] puts metadata and small files in LevelDB and keeps larger files on ext4. KVFS [45] uses stitching to enhance sequential write performance on VT-trees, variants of LSM-trees. TokuFS [15], a precursor to BetrFS, uses full-path indexing on B^ε-trees, showing good performance for small writes and directory scans.

Trading writes for reads. IBM VSAM storage system, in the Key Sequenced Data Set (KSDS) configuration, can be thought of as an early key-value store using a B+ tree. One can think of using KSDS as a full-path indexed file system, optimized for queries. Unlike a POSIX file system, KSDS does not allow keys to be renamed, only deleted and reinserted [29].

In the database literature, a number of techniques have been developed that optimize for read-intensive work-



(a) IMAP server throughput. Higher is better.

(b) Rsync throughput. Higher is better.

(c) Git latency. Lower is better.

(d) Tar latency. Lower is better.

Figure 6: Application benchmarks. BetrFS 0.4 is the fastest file system, or essentially tied for fastest, in 4 out of the 7 benchmarks. No other file system offered comparable across-the-board performance. Furthermore, BetrFS 0.4's improvements over BetrFS in the in-place rsync, git clone, and untar benchmarks demonstrate that eliminating zone-maintenance overheads can benefit real application performance.

loads, but make schema changes or data writes more expensive [1–3, 13, 21, 24]. For instance, denormalization stores redundant copies of data in other tables, which can be used to reduce the costs of joins during query, but make updates more expensive. Similarly, materialized views of a database can store incremental results of queries, but keeping these views consistent with updates is more expensive.

Tree surgery. Most trees used in storage systems only modify or rebalance nodes as the result of insertions and deletions. Violent changes, such as tree surgery, are uncommon. Order Indexes [16] introduces relocation updates, which moves nodes in the tree, to support dynamic indexing. Ceph [52] performs dynamic subtree partitioning [51] on the directory tree to adaptively distribute metadata data to different metadata servers.

Hashing full paths. A number of systems store *metadata* in a hash table, keyed by full path, to lookup metadata in one I/O. The Direct Lookup File System (DLFS) maps file metadata to on-disk buckets by hashing full paths [27]. Hashing full paths creates two challenges: files in the same directory may be scattered across disk, harming locality, and DLFS directory renames require deep recursive copies of both data and metadata.

A number of distributed file systems have stored file metadata in a hash table, keyed by full path [18, 38, 48]. In a distributed system, using a hash table for metadata has the advantage of easy load balancing across nodes, as well as fast lookups. We note that the concerns of indexing *metadata* in a distributed file system are quite different from keeping logically contiguous *data* physically contiguous on disk. Some systems, such as the Google File System, also do not support common POSIX operations, such as listing a directory.

Tsai et al. [50] demonstrate that indexing the *in-memory* kernel directory cache by full paths can improve path lookup operations, such as `open`.

9 Conclusion

This paper presents a new on-disk indexing structure, the lifted B^ε-tree, which can leverage full-path indexing without incurring large rename overheads. Our prototype, BetrFS 0.4, is a nearly strict improvement over BetrFS 0.3. The main cases where BetrFS 0.4 does worse than BetrFS 0.3 are where node splitting and merging is on the critical path, and the extra computational costs of lifting harm overall performance. We believe these costs can be reduced in future work.

BetrFS 0.4 demonstrates the power of consolidating optimization effort into a single framework. A critical downside of zoning is that multiple, independent heuristics make independent placement decisions, leading to sub-optimal results and significant overheads. By using the keyspace to communicate information about application behavior, a single codebase can make decisions such as when to move data to recover locality, and when the cost of indirection can be amortized. In future work, we will continue exploring additional optimizations and functionality unlocked by full-path indexing.

Source code for BetrFS is available at `betrfs.org`.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ethan Miller for their insightful comments on earlier drafts of the work. Part of this work was done while Yuan was at Farmingdale State College of SUNY. This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, IIS 1251137, IIS-1247750, CCF 1617618, CCF 1439084, CCF-1314547, and by NIH grant NIH grant CA198952-01. The work was also supported by VMware, by EMC, and by NetApp Faculty Fellowships.

References

[1] AHMAD, Y., KENNEDY, O., KOCH, C., AND NIKOLIC, M. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB 5*, 10 (2012), 968–979.

[2] AHMAD, Y., AND KOCH, C. Dbtoaster: A SQL compiler for high-performance delta processing in main-memory databases. *PVLDB 2*, 2 (2009), 1566–1569.

[3] ARASU, A., BABCOCK, B., BABU, S., DATAR, M., ITO, K., NISHIZAWA, I., ROSENSTEIN, J., AND WIDOM, J. STREAM: the stanford stream data manager. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003* (2003), A. Y. Halevy, Z. G. Ives, and A. Doan, Eds., ACM, p. 665.

[4] BALMAU, O., GUERRAOUI, R., TRIGONAKIS, V., AND ZABLOTCHI, I. Flodb: Unlocking memory in persistent key-value stores. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017* (2017), G. Alonso, R. Bianchini, and M. Vukolic, Eds., ACM, pp. 80–94.

[5] BENDER, M. A., COLE, R., DEMAINE, E. D., AND FARACH-COLTON, M. Scanning and traversing: Maintaining data for traversals in a memory hierarchy. In *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings* (2002), R. H. Möhring and R. Raman, Eds., vol. 2461 of *Lecture Notes in Computer Science*, Springer, pp. 139–151.

[6] BENDER, M. A., FARACH-COLTON, M., FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL, B. C., AND NELSON, J. Cache-oblivious streaming b-trees. In *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007* (2007), P. B. Gibbons and C. Scheideler, Eds., ACM, pp. 81–92.

[7] BENDER, M. A., FARACH-COLTON, M., JANNEN, W., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., YUAN, J., AND ZHAN, Y. An introduction to B^e-trees and write-optimization. *:login; Magazine 40*, 5 (Oct 2015), 22–28.

[8] BONWICK, J. ZFS: the last word in file systems. `https://blogs.oracle.com/video/entry/zfs_the_last_word_in`, Sept. 2004.

[9] BRODAL, G. S., DEMAINE, E. D., FINEMAN, J. T., IACONO, J., LANGERMAN, S., AND MUNRO, J. I. Cache-oblivious dynamic dictionaries with update/query tradeoffs. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010* (2010), M. Charikar, Ed., SIAM, pp. 1448–1456.

[10] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.* (2003), ACM/SIAM, pp. 546–554.

[11] BUCHSBAUM, A. L., GOLDWASSER, M. H., VENKATASUBRAMANIAN, S., AND WESTBROOK, J. On external memory graph traversal. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA.* (2000), D. B. Shmoys, Ed., ACM/SIAM, pp. 859–860.

[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. *ACM Trans. Comput. Syst. 26*, 2 (2008), 4:1–4:26.

[13] CIPAR, J., GANGER, G. R., KEETON, K., III, C. B. M., SOULES, C. A. N., AND VEITCH, A. C. Lazybase: trading freshness for performance in a scalable database. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012* (2012), P. Felber, F. Bellosa, and H. Bos, Eds., ACM, pp. 169–182.

[14] CONWAY, A., BAKSHI, A., JIAO, Y., JANNEN, W., ZHAN, Y., YUAN, J., BENDER, M. A., JOHNSON, R., KUSZMAUL, B. C., PORTER, D. E., AND FARACH-COLTON, M. File systems fated for senescence? nonsense, says science! In *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017* (2017), G. Kuenning and C. A. Waldspurger, Eds., USENIX Association, pp. 45–58.

[15] ESMET, J., BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. The tokufs streaming file system. In *4th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'12, Boston, MA, USA, June 13-14, 2012* (2012), R. Rangaswami, Ed., USENIX Association.

[16] FINIS, J., BRUNEL, R., KEMPER, A., NEUMANN, T., MAY, N., AND FÄRBER, F. Indexing highly dynamic hierarchical data. *PVLDB 8*, 10 (2015), 986–997.

[17] File system in userspace. `http://fuse.sourceforge.net/`, Last Accessed May 16, 2015, 2015.

[18] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003* (2003), M. L. Scott and L. L. Peterson, Eds., ACM, pp. 29–43.

[19] GOLAN-GUETA, G., BORTNIKOV, E., HILLEL, E., AND KEIDAR, I. Scaling concurrent log-structured data stores. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015* (2015), L. Réveillère, T. Harris, and M. Herlihy, Eds., ACM, pp. 32:1–32:14.

[20] GOOGLE, INC. LevelDB: A fast and lightweight key/value database library by Google. `http://github.com/leveldb/`, Last Accessed May 16, 2015, 2015.

[21] HONG, M., DEMERS, A. J., GEHRKE, J., KOCH, C., RIEDEWALD, M., AND WHITE, W. M. Massively multi-query join processing in publish/subscribe systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007* (2007), C. Y. Chan, B. C. Ooi, and A. Zhou, Eds., ACM, pp. 761–772.

[22] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Betrfs: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015* (2015), J. Schindler and E. Zadok, Eds., USENIX Association, pp. 301–315.

[23] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Betrfs: Write-optimization in a kernel file system. *TOS 11*, 4 (2015), 18:1–18:29.

[24] JOHNSON, C., KEETON, K., III, C. B. M., SOULES, C. A. N., VEITCH, A. C., BACON, S., BATUNER, O., CONDOTTA, M., COUTINHO, H., DOYLE, P. J., EICHELBERGER, R., KIEHL, H., MAGALHAES, G. R., MCEVOY, J., NAGARAJAN, P., OSBORNE, P., SOUZA, J., SPARKES, A., SPITZER, M., TANDEL, S., THOMAS, L., AND ZANGARO, S. From research to practice: experiences engineering a production metadata database for a scale out file system. In *Proceedings of the 12th USENIX conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, February 17-20, 2014* (2014), B. Schroeder and E. Thereska, Eds., USENIX, pp. 191–198.

[25] KIM, S., LEE, M. Z., DUNN, A. M., HOFMANN, O. S., WANG, X., WITCHEL, E., AND PORTER, D. E. Improving server applications with system transactions. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 15–28.

[26] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *Operating Systems Review 44*, 2 (2010), 35–40.

[27] LENSING, P. H., CORTES, T., AND BRINKMANN, A. Direct lookup and hash-based metadata placement for local file systems. In *6th Annual International Systems and Storage Conference, SYSTOR '13, Haifa, Israel - June 30 - July 02, 2013* (2013), R. I. Kat, M. Baker, and S. Toledo, Eds., ACM, pp. 5:1–5:11.

[28] Linux kernel source tree. `https://github.com/torvalds/linux`.

[29] LOVELACE, M., DOVIDAUSKAS, J., SALLA, A., AND SOKAI, V. VSAM Demystified. `http://www.redbooks.ibm.com/redbooks/SG246105/wwhelp/wwhimpl/js/html/wwhelp.htm`, 2004.

[30] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.* (2016), pp. 133–148.

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

[32] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2*, 3 (1984), 181–197.

[33] MEI, F., CAO, Q., JIANG, H., AND TIAN, L. Lsm-tree managed storage for large-scale key-value store. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, September 24-27, 2017* (2017), pp. 142–156.

[34] NILFS: Continuous Snapshotting Filesystem. `https://nilfs.sourceforge.io/en/`.

[35] OLSON, J. Enhance your apps with file system transactions. *MSDN Magazine* (July 2007). http://msdn2.microsoft.com/en-us/magazine/cc163388.aspx.

[36] O'NEIL, P. E., CHENG, E., GAWLICK, D., AND O'NEIL, E. J. The log-structured merge-tree (lsm-tree). *Acta Inf. 33*, 4 (1996), 351–385.

[37] PAPAGIANNIS, A., SALOUSTROS, G., GONZÁLEZ-FÉREZ, P., AND BILAS, A. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.* (2016), A. Gulati and H. Weatherspoon, Eds., USENIX Association, pp. 537–550.

[38] PEERY, C., CUENCA-ACUNA, F. M., MARTIN, R. P., AND NGUYEN, T. D. Wayfinder: Navigating and sharing information in a decentralized world. In *Databases, Information Systems, and Peer-to-Peer Computing - Second International Workshop, DBISP2P 2004, Toronto, Canada, August 29-30, 2004, Revised Selected Papers* (2004), W. S. Ng, B. C. Ooi, A. M. Ouksel, and C. Sartori, Eds., vol. 3367 of *Lecture Notes in Computer Science*, Springer, pp. 200–214.

[39] PORTER, D. E., HOFMANN, O. S., ROSSBACH, C. J., BENN, A., AND WITCHEL, E. Operating systems transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009* (2009), J. N. Matthews and T. E. Anderson, Eds., ACM, pp. 161–176.

[40] REN, K., AND GIBSON, G. A. TABLEFS: enhancing metadata efficiency in the local file system. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013* (2013), A. Birrell and E. G. Sirer, Eds., USENIX Association, pp. 145–156.

[41] RocksDB. `rocksdb.org`, 2014. Viewed April 19, 2014.

[42] RODEH, O., BACIK, J., AND MASON, C. BTRFS: the linux b-tree filesystem. *TOS 9*, 3 (2013), 9:1–9:32.

[43] SEARS, R., CALLAGHAN, M., AND BREWER, E. A. *Rose*: compressed, log-structured replication. *PVLDB 1*, 1 (2008), 526–537.

[44] SEARS, R., AND RAMAKRISHNAN, R. blsm: a general purpose log structured merge tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012* (2012), K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds., ACM, pp. 217–228.

[45] SHETTY, P., SPILLANE, R. P., MALPANI, R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with vt-trees. In *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013* (2013), K. A. Smith and Y. Zhou, Eds., USENIX, pp. 17–30.

[46] SPILLANE, R. P., GAIKWAD, S., CHINNI, M., ZADOK, E., AND WRIGHT, C. P. Enabling transactional file access via lightweight kernel extensions. In *7th USENIX Conference on File and Storage Technologies, February 24-27, 2009, San Francisco, CA, USA. Proceedings* (2009), M. I. Seltzer and R. Wheeler, Eds., USENIX, pp. 29–42.

[47] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996* (1996), USENIX Association, pp. 1–14.

[48] THOMSON, A., AND ABADI, D. J. Calvinfs: Consistent WAN replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015* (2015), pp. 1–14.

[49] TOKUTEK, INC. TokuDB v6.5 for MySQL and MariaDB. `http://www.tokutek.com/products/tokudb-for-mysql/`, 2013. See `https://web.archive.org/web/20121011120047/http://www.tokutek.com/products/tokudb-for-mysql/`.

[50] TSAI, C., ZHAN, Y., REDDY, J., JIAO, Y., ZHANG, T., AND PORTER, D. E. How to get more value from your file system directory cache. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), E. L. Miller and S. Hand, Eds., ACM, pp. 441–456.

[51] WEIL, S., POLLACK, K., BRANDT, S. A., AND MILLER, E. L. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)* (Nov. 2004).

[52] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 307–320.

[53] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA, USA, July 8–10 2015), pp. 71–82.

[54] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Optimizing every operation in a write-optimized file system. In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.* (2016), A. D. Brown and F. I. Popovici, Eds., USENIX Association, pp. 1–14.

[55] YUAN, J., ZHAN, Y., JANNEN, W., PANDEY, P., AKSHINTALA, A., CHANDNANI, K., DEO, P., KASHEFF, Z., WALSH, L., BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Writes wrought right, and other adventures in file system optimization. *TOS 13*, 1 (2017), 3:1–3:26.

[56] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA* (2006), B. N. Bershad and J. C. Mogul, Eds., USENIX Association, pp. 263–278.

Clay Codes: Moulding MDS Codes to Yield an MSR Code

Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini,
Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar
Indian Institute of Science, Bangalore

Alexander Barg, Min Ye
University of Maryland

Srinivasan Narayanamurthy, Syed Hussain, Siddhartha Nandi
NetApp ATG, Bangalore

Abstract

With increase in scale, the number of node failures in a data center increases sharply. To ensure availability of data, failure-tolerance schemes such as Reed-Solomon (RS) or more generally, Maximum Distance Separable (MDS) erasure codes are used. However, while MDS codes offer minimum storage overhead for a given amount of failure tolerance, they do not meet other practical needs of today's data centers. Although modern codes such as Minimum Storage Regenerating (MSR) codes are designed to meet these practical needs, they are available only in highly-constrained theoretical constructions, that are not sufficiently mature enough for practical implementation. We present *Clay codes* that extract the best from both worlds. Clay (short for Coupled-Layer) codes are MSR codes that offer a simplified construction for decoding/repair by using pairwise coupling across multiple stacked layers of any single MDS code.

In addition, Clay codes provide the first practical implementation of an MSR code that offers (a) low storage overhead, (b) simultaneous optimality in terms of three key parameters: repair bandwidth, sub-packetization level and disk I/O, (c) uniform repair performance of data and parity nodes and (d) support for both single and multiple-node repairs, while permitting faster and more efficient repair.

While all MSR codes are vector codes, none of the distributed storage systems support vector codes. We have modified Ceph to support any vector code, and our contribution is now a part of Ceph's master codebase. We have implemented Clay codes, and integrated it as a plugin to Ceph. Six example Clay codes were evaluated on a cluster of Amazon EC2 instances and code parameters were carefully chosen to match known erasure-code deployments in practice. A particular example code, with storage overhead 1.25x, is shown to reduce repair network traffic by a factor of 2.9 in comparison with RS codes and similar reductions are obtained for both repair time and disk read.

1 Introduction

The number of failures in storage subsystems increase as data centers scale [11] [17] [29]. In order to ensure data availability and durability, failure-tolerant solutions such as replication and erasure codes are used. It is important for these solutions to be highly efficient so that they incur low cost in terms of their utilization of storage, computing and network resources. This additional cost is considered an overhead, as the redundancy introduced for failure tolerance does not aid the performance of the application utilizing the data.

In order to be failure tolerant, data centers have increasingly started to adopt erasure codes in place of replication. A class of erasure codes known as maximum distance separable (MDS) codes offer the same level of failure tolerance as replication codes with minimal storage overhead. For example, Facebook [19] reported reduced storage overhead of 1.4x by using Reed-Solomon (RS) codes, a popular class of MDS codes, as opposed to the storage overhead of 3x incurred in triple replication [13]. The disadvantage of the traditional MDS codes is their high repair cost. In case of replication, when a node or storage subsystem fails, an exact copy of the lost data can be copied from surviving nodes. However, in case of erasure codes, dependent data that is more voluminous in comparison with the lost data, is copied from surviving nodes and the lost data is then computed by a repair node, which results in a higher repair cost when compared to replication. This leads to increased repair bandwidth and repair time.

A class of erasure codes, termed as minimum storage regenerating (MSR) codes, offer all the advantages of MDS codes but require lesser repair bandwidth. Until recently, MSR codes lacked several key desirable properties that are important for practical systems. For example, they were computationally more complex [14], or demonstrated non-uniform repair characteristics for different types of node failures [18], or were able to recover

from only a limited (one or two) number of failures [20], or they lacked constructions of common erasure code configurations [24], [20]. The first theoretical construction that offered all the desirable properties of an MSR code was presented by Ye and Barg [35].

This paper presents Clay codes that extend the theoretical construction presented in [35], with practical considerations. Clay codes are constructed by placing any MDS code in multiple layers and performing pair-wise coupling across layers. Such a construction offers efficient repair with optimal repair bandwidth, causing Clay codes to fall in the MSR arena.

We implement Clay codes and make it available as open-source under LGPL. We also integrate Clay codes as a plugin with Ceph, a distributed object storage system. Ceph supports scalar erasure codes such as RS codes. However, it does not support vector codes. We modified Ceph to support any vector code, and our contribution is now included in Ceph's master codebase [4].

In erasure coding terminology, scalar codes require block-granular repair data, while vector codes can work at the sub-block granularity for repair. In Ceph, the equivalent of an erasure-coded block is one chunk of object. By this, we mean that Ceph supports chunk-granular repair data, while our contribution extended it to sub-chunk granularity. To the best of our knowledge, after our contribution, Ceph has become the first distributed storage system to support vector codes. Also, if Clay codes become part of Ceph's codebase, this will be the first-ever implementation of an MSR code that provides all desirable practical properties, and which is integrated to a distributed storage system.

Our contributions include (a) the construction of Clay codes as explained in Section 3, (b) the modification made to Ceph in order to support any vector code, explained in Section 4, and (c) the integration of Clay codes as a plugin to Ceph, explained in Section 4. We conducted experiments to compare the performance of Clay codes with RS codes available in Ceph and the results are presented in Section 5. One of the example Clay codes that we evaluated, which has a storage overhead of 1.25x, was able to bring down the repair network traffic by a factor of 2.9 when compared with the RS code of same parameters. Similar reductions were also obtained for disk read and repair time.

2 Background and Preliminaries

Erasure Code Erasure codes are an alternative to replication for ensuring failure tolerance in data storage. In an $[n,k]$ erasure-coded system, data pertaining to an object is first divided into k data chunks and then encoded to obtain $m = n - k$ parity chunks. When we do not wish to distinguish between a data or parity chunk, we will simply refer to the chunk as a *coded chunk*. The collection of n coded chunks obtained after encoding are stored in n distinct nodes. Here, by *node*, we mean an independent failure domain such as a disk or a storage node of a distributed storage system (DSS). The storage efficiency of an erasure code is measured by *storage overhead* defined as the ratio of the number of coded chunks n to the number of data chunks k. Every erasure code has an underlying finite field over which computations are performed. For the sake of simplicity, we assume here that the field is of size 2^8 and hence each element of the finite field can be represented by a byte[1]. It is convenient to differentiate at this point, between *scalar* and *vector codes*.

Scalar Codes Let each data chunk be comprised of L bytes. In the case of a *scalar* code, one byte from each of the k data chunks is picked and the k bytes are linearly combined in m different ways, to obtain m parity bytes. The resultant set of $n = k + m$ bytes so obtained is called a *codeword*. This operation is repeated in parallel for all the L bytes in a data chunk to obtain L codewords. This operation will also result in the creation of m parity chunks, each composed of L bytes (see Fig. 1). As mentioned above, every coded chunk is stored on a different node.

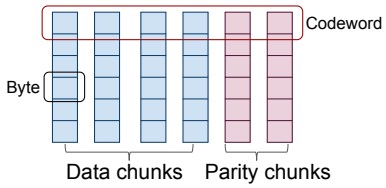


Figure 1: A pictorial representation of a scalar code. The $L = 6$ horizontal layers are the codewords and the $n = 6$ vertical columns, the chunks, with the first $k = 4$ chunks corresponding to data chunks and the last $(n-k) = 2$ chunks, the parity chunks. Each unit (tiny rectangle) in the figure corresponds to a single byte.

Vector Codes The difference in the case of vector codes is that here, one works with ordered collections of $\alpha \geq 1$ bytes at a time. For convenience, we will refer to such an ordered collection of α bytes as a *superbyte*. In the encoding process, a superbyte from each of the k data chunks is picked and the k superbytes are then linearly combined in m different ways, to obtain m parity superbytes. The resultant set of $n = k + m$ superbytes is called a (vector) *codeword*. This operation is repeated in parallel for all the $N = \frac{L}{\alpha}$ superbytes in a data chunk to obtain N codewords. Figure 2 shows a simple example where each superbyte consists of just two bytes.

The number α of bytes within a superbyte is termed the sub-packetization level of the code. Scalar codes

[1]The codes described in this paper can however, be constructed over a finite field whose size is significantly smaller, and approximately equal to the parameter n. Apart from simplicity, we use the word byte here since the finite field of size 2^8 is a popular choice in practice.

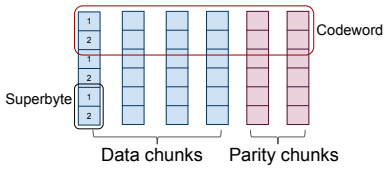


Figure 2: A pictorial representation of a vector code where each superbyte consists of 2 bytes. The picture shows $N = 3$ codewords. A single chunk, either data or parity, stores 3 superbytes, each corresponding to a different codeword.

such as RS codes can be regarded as having sub-packetization level $\alpha = 1$. Seen differently, one could view a vector code as replacing α scalar codewords with a single vector codeword. The advantage of vector codes is that repair of a coded chunk in a failed node can potentially be accomplished by accessing only a subset of the α bytes within the superbyte, present in each of the remaining coded chunks, corresponding to the same codeword. This reduces network traffic arising from node repair.

Sub-chunking through Interleaving In Fig. 2, we have shown the α bytes associated to a superbyte as being stored contiguously. When the sub-packetization level α is large, given that operations involving multiple codewords are carried out in parallel, it is advantageous, from an ease-of-memory-access viewpoint, to interleave the bytes so that the corresponding bytes across different codewords are stored contiguously as shown in Fig. 3. This is particularly true, when the number N of superbytes within a chunk is large, for example, when $L = 8KB$ and $\alpha = 2$, contiguous access to $N = 4K$ bytes is possible. With interleaving, each data chunk is partitioned into α subsets, which we shall refer to as sub-chunks. Thus each sub-chunk within a node, holds one byte from each of the N codewords stored in the node.

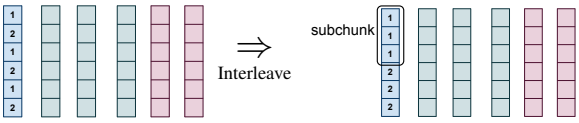


Figure 3: This figure shows the interleaving of the corresponding bytes within a superbyte across codewords, for the particularly simple case of two bytes within a superbyte. This results in a partitioning of the data chunk into sub-chunks and can lead to improved-memory-access performance.

MDS Codes The sub-class of (n,k) erasure codes, either scalar or vector, having the property that they can recover from the failure of any $(n-k)$ nodes are called MDS codes. For a fixed k, these codes have the smallest storage overhead $\frac{n}{k}$ among any of the erasure codes that can recover from a failure of a fixed number of $n-k$ nodes. Examples include RS, Row-Diagonal Parity [9] and EVENODD [7] codes, see [5] for additional examples. Facebook data centers [28] have employed an $(14,10)$ RS code in their data warehouse cluster.

Node Repair The need for node repair in a distributed storage system can arise either because a particular hardware component has failed, is undergoing maintenance, is being rebooted or else, is simply busy serving other simultaneous requests for data. A substantial amount of network traffic is generated on account of node-repair operations. An example cited in [28], is one of a Facebook data-warehouse, that stores multiple petabytes of data, where the median amount of data transferred through top-of-rack switches for the purposes of node repair, is in excess of 0.2 petabytes per day. The traffic arising from node-repair requests, eats into the bandwidth available to serve user requests for data. The time taken for node repair also directly affects system availability. Thus there is strong interest in coding schemes that minimize the amount of data transfer across the network, and the time taken to repair a failed node. Under the conventional approach to repairing an RS code for instance, one would have to download k times the amount of data as is stored in a failed node to restore the failed node, which quite clearly, is inefficient.

MSR Codes MSR codes [10] are a sub-class of vector MDS codes that have the smallest possible repair bandwidth. To restore a failed node containing α bytes in an (n,k) MSR code, the code first contacts an arbitrarily-chosen subset of d helper nodes, where d is a design parameter that can take on values ranging from k to $(n-1)$. It then downloads $\beta = \frac{\alpha}{d-k+1}$ bytes from each helper node, and restores the failed node using the helper data. The total amount $d\beta$ of bytes downloaded is typically much smaller than the total amount $k\alpha$ bytes of data stored in the k nodes. Here α is the sub-packetization level of an MSR code. The total number $d\beta$ of bytes downloaded for node repair, is called the *repair bandwidth*. Let us define the *normalized* repair bandwidth to be the quantity $\frac{d\beta}{k\alpha} = \frac{d}{k(d-k+1)}$. The normalization by $k\alpha$ can be motivated by viewing a single MSR codeword having sub-packetization level α as a replacement for α scalar RS codewords. The download bandwidth under the conventional repair of α scalar RS codes equals $k\alpha$ bytes, corresponding to a normalized repair bandwidth of 1. For the particular case $d = (n-1)$, the normalized value equals $\frac{n-1}{k(n-k)}$. It follows that the larger the number $(n-k)$ of parity chunks, the greater the reduction in repair traffic. We will also use the parameter $M = k\alpha$ to denote the total number of databytes contained in an MSR codeword. Thus an MSR code has associated parameter set given by $\{(n,k),d,(\alpha,\beta),M\}$ with $\beta = \frac{\alpha}{d-k+1}$ and $M = k\alpha$.

Additional Desired Attributes: Over and above the low repair-bandwidth and low storage-overhead attributes of MSR codes, there are some additional properties that one would like a code to have. These include (a) uniform-

Code	Storage O/h	Failure Tolerance	All-Node Optimal Repair	Disk Read Optimal	Repair-bandwidth Optimal	α	Order of GF	Implemented Distributed System
RS	Low	$n-k$	No	No	No	1	Low	HDFS, Ceph, Swift, etc.
PM-RBT [24]	High	$n-k$	Yes	Yes	Yes	Linear	Low	Own system
Butterfly [20]	Low	2	Yes	No	Yes	Exponential	Low	HDFS, Ceph
HashTag [18]	Low	$n-k$	No	No	Yes	Polynomial	High	HDFS
Clay Code	Low	$n-k$	Yes	Yes	Yes	Polynomial	Low	Ceph

Table 1: Detailed comparison of Clay codes with RS and other practical MSR codes. Here, the scaling of α is with respect to n for a fixed storage overhead (n/k).

repair capability, i.e., the ability to repair data and parity nodes with the same low repair bandwidth, (b) minimal disk read, meaning that the amount of data read from disk for node repair in a helper node is the same as the amount of data transferred over the network from the helper node and (c) low value of sub-packetization parameter α, and (d) a small size of underlying finite field over which the code is constructed. In MSR codes that possess the disk read optimal property, both network traffic and number of disk reads during node repair are simultaneously minimized and are the same.

2.1 Related Work

The problem of efficient node repair has been studied for some time and several solutions have been proposed. Locally repairable codes such as the Windows Azure Code [15] and Xorbas [28] trade the MDS property to allow efficient node-repair by accessing a smaller number of helper nodes. The piggy-backed RS codes introduced in [26] achieve reductions in network traffic while retaining the MDS property but they do not achieve the savings that are possible with an MSR code.

Though there are multiple implementations of MSR codes, these are lacking in one or the other of the desired attributes (see Table 1). In [8], the authors present 2- parity FMSR codes, that allow efficient repair, but reconstruct a function of the data that is not necessarily same as the failed node data. This demands an additional decoding operation to be performed to retrieve original data. In [24], the authors implement a modified product-matrix MSR construction [27]. Although the code displays optimal disk I/O performance, the storage overhead is on the higher side and of the form $(2-\frac{1}{k})$. In [20], the authors implement an MSR code known as the Butterfly code and experimentally validate the theoretically-proven benefits of reduced data download for node repair. However, the Butterfly code is limited to $(n-k)=m=2$ and has large value of sub-packetization 2^{k-1}. The restriction to small values of parameter m limits the efficiency of repair, as the normalized repair bandwidth can be no smaller than $\frac{1}{2}$. In [18], the authors propose a class of MDS array codes named as HashTag codes with $\alpha \leq (n-k)^{k/n-k}$ that permit flexibility in choice of α at the expense of repair bandwidth. However, the code supports efficient repair only for systematic nodes, requires computations at helper nodes, and involves operations in a large finite-field. The authors have presented an evaluation of HashTag codes in Hadoop.

In a parallel line of work, many theoretical constructions of MSR codes are proposed in literature. The product-matrix MSR codes proposed in [27] operate with very low sub-packetization and small finite-field size, however require a large storage overhead. In a second notable construction known as zig-zag codes [30], the authors present the first theoretical construction of low-storage-overhead MSR codes for every n,k, when $d=(n-1)$. The construction of zig-zag code is non-explicit in the sense that the finite-field coefficients determining the parities have to be found by computer search. Thus, despite the many theoretical constructions and a smaller number of practical implementations, the search for an MSR code having all of the desirable properties described above and its practical evaluation continued to remain elusive. The recent theoretical results of Ye and Barg [35] have resulted in an altered situation. In this work, the authors provide a construction that permits storage overhead as close to 1 as desired, sub-packetization level close to the minimum possible, finite field size no larger than n, optimal disk I/O, and all-node optimal repair.Clay codes offer a practical perspective and an implementation of the Ye-Barg theoretical construction, along with several additional attributes. In other words, Clay codes possess all of the desirable properties mentioned above, and also offer several additional advantages compared to the Ye-Barg code.

2.2 Refinements over Ye-Barg Code

The presentation of the Clay code here is from a coupled-layer perspective that leads directly to implementation, whereas the description in [35] is primarily in terms of parity-check matrices. For example, using the coupled-layer viewpoint, both data decoding (by which we mean recovery from a maximum of $(n-k)$ erasures) as well as node-repair algorithms can be described in terms of

two simple operations: (a) decoding of the scalar MDS code, and (b) an elementary linear transformation between pairs of bytes (see Section 3). While this coupled-layer view-point was implicit in the Ye-Barg paper [35], we make it explicit here.

In addition, Clay codes can be constructed using any scalar MDS code as building blocks, while Ye-Barg code is based only on Vandermonde-RS codes. Therefore, scalar MDS codes that have been time-tested, and best suited for a given application or workload need not be modified in order to make the switch to MSR codes. By using Clay codes, these applications can use the same MDS code in a coupled-layer architecture and get the added benefits of MSR codes. The third important distinction is that, in [35], only the single node-failure case is discussed. In the case of Clay codes, we have come up with a generic algorithm to repair multiple failures, that has allowed us to repair many instances of multiple node repair with reduced repair bandwidth. Our refinements over Ye-Barg code primarily aiming at its practical realization precede certain theoretical developments that are to come later. In a recent work [6], it is proved that the sub-packetization of Clay codes is the minimum possible for any disk-read-optimal MSR code. In [31], authors propose a permuatation-based transformation that converts a non-binary (n,k) MDS code to another MDS code permitting efficient repair of a set of $(n-k)$ nodes, at the cost of increasing the sub-packetization $(n-k)$ times. An MSR code obtained by repeated application of the transformation results in the same sub-packetization as that of the Ye-Barg code.

3 Construction of the Clay Code

Single Codeword Description In Section 2, we noted that each node stores a data chunk and that a data chunk is comprised of L bytes from N codewords. In the present section we will restrict our attention to the case of a single codeword, i.e., to the case when $N = 1$, $L = \alpha$.

Parameters of Clay Codes Evaluated Table 2 lists the parameters of the Clay codes evaluated here. As can be seen, the normalized repair bandwidth can be made much smaller by increasing the value of $(d-k+1)$. For example, the normalized repair bandwidth for a $(20,16)$ code equals 0.297, meaning that the repair bandwidth of a Clay code, is less than 30% of the corresponding value for $\alpha = 1024$ layers of a $(20,16)$ RS code.

Explaining Through Example We will describe the Clay code via an example code having parameters: $\{(n=4,k=2),\ d=3,(\alpha=4,\beta=2),\ M=8\}$. The codeword is stored across $n=4$ nodes of which $k=2$ are data nodes and $n-k=2$ are parity nodes. Each node stores a superbyte made up of $\alpha=4$ bytes. The code has storage overhead $\frac{n\alpha}{k\alpha} = \frac{n}{k} = 2$ which is the ratio of the total number $n\alpha = 16$ of bytes stored to the number $M = k\alpha = 8$ of data bytes. During repair of a failed node, $\beta = 2$ bytes of data are downloaded from each of the $d = 3$ helper nodes, resulting in a normalized repair bandwidth of $\frac{d\beta}{k\alpha} = \frac{d}{k(d-k+1)} = 0.75$.

(n,k)	d	(α,β)	$(d\beta)/(k\alpha)$
(6,4)	5	(8,4)	0.625
(12,9)	11	(81,27)	0.407
(14,10)	13	(256,64)	0.325
(14,10)	12	(243,81)	0.4
(14,10)	11	(128,64)	0.55
(20,16)	19	(1024,256)	0.297

Table 2: Parameters of the Clay codes evaluated here.

Starting Point: A $(4,2)$ Scalar RS Code We begin our description of the Clay code with a simple, distributed data storage setup composed of 4 nodes, where the nodes are indexed by (x,y) coordinates: $\{(x,y) \mid (x,y) \in J\},\ J = \{(0,0),(1,0),(0,1),(1,1)\}\}$. Let us assume that a $(4,2)$ RS code $\mathscr{M}$ is used to encode and store data on these 4 nodes. We assume that nodes $(0,0),(1,0)$ store data, nodes $(0,1),(1,1)$ store parity. Two nodes are said to be in same y-section, if they have the same y-coordinate.

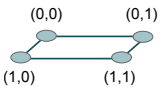


Figure 4: The $(4,2)$ MDS code $\mathscr{M}$.

The Uncoupled Code Next, consider storing on the same 4 nodes, 4 codewords drawn from the same RS code $\mathscr{M}$. Thus each node now stores 4 bytes, each associated to a different codeword. We will use the parameter $z \in \{0,1,2,3\}$ to index the 4 codewords. Together these 4 codewords form the *uncoupled* code $\mathscr{U}$, whose bytes are denoted by $\{U(x,y,z) \mid (x,y) \in J, z \in \{0,1,2,3\}\}$. These 16 bytes can be viewed as being stored in a data cube composed of 4 horizontal layers (or planes), with 4 bytes to a layer (Fig. 5). The data cube can also be viewed as being composed of 4 (vertical) columns, each column composed of 4 cylinders. Each column stores a superbyte while each of the 4 cylinders within a column stores a single byte.

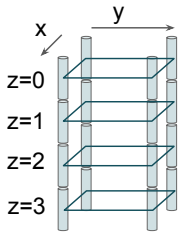


Figure 5: The uncoupled code $\mathscr{U}$.

It can be verified that the uncoupled code inherits the property that data stored in the 4 nodes can be recovered by connecting to any 2 nodes. As one might expect, this code offers no savings in repair bandwidth over that of the constituent RS codes, since we have simply replicated the same RS code 4 times. We show below how the uncoupled code can be used to create a new coupled-layer (Clay) code that is an MSR code having the desired optimal, repair bandwidth.

Using a Pair of Coordinates to Represent a Layer The coupling of the layers is easier explained in terms of a binary representation (z_0, z_1) of the layer-index z, defined by $z = 2z_0 + z_1$ i.e., $0 \Rightarrow (0,0)$, $1 \Rightarrow (0,1)$, $2 \Rightarrow (1,0)$ and $3 \Rightarrow (1,1)$. We color in red, vertices within a layer for which $x = z_y$ as a means of identifying the layer. For example in Fig. 6, in layer $(z_0, z_1) = (1,1)$, the vertices $(1,0)$, $(1,1)$ are colored red.

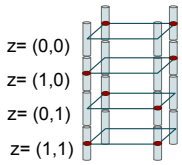


Figure 6: The uncoupled code $\mathscr{U}$.

Pairing of Vertices and Bytes We will abbreviate and write $p = (x, y, z)$ in place of (x, y, z) and introduce a pairing (p, p^*) of vertices within the data cube. The vertices that are colored red are unpaired. The remaining vertices are paired such that a vertex p and its companion p^* both belong to the same y-section. In the data cube of our example code, there are a total of $4 * 4 = 16$ vertices of which 8 are unpaired. The remaining 8 vertices form 4 pairs. Each pair is shown in the data cube appearing on the left in Fig. 7 using a pair of yellow rectangles linked by a dotted line. Mathematically, p^* is obtained from $p = (x, y, z)$ simply by interchanging the values of x and z_y. Examples are presented in Table 3. As mentioned

Vertex $p = (x, y, z_0, z_1)$	Companion p^* (interchange x, z_y)
(0, 0, 1, 0)	(1, 0, 0, 0)
(1, 1, 1, 0)	(0, 1, 1, 1)
(0, 1, 1, 0)	(0, 1, 1, 0) a red vertex, $(p = p^*)$

Table 3: Example vertex pairings.

earlier, each vertex p of the data cube is associated to a byte $U(p) = U(x, y, z)$ of data in the uncoupled code $\mathscr{U}$. We will use $U^*(p)$ to denote the companion $U(p^*)$, of the byte $U(p)$.

Transforming from Uncoupled to Coupled-Layer Code We now show how one can transform in a simply way, a codeword belonging to the uncoupled code $\mathscr{U}$ to a codeword belonging to the Coupled-layer (Clay) code $\mathscr{C}$. As with the uncoupled code, there are a total of 16 bytes making up each codeword in the Clay code. These 16 bytes are stored in a second, identical data cube that is again, composed of 4 horizontal layers, 4 vertical columns with 4 vertices in a layer and 4 vertices per column. Each node corresponds to a column of the data cube and stores a superbyte, made up of 4 bytes. The Clay code $\mathscr{C}$ associates a byte $C(p)$ with each vertex p of the data cube just as does the uncoupled code $\mathscr{U}$. The bytes $U(p)$ and $C(p)$ are related in a simple manner. If p corresponds to an unpaired (and hence colored in red) vertex, we simply set $C(p) = U(p)$. If (p, p^*) are a pair of companion vertices, $p \neq p^*$, $U(p), U^*(p)$ and $C(p), C^*(p)$ are related by the the following pairwise forward transform (PFT):

$$\begin{bmatrix} C(p) \\ C^*(p) \end{bmatrix} = \begin{bmatrix} 1 & \gamma \\ \gamma & 1 \end{bmatrix}^{-1} \begin{bmatrix} U(p) \\ U^*(p) \end{bmatrix}. \quad (1)$$

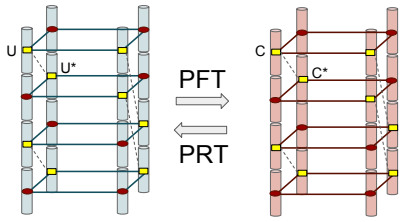


Figure 7: Bytes $C(x, y, z)$ of the Clay code can be obtained from bytes $U(x, y, z)$ of the uncoupled code through a pairwise forward transform and in the reverse direction, by the corresponding pairwise reverse transform. Vertex pairs within a data cube are identified by a pair of yellow rectangles linked by a dotted line.

In the reverse direction, we have $U(p) = C(p)$ respectively if p is unpaired. Else, $U(p), C(p)$ are related by the pairwise reverse transform (PRT):

$$\begin{bmatrix} U(p) \\ U^*(p) \end{bmatrix} = \begin{bmatrix} 1 & \gamma \\ \gamma & 1 \end{bmatrix} \begin{bmatrix} C(p) \\ C^*(p) \end{bmatrix}. \quad (2)$$

We assume γ to be chosen such that $\gamma \neq 0$, $\gamma^2 \neq 1$, and under this condition, it can be verified that any two bytes in the set $\{U(p), U^*(p), C(p), C^*(p)\}$ can be recovered from the remaining two bytes.

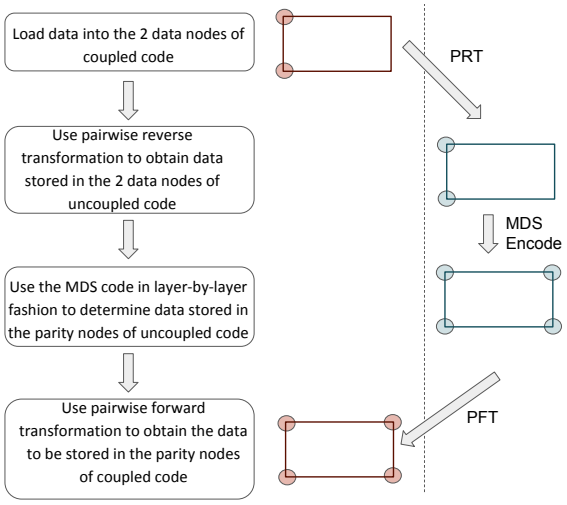


Figure 8: Encoding flowchart for the Clay code. A top view of the nodes is shown on the right. The nodes in pink and blue correspond respectively, to the coupled and uncoupled codes.

Encoding the Clay code The flow chart in Fig.8 provides a self-explanatory description of the encoding process.

Reduced Repair Bandwidth of the Clay Code The savings in repair bandwidth of the Clay code arises from the fact that parity-check constraints are judiciously spread across layers of the C data cube.

In Fig. 9, which shows a portion of the bytes in $\mathscr{C}$, the dotted column corresponds to the failed node having coordinates $(x,y) = (1,0)$. To repair the node, only the two layers $z = (1,0)$ and $z = (1,1)$ corresponding to the presence of red dots within the dotted column are called upon for node repair. Thus each helper node contributes only 2 bytes, as opposed to 4 in an RS code, towards node repair and this explains the savings in repair bandwidth. To understand how repair is accomplished, we turn to Fig. 11. As shown in the figure, the PRT allows us to determine from the the bytes in layers $z = (1,0)$ and $z = (1,1)$ belonging to y-section $y = 1$ in data cube C, the corresponding bytes in data cube U. RS decoding allows us to then recover the bytes $U(p)$ belonging to y-section $y = 0$ in the same two planes. At this point, we have access to the bytes $C(p), U(p)$ for p corresponding to vertices lying in planes $z = (1,0)$ and $z = (1,1)$ and lying in y-section $y = 0$. This set includes 2 of the bytes $C(p)$ in the column corresponding to the failed node. The remaining two bytes $C(p)$ in the failed column can be determined using properties of the PFT.

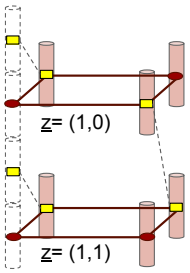
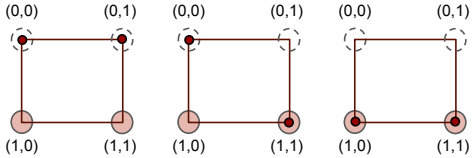


Figure 9: Identifying the failed node and helper data transferred.

Intersection Score To explain decoding, we introduce the notion of an Intersection Score (IS). The IS of a layer is given by the number of hole-dot pairs, i.e., the vertices that correspond to erased bytes and which are at the same time colored red. For example in Fig. 10, when nodes $(0,0)$, $(0,1)$ are erased, layers $(0,0),(0,1),(1,1)$ have respective IS=$2,1,0$.



(a) IS=2 (b) IS=1 (c) IS=0

Figure 10: Illustration of the intersection score (IS) for erasures at $(0,1),(0,2)$.

Decoding The "Decode" algorithm of the Clay code is able to correct the erasure of any $n-k=2$ nodes. Decoding is carried out sequentially, layer-by-layer, in order of increasing IS. This is explained in Fig.12 for the case when nodes $(0,0)$, $(0,1)$ are erased and for layers having IS$=0$, IS$=1$. In a layer with IS$=0$, U bytes can be computed for all non-erased vertices from the known symbols. The erased U bytes are then calculated using RS code decoding. For a layer with IS$=1$, to compute U bytes for all non-erased vertices, we make use of U bytes recovered in layers with IS$=0$. Thus the processing of a layer with $IS = 0$ has to take place prior to processing a layer with $IS = 1$ and so on. Once all the U bytes are recovered, the C bytes can be computed using the PFT. As a result of the simple, pairwise nature of the PFT and PRT, encoding and decoding times are not unduly affected by the coupled-layer structure.

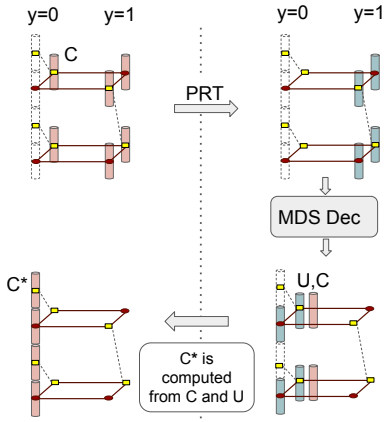


Figure 11: The dotted cylinder identifies the erased node. The bytes shown on the top left represent helper data (6 bytes in all) transferred for repair. The PRT is performed on helper data in C to obtain the bytes (4 bytes) $U(p)$ belonging to the same layers and lying y-section $y = 1$. RS code decoding within each of the two layers is used to obtain the 4 missing $U(p)$ bytes. The bytes corresponding to the erased node in C can then be computed using properties of the PFT.

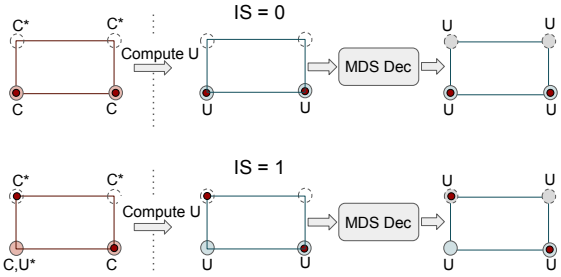


Figure 12: Illustrating how the Clay code recovers from 2 erasures. We begin with a layer having $IS = 0$ (top) before moving to a layer with $IS = 1$ (bottom). Symbols alongside each vertex, indicate which of the 4 bytes $\{C, C^*, U, U^*\}$ are known. (Left) Pink circles indicate non-erased vertices in C. (Middle) Blue vertices indicate vertices in U whose contents can be determined from the available C, U bytes. (Right) Invoking the parity-check equations in U allows all bytes in U to be recovered. Once all the U bytes are recovered, one recovers the remaining unknown bytes C using the PFT.

Clay code parameters Clay codes can be constructed for any parameter set of the form:

$$(n = qt,\ k,\ d)\ \ (\alpha = q^t, \beta = q^{t-1}),\ \text{with}\ q = (d-k+1),$$

for any integer $t \geq 1$ over any finite field of size $Q > n$. The encoding, decoding and repair algorithms can all be generalized for the parameters above. However, in the case $d < n-1$, during single node repair, while picking the d helper nodes, one must include among the d helper nodes, all the nodes belonging to the failed node's y-section.

Clay codes for any (n,k,d) The parameters indicated above have the restriction that $q = (d-k+1)$ divide n. But the construction can be extended in a simple way to the case when q is not a factor of n. For example, for parameters $(n = 14, k = 10, d = 13)$, $q = d-k+1 = 4$. We construct the Clay code taking $n' = 16$, the nearest multiple of q larger than n, and $k' = k+(n'-n) = 12$. While encoding, we set data bytes in $s = (n'-n) = 2$ systematic nodes as zero, and thus the resultant code has parameters $(n = 14, k = 10, d = 13)$. The technique used is called *shortening* in the coding theory literature. We use s temporary buffers each of size equal to chunk size during the encoding, decoding and repair operations. Our implementation of Clay code includes this generalization.

4 Ceph and Vector MDS Codes

4.1 Introduction to Ceph

Ceph [32] is a popular, open-source distributed storage system [33], that permits the storage of data as objects. Object Storage Daemon (OSD) is the daemon process of Ceph, associated with a storage unit such as a solid-state or hard-disk drive, on which user data is stored.

Ceph supports multiple erasure-codes, and a code can be chosen by setting attributes of the erasure-code-profile. Objects will then be stored in logical partitions referred to as pools associated with an erasure-code-profile. Each pool can have a single or multiple placement groups (PG) associated with it. A PG is a collection of n OSDs, where n is the block length of the erasure code associated to the pool.

The allocation of OSDs to a PG is dynamic, and is carried out by the CRUSH algorithm [34]. When an object is streamed to Ceph, the CRUSH algorithm allocates a PG to it. It also performs load balancing dynamically whenever new objects are added, or when active OSDs fail. Each PG contains a single, distinct OSD designated as the primary OSD (p-OSD). When it is required to store an object in a Ceph cluster, the object is passed on to the p-OSD of the allocated PG. The p-OSD is also responsible for initiating the encoding and recovery operations.

In Ceph, the passage from data object to data chunks by the p-OSD is carried out in two steps as opposed to the single-step description in Section 2. For a large object, the amount of buffer memory required to perform encoding and decoding operations will be high. Hence, as an intermediate step, an object is first divided into smaller units called *stripes*, whose size is denoted by S (in bytes). If an object's size is not divisible by S, zeros are padded. The object is then encoded by the p-OSD one stripe at a time. The stripe-size is to be specified within the cluster's configuration file. Both zero padding and system performance are important factors to be considered while fixing a stripe-size.

4.2 Sub-Chunking through Interleaving

To encode, the p-OSD first zero pads each stripe as necessary in order to ensure that the strip size S is divisible by $k\alpha$. The reason for the divisibility by a factor of k is because as described earlier, the first step in encoding is to break up each stripe into k data chunks of equal size. The reason for the additional divisibility requirement by a further factor α arises because we are dealing with a vector code and as explained in Section 2, operations in a vector code involve superbytes, where each superbyte contains α bytes. In what follows, we will assume that S is divisible by $k\alpha$.

The encoding of a stripe is thus equivalent to encoding $N = \frac{S}{k\alpha}$ codewords at a time. The next step as explained in Section 2, is interleaving at the end of which one obtains α sub-chunks per OSD, each of size N bytes. We note that the parameter L introduced in Section 2, is the number of bytes per data chunk and is thus given by $L = \frac{S}{k}$. This notion of sub-chunk is not native to Ceph, but rather is a modification to the Ceph architecture proposed here, to enable the support of vector codes.

The advantage of a vector code is that it potentially enables the repair of an erased coded chunk by passing on a subset of the α sub-chunks. For example, in the Clay code implemented in Ceph is an MSR code, it suffices for each node to pass on β sub-chunks. However, when these β sub-chunks are not sequentially located within the storage unit, it can result in fragmented reads. We analyze such disk read performance degradation in Section 5.

4.3 Implementation in Ceph

Our implementation makes use of the Jerasure [22] and GF-Complete [21] libraries which provide implementations of various MDS codes and Galois-field arithmetic. We chose in our implementation to employ the finite field of size 2^8 to exploit the computational efficiency for this field size provided by the GF-complete library in Ceph.

In our implementation, we employ an additional buffer, termed as *U-buffer*, that stores the sub-chunks associated with the uncoupled symbols U introduced in Section 3. This buffer is of size $nL = S\frac{n}{k}$ bytes. The *U-buffer* is allocated once for a PG, and is used repetitively during encode, decode and repair operations of any object belonging to that PG.

Pairwise Transforms We introduced functions that compute any two sub-chunks in the set $\{U, U^*, C, C^*\}$ given the remaining two sub-chunks. We implemented these functions using the function *jerasure_matrix_dotprod()*, which is built on top of function *galois_w08_region_multiply()*.

Encoding Encoding of an object is carried out by p-OSD by pretending that m parity chunks have been

erased, and then recovering the m chunks using the k data chunks by initiating the decoding algorithm for the code. Pairwise forward and reverse transforms are the only additional computations required for Clay encoding in comparison with MDS encoding.

Enabling Selection Between Repair & Decoding When one or more OSDs go down, multiple PGs are affected. Within an affected PG, recovery operations are triggered for all associated objects. We introduced a boolean function *is_repair()* in order to choose between a bandwidth, disk I/O efficient repair algorithm and the default decode algorithm. For the case of single OSD failure, *is_repair()* always returns *true*. There are multiple failure cases as well for which *is_repair()* returns *true* i.e., efficient repair is possible. We discuss these cases in detail in Appendix A.

Helper-Chunk Identification In the current Ceph architecture, when a failure happens, *minimum_to_decode()* is called in order to determine the k helper chunk indices. We introduced a function *minimum_to_repair()* to determine the d helper chunk indices when repair can be performed efficiently i.e., when *is_repair()* returns *true*. OSDs corresponding to these indices are contacted to get information needed for repair/decode. When there is a single failure, *minimum_to_repair()* returns d chunk indices such that all the chunks that fall in the y-cross-section of the failed chunk are included. We describe the case of multiple erasure cases in detail in Appendix A

Fractional Read For the case of efficient repair, we only read a fraction of chunk, this functionality is implemented by feeding repair parameters to an existing structure *ECSubRead* that is used in inter-OSD communication. We have also introduced a new read function with Filestore of Ceph that supports sub-chunk reads.

Decode and Repair Either the decode or repair function is called depending on whether if *is_repair()* returns *true* or *false* respectively. The decoding algorithm is described in Section 3. Our repair algorithm supports in addition to single-node failure (Section.3), some multiple-erasure failure patterns as well (Section 6).

4.4 Contributions to Ceph

Enabling vector codes in Ceph: We introduced the notion of sub-chunking in order to enable new vector erasure code plugins. This contribution is currently available in Ceph's master codebase [4].

Clay codes in Ceph: We implemented Clay codes as a technique (*cl_msr*) within the *jerasure* plugin. The current implementation gives flexibility for a client to pick any n,k,d parameters for the code. It also gives an option to choose the MDS code used within to be either a Vandermonde-based-RS or Cauchy-original code. The Clay code [2] is yet to be part of Ceph's master codebase.

5 Experiments and Results

The experiments conducted to evaluate the performance of Clay codes in Ceph while recovering from a single node failure are discussed in the present section. Experimental results relating multiple node-failure case can be found in Section 6.1.

5.1 Overview and Setup

Codes Evaluated While Clay codes can be constructed for any parameter set (n,k,d), we have carried out experimental evaluation for selected parameter sets close to those of codes employed in practice, see Table 4. Code C1 has (n,k) parameters comparable to that of the RDP code [9], Code C2 with the locally repairable code used in Windows Azure [16], and Code C3 with the $(20,17)$-RS code used in Backblaze [1]. There are three other codes C4, C5 and C6 that match with the $(14,10)$-RS code used in Facebook data-analytic clusters [25]. Results relating to Codes C4-C6 can be found in Section 6.1, which focuses on repair in the multiple-erasure case.

	(n,k,d)	α	Storage overhead	$\frac{\beta}{\alpha}$
C1	(6,4,5)	8	1.5	0.5
C2	(12,9,11)	81	1.33	0.33
C3	(20,16,19)	1024	1.25	0.25
C4	(14,10,11)	128	1.4	0.5
C5	(14,10,12)	243	1.4	0.33
C6	(14,10,13)	256	1.4	0.25

Table 4: Codes C1-C3 are evaluated in Ceph for single-node repair. The evaluation of Codes C4-C6 is carried out for both single and multiple-node failures.

The experimental results for Clay codes are compared against those for RS codes possessing the same (n,k) parameters. By an RS code, we mean an MDS-code implementation based on the *cauchy_orig* technique of Ceph's *jerasure* plugin. The same MDS code is also employed as the MDS code appearing in the Clay-code construction evaluated here.

Experimental Setup All evaluations are carried out on Amazon EC2 instances of the m4.xlarge (16GB RAM, 4 CPU cores) configuration. Each instance is attached to an SSD-type volume of size 500GB. We integrated the Clay code in Ceph Jewel 10.2.2 to perform evaluations. The Ceph storage cluster deployed consists of 26 nodes. One server is dedicated for the MON daemon, while the remaining 25 nodes each run one OSD. Apart from the installed operating system, the entire 500GB disk is dedicated to the OSD. Thus the total storage capacity of the cluster is approximately 12.2TB.

Model	Object Distribution		Total, T (GB)	Stripe size, S
	Object size (MB)	# Objects		
Fixed (W_1)	64	8192	512	64MB
Variable (W_2)	64	6758	448	1MB
	32	820		
	1	614		

Table 5: Workload models used in experiments.

Overview Experiments are carried out on both fixed and variable object-size workloads, respectively referred to as W_1 and W_2. Workload W_1 has all objects of fixed size 64MB, while in the W_2 workload we choose objects of sizes 64MB, 32MB and 1MB distributed in respective proportions of 82.5%, 10% and 7.5%. Our choices of object sizes cover a good range of medium (1MB), medium/large(32MB) and large (64MB) objects[3], and the distribution is chosen in accordance with that in the Facebook data analytic cluster reported in [23]. The workloads used for evaluation are summarized in Table 5. The stripe-size S is set as 64MB and 1MB for workloads W_1 and W_2 respectively, so as to avoid zero-padding.

The failure domain is chosen to be a node. Since we have one OSD per node, this is equivalent to having a single OSD as the failure domain. We inject node failures into the system by removing OSDs from the cluster. Measurements are taken using *nmon* and *NMONVisualizer* tools. We run experiments with a single PG, and validate the results against the theoretical prediction. We also run the same experiments with 512 PGs, which we will refer to as the multiple-PG case. Measurements are made of (a) repair network traffic, (b) repair disk read, (c) repair time, (d) encoding time and (e) I/O performance for degraded, normal operations.

5.2 Evaluations

Network Traffic: Single Node Failure Network traffic refers to the data transferred across the network during single-node repair. Repair is carried out by the p-OSD, which also acts as a helper node. The network traffic during repair includes both the transfer of helper data to the primary OSD and the transfer of recovered chunk from primary OSD to the replacement OSD. The theoretical estimate for the amount of network traffic is $\frac{T}{k}((d-1)\frac{\beta}{\alpha}+1)$ bytes for a Clay code, versus T bytes for an RS code. Our evaluations confirm the expected savings, and we observed reductions of 25%, 52% and 66%, (a factor of 2.9×) in network traffic for codes C1, C2 and C3 respectively in comparison with the corresponding RS codes under fixed and variable workloads (see Fig. 13(a), 13(d).) As can be seen, the code C3 with the largest value of $q = (d-k+1)$ offer the largest savings in network traffic.

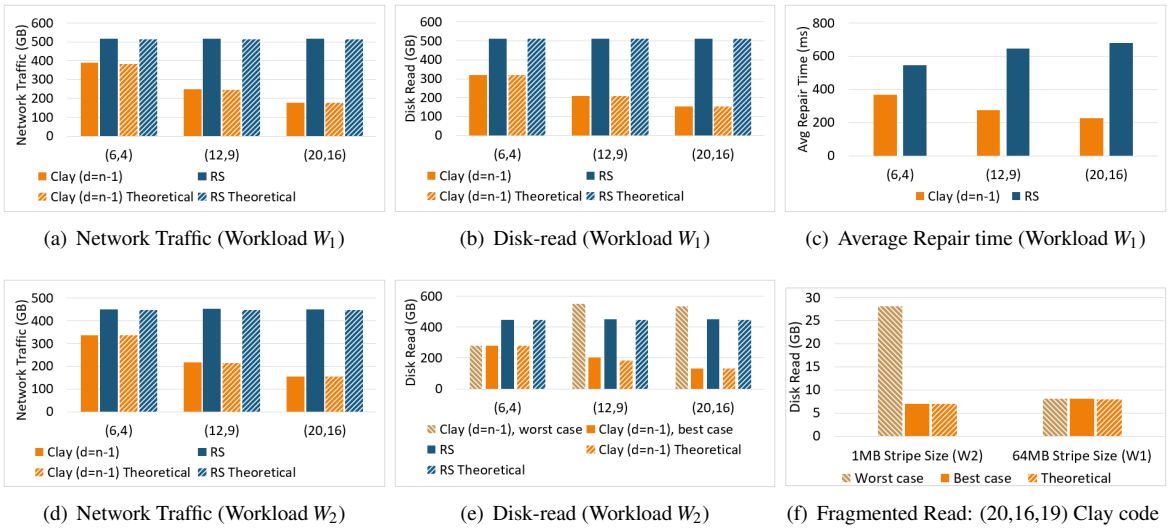
In Ceph, the assignment of OSDs and objects to PGs are done in a dynamic fashion. Hence, the number of objects affected by failure of an OSD can vary across different runs of multiple-PG experiment. We present an network bandwidth performance with 512 PGs under the W_1 workload averaged across 3 runs in Fig. 14. It was observed that in certain situations, an OSD that is already part of the PG can get reassigned as a replacement for the failed OSD. In such cases, the number of failures are treated as two resulting in inferior network-traffic performance in multiple-PG setting.

Disk Read: Single Node Failure The amount of data read from the disks of the helper nodes during the repair of a failed node is referred to as disk read and is an important parameter to minimize.

Depending on the index of the failed node, the sub-chunks to be fetched from helper nodes in a Clay code can be contiguous or non-contiguous. Non-contiguous reads in HDD volumes lead to a slow-down in performance [20]. Even for SSD volumes that permit reads at a granularity of 4kB, the amount of disk read needed depends on the sub-chunk-size. Let us look at, for instance, disk read from a helper node in the case of single node failure for code C3 in workload W2. The stripe-size $S =$ 1MB, and the chunk size is given by $L = S/k =$ 64kB. During repair of a node, $L/(d-k+1) =$ 16kB of data is to be read from each helper node. In the best-case scenario (for example, a systematic node failure), the 16kB data is contiguous, whereas for the worst-case scenario (as in the case of parity node failure) the reads are fragmented. In the latter case, $\beta = 256$ fragments with each of size $L/\alpha = 64$ bytes are read. As a consequence, when 4kB of data is read from the disk, only 1kB ends up being useful for the repair operation. Therefore, the disk read is 4 times the amount of data needed for repair. This is evident in disk read measurements from a helper node in the worst-case as shown in Fig. 13(f). A similar analysis shows that for workload W2, the code C2 leads to additional disk read while C1 does not. This is observed experimentally as well.

On the other hand, for workload W1 with stripe-size $S =$ 64MB, all the three codes C1, C2, and C3 do not cause any additional disk read as shown in Fig. 13(b). For instance, with code C3, fragments of size $S/k\alpha =$ 4kB are to be read in the worst-case scenario. As the size is aligned to the granularity of SSD reads, disk read for the worst-case is equal to $256 * 4$kB=1MB. This is exactly the amount read during best-case as well. (see Fig. 13(f)). In summary, all the three codes result in disk I/O savings for the W1 workload whereas for workload W2 only $C1$ results in an advantage.

The expected disk read from all helper nodes during repair is $\frac{Td\beta}{k\alpha}$ bytes for a Clay code in contrast to T bytes



(a) Network Traffic (Workload W_1) (b) Disk-read (Workload W_1) (c) Average Repair time (Workload W_1)

(d) Network Traffic (Workload W_2) (e) Disk-read (Workload W_2) (f) Fragmented Read: (20,16,19) Clay code

Figure 13: Experimental evaluation of C1, C2 and C3 in comparison with RS codes in a single-PG setting is presented in plots (a)-(e). The plot (f) gives a relative comparison of disk read in a helper node for stripe-sizes 1MB and 64MB for code C3.

for an RS code. In experiments with fixed object-size (see Fig. 13(b)), we obtain savings of 37.5%, 59.3% and 70.2% (a factor of 3.4×) for codes C1, C2 and C3 respectively, when compared against the corresponding RS code. Fig. 14 shows the disk read in the multiple-PG setting.

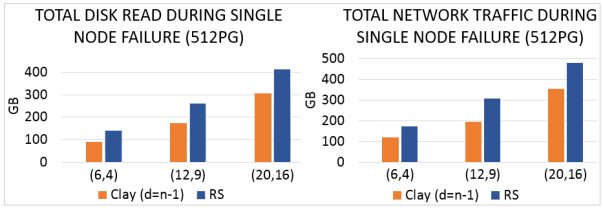


Figure 14: Network traffic and disk read during repair of single node in a setting with 512 PGs, for W_1 workload.

I/O Performance We measured the normal and degraded (i.e., with a repair executing in the background) I/O performance of Clay codes C1-C3, and RS codes with same parameters. This was done using the standard Ceph benchmarking tests for read and write operations. The results are shown in Fig. 15. Under the normal operation, the write, sequential-read and random-read performances are same for both Clay and RS codes. However in the degraded situation, the I/O performance of Clay codes is observed to be better in comparison with RS codes. In particular, the degraded write, read throughput of $(20,16,19)$ Clay code is observed to be more than the $(20,16)$ RS code by 106% and 27% respectively. This can possibly be attributed to the reduced amount of repair data that is read, transmitted and computed on to build the lost data in the erased node.

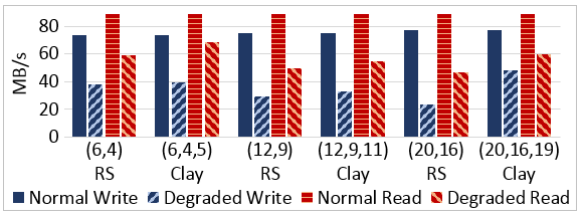


Figure 15: Normal and degraded I/O performance of codes C1, C2, C3 in comparison with RS. The observed values for sequential and random reads are almost the same, and hence plotted as a single value.

Repair Time and Encoding Time We measure the time taken for repair by capturing the starting and stopping times of network activity within the cluster. We observed a significant reduction in repair time for Clay codes in comparison with an RS code. For the code C3 in a single-PG setting, we observe a reduction by a factor of 3× in comparison with an RS code. This is mainly due to reduction in network traffic and disk I/O required during repair. Every affected object requires recovery of $(1/k)$-th fraction of the object size, and the average repair time per object is plotted in Fig. 13(c).

We define the time required by the RADOS utility to place an object into Ceph object-store as the *encoding time*. The encoding time includes times taken for computation, disk-I/O operations, and data transfer across the network. We define the time taken for computing the code chunks based on the encoding algorithm as the *encode computation time*. During encoding, the network traffic and I/O operations are the same for both the classes of codes. Although the encode computation time of Clay code is higher than that of the RS code (See Fig. 16.) the encoding time of a Clay code remains close to that of the corresponding RS code. The increase in the

computation time for the Clay code is due to the multiplications involved in PFT and PRT operations. In storage systems, while data-write is primarily a one-time operation, failure is a norm and thus recovery from failures is a routine activity [12],[24]. The significant savings in network traffic and disk reads during node repair are a sufficient incentive for putting up with overheads in the encode computation time. The decoding time will be almost same as encoding time, since we perform encoding using the decoding function as described in Section 4.3.

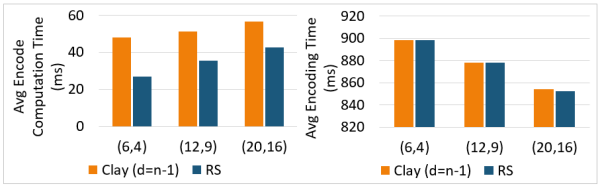


Figure 16: Comparison of average encoding times for C1, C2 and C3 in comparison with RS codes, for the W_1 workload.

6 Handling Failure of Multiple Nodes

The Clay code is capable of recovering from multiple node-failures with savings in repair bandwidth. In the case of multiple erasures, the bandwidth needed for repair varies with the erasure pattern. In Fig. 17, we show the average network traffic of Clay codes with parameters $(n = 14, k = 10, d)$ for $d = 11, 12, 13$ while repairing $f = 1, 2, 3$, and 4 node failures. The average network traffic for repairing f nodes is computed under the assumption that all the f-node-failure patterns are equally likely. Detailed analysis of savings in network traffic for multiple erasures is relegated to Appendix A.

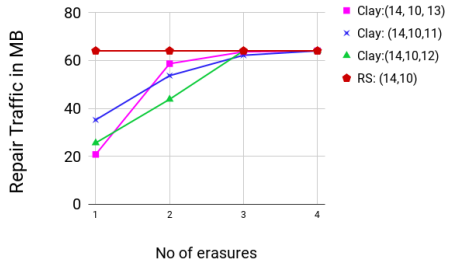


Figure 17: Average theoretical network traffic during repair of 64MB object.

6.1 Evaluation of Multiple Erasures

Network Traffic and Disk Read While the primary benefit of the Clay code is optimal network traffic and disk read during repair of a single node failure, it also yields savings over RS counterpart code in the case of a large number of multiple-node failure patterns. We evaluate the performance of codes C4-C6 (see Table 4) under W_1 workload injecting multiple node-failures in a setting of 512PGs. The plots for network traffic and disk read are shown in Fig. 18, 19.

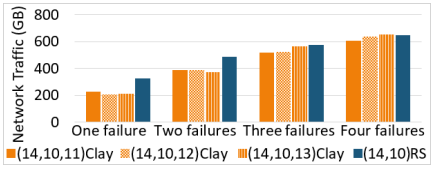


Figure 18: Network traffic evaluation of C4-C6 against RS codes (W_1 workload, multiple-PG).

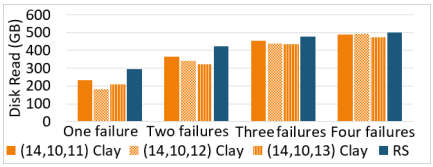


Figure 19: Disk-read evaluation of C4-C6 against RS codes (W_1 workload, multiple-PG).

7 Conclusions

Clay codes extend the theoretical construction presented by Ye & Barg with practical considerations from a coupled-layer perspective that leads directly to implementation. Within the class of MDS codes, Clay codes have minimum possible repair bandwidth and disk I/O. Within the class of MSR codes, Clay codes possess the least possible level of sub-packetization. A natural question to ask is if these impressive theoretical credentials of the Clay code result in matching practical performance. We answer this in the affirmative here by studying the real-world performance of the Clay code in a Ceph setting, with respect to network traffic for repair, disk I/O during repair, repair time and degraded I/O performance. Along the way, we also modified Ceph to support any vector code, and our contribution is now a part of Ceph's master code-base. A particular Clay code, with storage overhead 1.25x, is shown to reduce repair network traffic, disk read and repair times by factors of 2.9, 3.4 and 3 respectively. Much of this is made possible because Clay codes can be constructed via a simple two-step process where one first stacks in layers, α codewords drawn from an MDS code; in the next step, elements from different layers are paired and transformed to yield the Clay code. The same construction with minor modifications is shown to offer support for handling multiple erasures as well. It is our belief that Clay codes are well-poised to make the leap from theory to practice.

8 Acknowledgments

We thank our shepherd Cheng Huang and the anonymous reviewers for their valuable comments. P. V. Kumar would like to acknowledge support from NSF Grant No.1421848 as well as the UGC-ISF research program. The research of Alexander Barg and Min Ye was supported by NSF grants CCF1422955 and CCF1618603.

References

[1] Backblaze data service provider. https://www.backblaze.com/blog/reed-solomon/. Accessed: 2017-Sep-28.

[2] Coupled-layer source code. https://github.com/ceph/ceph/pull/14300/.

[3] Red hat ceph storage: Scalable object storage on qct servers - *a performance and sizing guide*. Reference Architecture.

[4] Sub-chunks: Enabling vector codes in ceph. https://github.com/ceph/ceph/pull/15193/.

[5] Tutorial: Erasure coding for storage applications. http://web.eecs.utk.edu/~plank/plank/papers/FAST-2013-Tutorial.html. Accessed: 2017-Sep-28.

[6] BALAJI, S. B., AND KUMAR, P. V. A tight lower bound on the sub-packetization level of optimal-access MSR and MDS codes. *CoRR abs/1710.05876* (2017).

[7] BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Trans. Computers 44*, 2 (1995), 192–202.

[8] CHEN, H. C., HU, Y., LEE, P. P., AND TANG, Y. Nccloud: A network-coding-based storage system in a cloud-of-clouds. *IEEE Transactions on Computers 63*, 1 (2013), 31–44.

[9] CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), pp. 1–14.

[10] DIMAKIS, A., GODFREY, P., WU, Y., WAINWRIGHT, M., AND RAMCHANDRAN, K. Network coding for distributed storage systems. *IEEE Transactions on Information Theory 56*, 9 (Sep. 2010), 4539–4551.

[11] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *Presented as part of the 9th USENIX Symposium on Operating Systems Design and Implementation* (Vancouver, BC, 2010), USENIX.

[12] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003* (2003), pp. 29–43.

[13] 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.

[14] HU, Y., CHEN, H., LEE, P., AND TANG, Y. NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In *Proceedings of the 10thth USENIX Conference on File and Storage Technologies(FAST)* (2012).

[15] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX, pp. 15–26.

[16] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in Windows Azure storage. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference* (Berkeley, CA, USA, 2012), USENIX ATC.

[17] JIANG, W., HU, C., ZHOU, Y., AND KANEVSKY, A. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *Trans. Storage 4*, 3 (Nov. 2008), 7:1–7:25.

[18] KRALEVSKA, K., GLIGOROSKI, D., JENSEN, R. E., AND VERBY, H. Hashtag erasure codes: From theory to practice. *IEEE Transactions on Big Data PP*, 99 (2017), 1–1.

[19] MURALIDHAR, S., LLOYD, W., ROY, S., HILL, C., LIN, E., LIU, W., PAN, S., SHANKAR, S., SIVAKUMAR, V., TANG, L., AND KUMAR, S. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 383–398.

[20] PAMIES-JUAREZ, L., BLAGOJEVIC, F., MATEESCU, R., GUYOT, C., GAD, E. E., AND BANDIC, Z. Opening the chrysalis: On the real repair performance of MSR codes. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies* (2016), pp. 81–94.

[21] PLANK, J., GREENAN, K., MILLER, E., AND HOUSTON, W. Gf-complete: A comprehensive open source library for galois field arithmetic. *University of Tennessee, Tech. Rep. UT-CS-13-703* (2013).

[22] PLANK, J. S., AND GREENAN, K. M. Jerasure: A library in c facilitating erasure coding for storage applications–version 2.0. Tech. rep., Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[23] RASHMI, K. V., CHOWDHURY, M., KOSAIAN, J., STOICA, I., AND RAMCHANDRAN, K. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* (2016), pp. 401–417.

[24] RASHMI, K. V., NAKKIRAN, P., WANG, J., SHAH, N. B., AND RAMCHANDRAN, K. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST,* (2015), pp. 81–94.

[25] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'13, 2013* (2013), USENIX Association.

[26] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *ACM SIGCOMM 2014 Conference,* (2014), pp. 331–342.

[27] RASHMI, K. V., SHAH, N. B., AND KUMAR, P. V. Optimal Exact-Regenerating Codes for Distributed Storage at the MSR and MBR Points via a Product-Matrix Construction. *IEEE Transactions on Information Theory 57*, 8 (Aug 2011), 5227–5239.

[28] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D. S., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. Xoring elephants: Novel erasure codes for big data. *PVLDB 6*, 5 (2013), 325–336.

[29] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), FAST '07, USENIX Association.

[30] TAMO, I., WANG, Z., AND BRUCK, J. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Transactions on Information Theory 59*, 3 (2013), 1597–1616.

[31] TIAN, C., LI, J., AND TANG, X. A generic transformation for optimal repair bandwidth and rebuilding access in MDS codes. In *2017 IEEE International Symposium on Information Theory, ISIT 2017, Aachen, Germany, June 25-30, 2017* (2017), pp. 1623–1627.

[32] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA* (2006), pp. 307–320.

[33] WEIL, S. A., BRANDT, S. A., MILLER, E. L., AND MALTZAHN, C. Grid resource management - CRUSH: controlled, scalable, decentralized placement of replicated data. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11-17, 2006, Tampa, FL, USA* (2006), p. 122.

[34] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW '07), November 11, 2007, Reno, Nevada, USA* (2007), pp. 35–44.

[35] YE, M., AND BARG, A. Explicit constructions of optimal-access MDS codes with nearly optimal sub-packetization. *IEEE Trans. Information Theory 63*, 10 (2017), 6307–6317.

Appendices

A Handling Failure of Multiple Nodes

The failure patterns that can be recovered with bandwidth-savings are referred to as *repairable failure patterns*. Non repairable failure patterns are recovered by using the decode algorithm.

Repairable Failure Patterns (i) $d < n-1$: Clay codes designed with $d < n-1$ can recover from e failures with savings in repair bandwidth when $e \leq n-d$, with a minor exception described in Remark 1. The helper nodes are to be chosen in such a way that if a y-section contains a failed node, then all the surviving nodes in that y-section must act as helper nodes. If no such choice of helper nodes is available then it is not a repairable failure pattern. For example, consider the code with parameters $(n = 14, k = 10, d = 11)$. The nodes can be put in a (2×7) grid, as $q = d-k+1 = 2$ and $t = \frac{n}{q} = 7$. In Fig.20, we assume that nodes $(0,0)$ and $(0,1)$ have failed, and therefore nodes $(1,0)$ and $(1,1)$ along with any 9 other nodes can be picked as helper nodes.

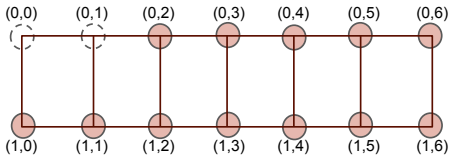


Figure 20: The (2×7) grid of 14 nodes in $(14, 10, 11)$ Clay code. The nodes $(0,0)$ and $(0,1)$ have failed.

(ii) $d = n-1$: When the code is designed for $d = (n-1)$, up to $(q-1)$ failures that occur within a single y-section can be recovered with savings in repair bandwidth. As the number of surviving nodes is smaller than d in such a case, all the surviving nodes are picked as helper nodes. See Fig. 21 for an example of a repairable failure-pattern in the case of a $(14, 10, 13)$ Clay code.

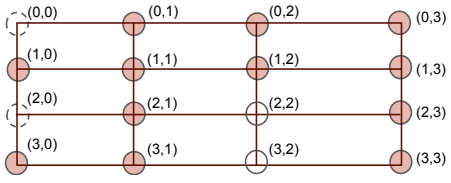


Figure 21: The (4×4) grid containing 14 nodes in $(14, 10, 13)$ Clay code. Note that the cells $(2,2)$ and $(3,2)$ in the grid do not represent nodes. The nodes $(0,0)$ and $(2,0)$ coming from 0-section have failed.

Repair Layers For the case of a single failure, we have already observed that all the layers with IS > 0 are picked. This remains the same for the case of multiple failures as well.

Repair Bandwidth Savings We describe here how to compute network traffic during the repair of a multiple-failure pattern. Let e_i be the number of erased nodes within $(y = i)$-section and $e = (e_0, \cdots, e_{t-1})$. The total number of failures is given by $f = \sum_{i=0}^{t-1} e_i$. The number of helper nodes $d_e = d$ if the code is designed for $d < (n-1)$, and $d_e = n-f$ if it is designed for $d = (n-1)$. Total number of sub-chunks β_e needed from each helper node is same as the number of layers with IS > 0. This can be obtained by subtracting the count of layers with IS$= 0$ from α. The number of helper sub-chunks per node is $\beta_e = \alpha - \prod_{i=0}^{t-1}(q - e_i)$, and network traffic for repair is $d_e\beta_e$. It can be observed that for a single node failure, $f = 1$ and $\beta_e = q^{t-1}$.

Remark 1 *Whenever $d_e\beta_e > k\alpha$, decode algorithm is a better option and the is_repair() function takes care of these cases by returning false. For example, when there are q failures within the same y-section, every layer will have IS > 0 giving $\beta_e = \alpha$ and hence repair is not efficient for this case.*

Repair Algorithm We present a repair algorithm in 1, that is generic for single and multiple erasures. This is invoked whenever savings in bandwidth are possible, i.e, when *is_repair()* returns *true*. In the algorithm, we refer to those non-erased nodes that are not helper nodes as *aloof nodes*.

Algorithm 1 repair

```
Input: E (erasures), I (aloof nodes).
repair_layers = get_repair_layers(E).
set s = 1.
set maxIS = max of IS(E ∪ I, z) over all z from repair_layers
while ( 1 ≤ s ≤ maxIS )
    for (z ∈ repair_layers and IS(E ∪ I, z) = s)
        if (IS(E, z) > 1) G = φ
        else {
          a = the erased node with hole-dot in layer z
          G is set of all nodes in a's y-section.}
        E' = E ∪ G ∪ I
        Compute U sub-chunks in layer z corresponding to all the nodes other than E'
        Invoke scalar MDS_decode to recover U sub-chunks for all nodes in E'
    end for
    s = s + 1
end while
Compute C chunks corresponding to all the erased nodes, from U sub-chunks in repair layers and the helper C sub-chunks in repair layers.
```

Towards Web-based Delta Synchronization for Cloud Storage Services

He Xiao
Tsinghua University

Zhenhua Li *
Tsinghua University

Ennan Zhai
Yale University

Tianyin Xu
UIUC

Yang Li
Tsinghua University

Yunhao Liu
Tsinghua University

Quanlu Zhang
Microsoft Research

Yao Liu
SUNY Binghamton

Abstract

Delta synchronization (sync) is crucial for network-level efficiency of cloud storage services. Practical delta sync techniques are, however, only available for PC clients and mobile apps, but not web browsers—the most pervasive and OS-independent access method. To understand the obstacles of web-based delta sync, we implement a delta sync solution, WebRsync, using state-of-the-art web techniques based on `rsync`, the de facto delta sync protocol for PC clients. Our measurements show that WebRsync severely suffers from the inefficiency of JavaScript execution inside web browsers, thus leading to frequent stagnation and even hanging. Given that the computation burden on the web browser mainly stems from data chunk search and comparison, we reverse the traditional delta sync approach by lifting all chunk search and comparison operations from the client side to the server side. Inevitably, this brings considerable computation overhead to the servers. Hence, we further leverage locality-aware chunk matching and lightweight checksum algorithms to reduce the overhead. The resulting solution, WebR2sync+, outpaces WebRsync by an order of magnitude, and is able to simultaneously support 6800–8500 web clients' delta sync using a standard VM server instance based on a Dropbox-like system architecture.

1 Introduction

Recent years have witnessed considerable popularity of cloud storage services, such as Dropbox, SugarSync, Google Drive, iCloud Drive, and Microsoft OneDrive. They have not only provided a convenient and pervasive data store for billions of Internet users, but also become a critical component of other online applications. Their popularity brings a large volume of network traffic overhead to both the client and cloud sides [28, 37]. Thus, a lot of efforts have been made to improve their network-level efficiency, such as batched sync, deferred sync, delta sync, compression and deduplication [24, 25, 27, 37, 38, 46]. Among these efforts, delta sync is of particular importance for its fine granularity (*i.e.,* the client only sends the changed content of a file to the cloud, instead of the entire file), thus achieving significant traffic savings in the presence of users' file edits [29, 39, 40].

Unfortunately, today delta sync is only available for PC clients and mobile apps, but not for the web—the most pervasive and OS-independent access method [37]. After a file f is edited into a new version f' by users, Dropbox's PC client will apply delta sync to automatically upload only the altered bits to the cloud; in contrast, Dropbox's web interface requires users to manually upload the *entire content* of f' to the cloud.[1] This gap significantly affects web-based user experiences in terms of both sync speed and traffic cost.

Web is a fairly popular access method for cloud storage services: all the major cloud storage services support web-based access, while only providing PC clients and mobile apps for a limited set of OS distributions and devices. One reason is that many users do not want to install PC clients or mobile apps on their devices to avoid the extra storage and CPU/memory overhead; in comparison, almost every device has web browsers. Specially, for the emerging cloud-oriented systems and devices (*e.g.,* Chrome OS and Chromebook) web browsers are perhaps the only option to access cloud storage.

To understand the fundamental obstacles of web-based delta sync, we implement a delta sync solution, WebRsync, using state-of-the-art web techniques including JavaScript, WebSocket, and HTML5 File APIs [14, 18]. WebRsync implements the algorithm of `rsync` [15], the de facto delta sync protocol for PC clients, and works with all modern web browsers that support HTML5. To optimize the execution of JavaScript, we use asm.js [4] to first implement the client side of WebRsync in efficient C code and then compile it to JavaScript. To unravel the performance of WebRsync from the users' perspective, we further develop StagMeter, an automated tool for accurately quantifying the *stagnation* of web browsers, *i.e.,* the browser's not responding to user actions (*e.g.,* mouse clicks) in time, when applying WebRsync.

Our experiments show that WebRsync is severely af-

*Corresponding author. Email: lizhenhua1983@gmail.com

[1] In this paper, we focus on *pervasive* file editing made by any applications that synchronize files to the cloud storage through web browsers, rather than *specific* web-based file editors such as Google Docs, Microsoft Word Online, Overleaf, and GitHub online editor. Technically, our measurements show that the latter usually leverages specific data structures (rather than delta sync) to avoid full-content transfer and save the network traffic incurred by file editing.

fected by the low execution efficiency of JavaScript inside web browsers. Even under simple (or says one-shot) file editing workloads, WebRsync is slower than PC client-based delta sync by 16–35 times, and most time is spent at the client side for performing computation-intensive chunk *search* and *comparison* operations.[2] This causes web browsers to frequently stagnate and even *hang* (*i.e.,* the browser never reacts to user actions). Also, we find that the drawback of WebRsync cannot be fundamentally addressed through native extension, parallelism, or client-side optimization (§4).

Driven by above observations, our first effort towards practical web-based delta sync is to "reverse" the WebRsync process by handing all chunk search and comparison operations over to the server side. This effort also enables us to re-implement these computation-intensive operations in efficient C code. The resulting solution is named WebR2sync (Web-based Reverse `rsync`). It significantly cuts the computation burden on the web client, but brings considerable computation overhead to the server side. To this end, we make two-fold additional efforts to optimize the server-side computation overhead. First, we exploit the locality of users' file edits which can help bypass most (up to ∼90%) chunk search operations in real usage scenarios. Second, by leveraging lightweight checksum algorithms, SipHash [20] and Spooky [17] instead of MD5, we can reduce the complexity of chunk comparison by ∼5 times. The final solution is referred to as WebR2sync+, and we make the source code of all our developed solutions publicly available at `https://WebDeltaSync.github.io`.

We evaluate the performance of WebR2sync+ using a deployed benchmark system based on a Dropbox-like system architecture. We show that WebR2sync+ outpaces WebRsync by an order of magnitude, approaching the performance of PC client-based `rsync`. Moreover, WebR2sync+ is able to simultaneously support 6800–8500 web clients' delta sync using a standard VM server instance under regular workloads [3]. Even under intensive workloads, a standard VM instance with WebR2sync+ deployed can simultaneously support 740 web clients.

2 Delta Sync Support in State-of-the-Art Cloud Storage Services

In this section, we present our qualitative study of delta sync support in state-of-the-art cloud storage services. The target services are selected for either their popularity (Dropbox, Google Drive, Microsoft OneDrive, iCloud Drive, and Box.com), or representativeness in terms of techniques used (SugarSync, Seafile, QuickSync, and DeltaCFS). For each service, we examined its delta sync support with different access methods, using its latest-version (as of April 2017) Windows PC client, Android app, and Chrome web browser. The only exception occurred to iCloud Drive for which we used its latest-version MacOS client, iOS app, and Safari web browser.

Service	PC Client	Mobile App	Web Browser
Dropbox	Yes	No	No
Google Drive	No	No	No
OneDrive	No	No	No
iCloud Drive	Yes	No	No
Box.com	No	No	No
SugarSync	Yes	No	No
Seafile [16]	Yes	No	No
QuickSync [25]	Yes	Yes	No
DeltaCFS [51]	Yes	Yes	No

Table 1: Delta sync support in 9 cloud storage services.

To examine a specific service with a specific access method, we first uploaded a 1-MB [4] highly-compressed new file (f) to the cloud (so the resulting network traffic would be slightly larger than 1 MB). Next, on the user side, we appended a single byte to f to generate an updated file f'. Afterwards, we synchronized f' from the user to the cloud with the specific access method, and meanwhile recorded the network traffic consumption. In this way, we can reveal if delta sync is applied by measuring the traffic consumption—if the traffic consumption was larger than 1 MB, the service did not adopt delta sync; otherwise (*i.e.,* the traffic consumption was just tens of KBs), the service had implemented delta sync.

Based on the examination results listed in Table 1, we have the following observations. First, delta sync has been widely adopted in the majority of PC clients of cloud storage services. On the other hand, it has never been used by the mobile apps of any popular cloud storage services, though two academic services [25,51] have implemented delta sync in their mobile apps and proved the efficacy. In fact, as the battery capacity and energy efficiency of mobile apps grow constantly, we expect delta sync to be widely adopted by mobile apps in the near future [36]. Finally, none of the studied cloud storage services supports *web-based* delta sync, despite web browsers constituting the most pervasive and OS-independent method for accessing Internet services.

3 WebRsync: The First Endeavor

WebRsync is the first workable implementation of web-based delta sync for cloud storage services. It is implemented in JavaScript based on HTML5 File APIs [18] and WebSocket. It follows the algorithm of `rsync` and thus keeps the same behavior as PC client-based ap-

[2] In contrast, when a user downloads a file from the cloud with a web browser, the client-side computation burden of delta sync is fairly low and thus would not cause the web browser to stagnate or hang.

[3] Detailed description of simple, regular, and intensive workloads we use in this work is presented in §6.2.

[4] We also experiment with files much larger than 1 MB in size, *i.e.,* 10 MB and 100 MB, and got the same results.

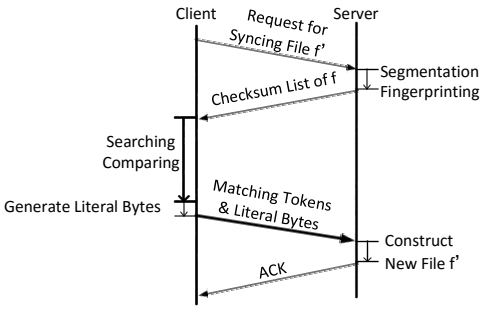


Figure 1: Design flow chart of WebRsync.

proaches. Although it is not a practically acceptable solution, it points out the challenges and opportunities of supporting delta sync under current web frameworks.

3.1 Design and Implementation

We design WebRsync by adapting the working procedure of `rsync` to the web browser scenario. As demonstrated in Figure 1, in WebRsync when a user edits a file from f to f', the client instantly sends a request to the server for the file synchronization. On receiving the request, the server first executes fixed-size chunk *segmentation* and *fingerprinting* operations on f (which is available on the cloud side), and then returns a *checksum list* of f to the client. Except for the last chunk, each data chunk is typically 8 KB in size. Thus when f is 1 MB in size, its checksum list contains 128 weak 32-bit checksums as well as 128 strong 128-bit MD5 checksums [15]. After that, based on the checksum list of f, the client first performs chunk *search* and *comparison* operations on f', and then generates both the *matching tokens* and *literal bytes*. Note that search and comparison operations are both conducted in a byte-by-byte manner on *rolling* checksums; in comparison, segmentation and fingerprinting operations are both conducted in a chunk-by-chunk manner so they incur much lower computation overhead. The matching tokens indicate the overlap between f and f', while the literal bytes represent the novel parts in f' relative to f. Both of them are sent to the server for constructing f'. Finally, the server returns an acknowledgment to the client to conclude the process.

We implement the client side of WebRsync based on the HTML5 File APIs [18] and the WebSocket protocol, using 1500 lines of JavaScript code. Following the common practice to optimize the performance of JavaScript execution, we adopt the asm.js language [4] to first write the client side of WebRsync in C code and then compile it to JavaScript. The server side of WebRsync is developed based on the node.js framework, with 500 lines of node.js code and 600 lines of C code; its architecture follows the server architecture of Dropbox (as an example of the state-of-the-art industrial cloud storage services). Similar to Dropbox, the web service of WebRsync runs on a VM server rent from Aliyun ECS [2], and the file content is hosted on object storage rent from Aliyun OSS [3]. More details on the server, client and network configurations are described in §6.1 and Figure 14.

3.2 Performance Benchmarking

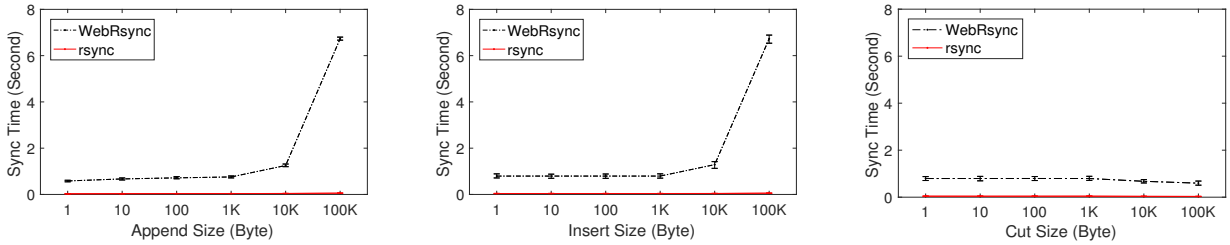
We first compare the performance of WebRsync and `rsync`. We perform random append, insert, and cut [5] operations of different edit sizes (ranging from 1 B, 10 B, 100 B, 1 KB, 10 KB, to 100 KB) upon real-world files collected from real-world cloud storage services. The dataset is collected in our previous work and is publicly released [37], where the average file size is nearly 1 MB. One file is edited for only once, and it is then synchronized from the client side to the server side. For an insert or cut operation, when its edit size reaches or exceeds 1 KB, it is first dispersed into a certain number of (typically 1–20) *continuous sub-edits* [6] to simulate the practical situation of a user edit, and then synchronized to the server. For each of the three different types of edit operations, we first measure its average sync time corresponding to each edit size, and then decompose the average sync time into three stages: server, network, and client. Moreover, we measure its average CPU utilization on the client side corresponding to each edit size.

As shown in Figure 2, for each type of file edit operations the sync time of WebRsync is significantly longer than that of `rsync` (by 16–35 times). In other words, WebRsync is much slower than `rsync` on handling the same file edit. Among the three types of file edits, we notice that syncing a cut operation with WebRsync is always faster than syncing an append/insert operation (for the same edit size), especially when the edit size is relatively large (10 KB or 100 KB). This is because a cut operation reduces the length of a file while an append/insert operation increases the length of a file.

Furthermore, we decompose the sync time of `rsync` and WebRsync into three stages: at the client side, across the network, and at the server side, as depicted in Figures 3a and 3b. For each type of file edits, around 40% of `rsync`'s sync time is spent at the client side and around 35% is spent at the network side; in comparison, the vast majority (60%–92%) of WebRsync's sync time is spent at the client side, while less than 5% is spent at the network side. This indicates that the sync bottleneck of WebRsync is due to the inefficiency of the web browser's executing JavaScript code. Additionally, Figure 3c illustrates that the CPU utilization of each type of file edits in WebRsync is as nearly twice as that of `rsync`, because JavaScript programs consume more CPU resources.

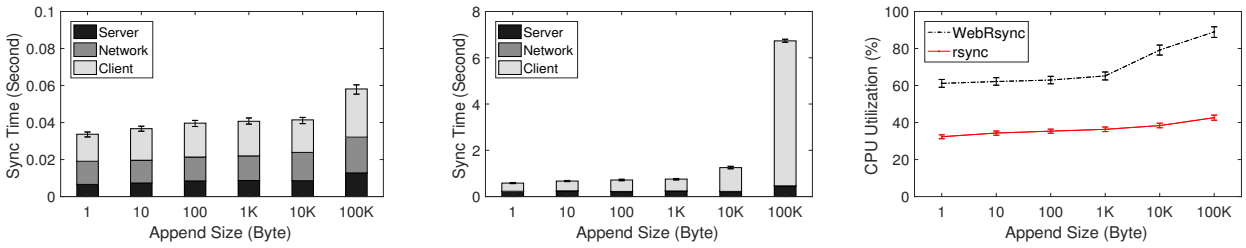
[5] Here "cut" means to remove some bytes from a file.

[6] A *continuous sub-edit* means that the sub-edit operation happens to continuous bytes in the file. More details are explained in § 5.2, especially in Figure 12 and Figure 13.



(a) Append. (b) Insert. (c) Cut.

Figure 2: Average sync time using WebRsync for various sizes of file edits (including append, insert, and cut) under a simple workload. The error bars show the minimum and maximum values at each point.



(a) Breakdown of the sync time of `rsync`. (b) Breakdown of the sync time of WebRsync. (c) Average client-side CPU utilization.

Figure 3: Breakdown of the sync time of (a) `rsync` and (b) WebRsync for append operations, as well as the corresponding average client-side CPU utilizations. The situations for insert and cut operations are similar.

3.3 Measuring Stagnation with StagMeter

As discussed in §3.2, WebRsync not only leads to more sync time, but also costs more computation resources at the client side. The heavy CPU consumption causes web browsers to frequently stagnate and even hang. To quantitatively understand the stagnation of web browser perceived by users, we develop the StagMeter tool to measure the stagnation time by automatically integrating a piece of JavaScript code into the web browser [7]. StagMeter periodically [8] prints the current timestamp on the concerned web page (*e.g.,* the web page that executes delta sync). If the current timestamp (say t) is successfully printed at the moment, there is no stagnation; otherwise, there is a stagnation and then the printing of the current timestamp will be postponed to $t' > t$. Therefore, the corresponding stagnation time is calculated as $t' - t$.

Using StagMeter, we measure and visualize the stagnations of WebRsync (on handling the three types of file edits) in Figure 4. Note that StagMeter only attempts to print 10 timestamps for the first second. Therefore, spaces between consecutive timestamps represent stagnation, and larger spaces imply longer stagnations. As indicated in all the three subfigures, stagnations are directly associated with high CPU utilizations.

[7] We can also directly use the native profiling tool of the Chrome browser to visualize the stagnation, whose results we found more complicated to interpret than those of StagMeter.

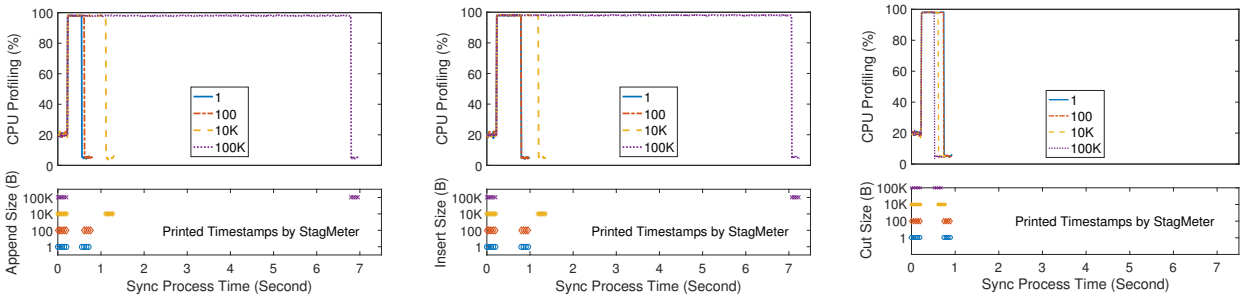
[8] By default we set the period as 100 ms, so as to simulate the minimum intervals of common web users' operations.

4 Native Extension, Parallelism, and Client-side Optimization of WebRsync

This section investigates three approaches to partially addressing the drawback of WebRsync. For each approach, we first describe its working principle, and then evaluate its performance using different types of file edits.

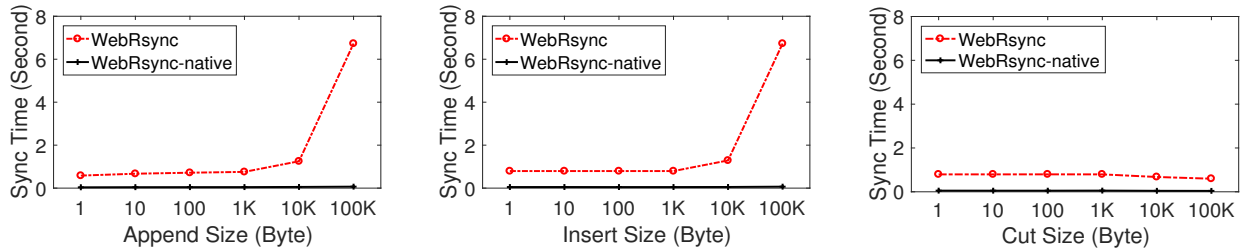
WebRsync-native. Given that the sync speed of WebRsync is much lower than that of the PC client-based delta sync solution (`rsync`), our first approach to optimizing WebRsync is to leverage the *native client* [13] for web browsers. Native client is a sandbox for efficiently and securely executing compiled C/C++ code in a web browser, and has been supported by all mainstream web browsers. In our implementation, we use the Chrome native client to accelerate the execution of WebRsync on the Chrome browser. We first use HTML5 and JavaScript to compose the webpage interface, through which a user can select a local file to synchronize (to the cloud). Then, the path of the selected local file is sent to our developed native client (written in C++). Afterwards, the native client reads the file content and synchronizes it to the cloud in a similar way as `rsync`. When the sync process finishes, the native client returns an acknowledgement message to the webpage interface, which then shows the user the success of the delta sync operation.

Figure 5 depicts the performance of WebRsync-native, in comparison to the performance of original WebRsync. Obviously, WebRsync-native significantly reduces the



(a) Append. (b) Insert. (c) Cut.

Figure 4: Stagnation captured by StagMeter for different edit operations and the associated CPU utilizations. The stagnation time is illustrated by the discontinuation of the timestamp on the sync process time.



(a) Append. (b) Insert. (c) Cut.

Figure 5: Average sync time using WebRsync-native for various sizes of file edits under a simple workload.

sync time of WebRsync, in fact close to the sync time of `rsync`. Accordingly, the CPU utilization is decreased and the stagnation of the Chrome browser is fully avoided. Nevertheless, using native client requires the user to download and install extra plug-in components for the web browser, which essentially impairs the usability and pervasiveness of WebRsync-native.

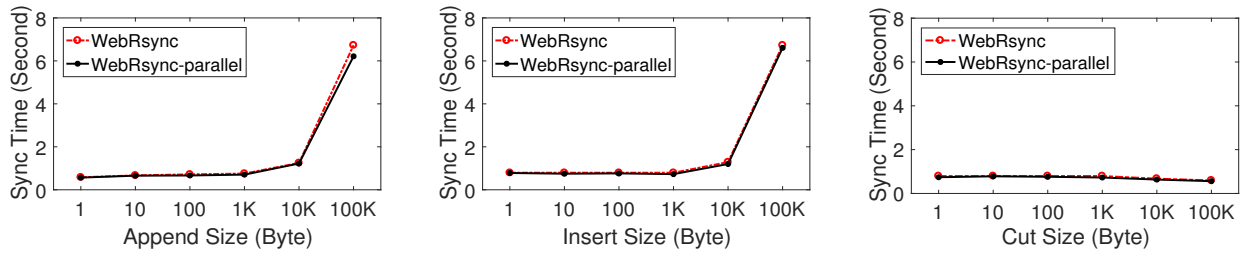
WebRsync-parallel. Our second approach is to use HTML5 *web workers* [10] for parallelism or threading. Generally speaking, when executing JavaScript code in a webpage, the webpage becomes unresponsive until the execution is finished—this is why WebRsync would lead to frequent stagnation and even hanging of the web browser. To address this problem, a web worker is a JavaScript program that runs in the background, independently of other JavaScript programs in the same webpage. When we apply it to WebRsync, the original single JavaScript program is divided to multiple JavaScript programs that work in parallel. Although this approach can hardly reduce the total sync time (as indicated in Figure 6) or the CPU utilizations (as shown in Figure 7, the upper part), it can fully avoid stagnation for the Chrome browser (as shown in Figure 7, the lower part).

WebRsync+. Later in §5.2 we describe in detail how we exploit users' file-edit locality and lightweight hash algorithms to reduce server-side computation overhead. As a matter of fact, the two-fold optimizations can also be applied to the client side. Thereby, we implement the two optimization mechanisms at the client side of WebRsync by translating them from C++ to JavaScript, and the resulting solution is referred to as WebRsync+. As illustrated in Figure 8, WebRsync+ stays between WebRsync and WebR2sync+ in terms of sync time, which is basically within our expectation. Further, we decompose the sync time of WebRsync+ into three stages: at the client side, across the network, and at the server side, as depicted in Figure 9. Comparing Figure 9 with Figure 3b (breakdown of the sync time of WebRsync into three stages), we find that the client-side time cost of WebRsync+ is remarkably reduced thanks to the two optimization mechanisms. However, WebRsync+ cannot fully avoid stagnation for web browsers; instead, it can only alleviate the stagnation compared to WebRsync.

Summary. With the above three-fold efforts, we conclude that the drawback of WebRsync cannot be fundamentally addressed via solely client-side optimizations. That is to say, we need more comprehensive solutions where the server side is also involved.

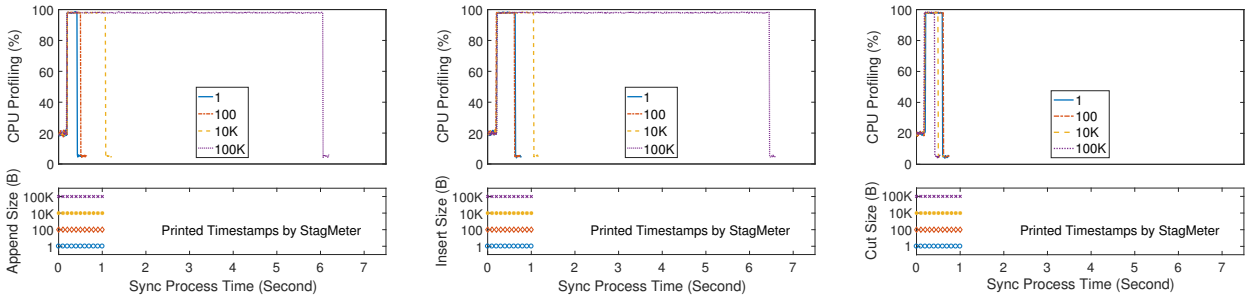
5 WebR2sync+: Web-based Delta Sync Made Practical

This section presents WebR2sync+, the practical solution for web-based delta sync. The practicality of WebR2sync+ is attributed to multi-fold endeavors at both



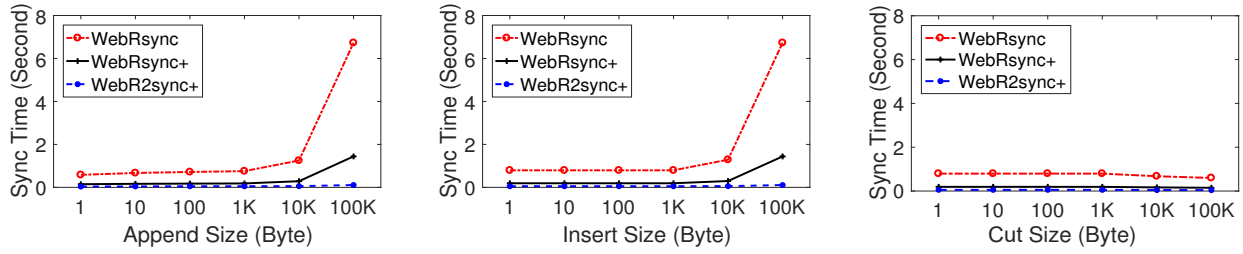
(a) Append. (b) Insert. (c) Cut.

Figure 6: Average sync time using WebRsync-parallel for various sizes of file edits under a simple workload.



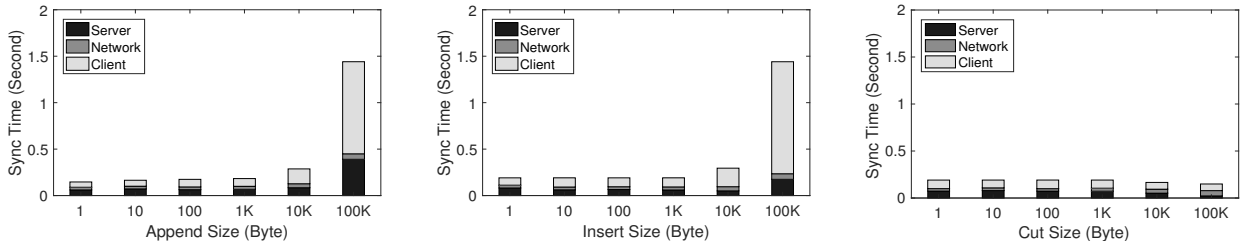
(a) Append. (b) Insert. (c) Cut.

Figure 7: Although WebRsync-parallel is unable to reduce the CPU utilizations (relative to WebRsync), it can fully avoid stagnation for the Chrome web browser by utilizing HTML5 web workers.



(a) Append. (b) Insert. (c) Cut.

Figure 8: Average sync time using WebRsync+ for various sizes of file edits under a simple workload.



(a) Append. (b) Insert. (c) Cut.

Figure 9: Breakdown of the sync time of WebRsync+ (shown in Figure 8) for different types of edit operations.

client and server sides. We first present the basic solution, WebR2sync, which improves WebRsync (§5.1), and then describe the server-side optimizations for mitigating the computation overhead (§5.2). The final solution that combines both WebR2sync with the server-side optimizations is referred to as WebR2sync+ in §5.3.

5.1 WebR2sync

As depicted in Figure 10, to address the overload issue, WebR2sync reverses the process of WebRsync (c.f., Figure 1) by moving the computation intensive search and comparison operations to the server side; meanwhile, it shifts the lightweight segmentation and finger-

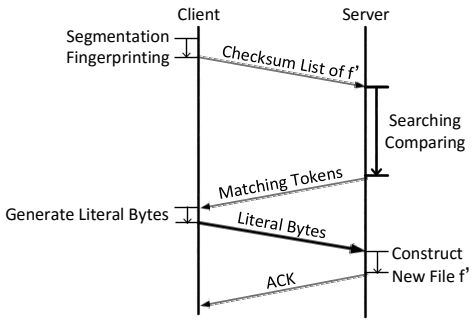


Figure 10: Design flow chart of WebR2sync.

printing operations to the client side. Compared with the workflow of conventional web-based delta sync, in WebRsync, the checksum list of f' is generated by the client and the matching tokens are generated by the server, while the literal bytes are still generated by the client. Note that this allows us to implement the search and comparison operations in C rather than in JavaScript at the server side. Therefore, WebR2sync can not only avoid stagnation for the web client, but also effectively shorten the duration of the whole delta sync process.

5.2 Server-side Optimizations

While WebR2sync significantly cuts the computation burden on the web client, it brings considerable computation overhead to the server side. To this end, we make two-fold additional efforts to optimize the server-side computation overhead.

Exploiting the locality of file edits in chunk search. When the server receives a checksum list from the client, WebR2sync uses a 3-level chunk searching scheme to figure out matched chunks between f and f', as shown in Figure 11 (which follows the 3-level chunk searching scheme of `rsync` [15]). Specifically, in the checksum list of f' there is a 32-bit weak rolling checksum (calculated by the Adler32 algorithm [26]) and a 128-bit strong MD5 checksum for each data chunk in f'. When this checksum list is sent to the server, the server leverages an additional *(rolling checksum) hash table* whose every entry is a 16-bit hash code of the 32-bit rolling checksum [15]. The checksum list is then sorted according to the 16-bit hash code of the 32-bit rolling checksums. Note that a 16-bit hash code can point to multiple rolling and MD5 checksums. Thereby, to find each matched chunk between f and f', the 3-level chunk searching scheme always goes from the 16-bit hash code to the 32-bit rolling checksum and further to the 128-bit MD5 checksum.

The 3-level chunk searching scheme can effectively minimize the computation overhead for general file-edit patterns, particularly random edits to a file. However, it has been observed that real-world file edits typically

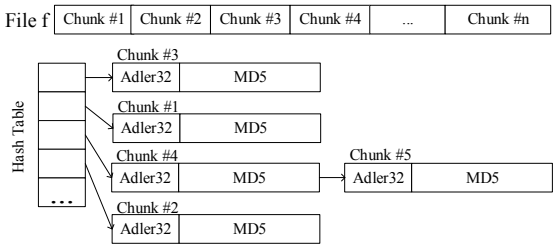
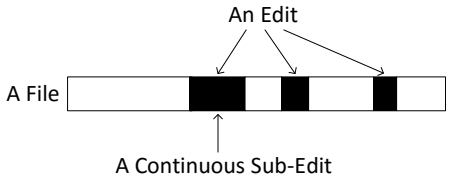


Figure 11: The three-level chunk searching scheme used by `rsync` and WebR2sync.



(a) An edit consists of several continuous sub-edits.

(b) The worst case of a file edit in terms of locality.

Figure 12: An example of continuous sub-edits due to the locality of file edits: (a) the relationship between a file edit and its constituent continuous sub-edits; (b) the worst-case scenario in terms of locality.

follow a local pattern rather than a general (random) pattern, which has been exploited to accelerate file compression and deduplication [41,47–49]. To exemplify this observation in a quantitative manner, we analyze two real-world fine-grained file editing traces with respect to Microsoft Word and Tencent WeChat collected by Zhang *et al.* [51]. The traces are fine-grained since they leveraged a loopback user-space file system (Dokan [7] for Windows) to record not only the detailed information (*e.g.,* edit type, edit offset, and edit length) of users' file operations but also the content of the updated data. In each trace, a user made several *continuous sub-edits* to a file and then did a save operation, and this behavior repeated for many times. Here a continuous sub-edit means that the sub-edit operation happens to continuous bytes in the file, as demonstrated in Figure 12. Our analysis results, in Figure 13, show that in nearly a half (46%) of cases a user saved 1–5 continuous sub-edits, thus indicating fine locality. Besides, in over one third (35%) of cases a user saved 6–10 continuous sub-edits, which still implies sound locality. On the other hand, in only a minority (5%) of cases a user saved more than 16 continuous sub-edits, which means undesirable locality.

The locality of real-world file edits offers us an opportunity to bypass a considerable portion of (unnecessary) chunk search operations. In essence, given that edits to a file are typically local, when we find that the i-th chunk

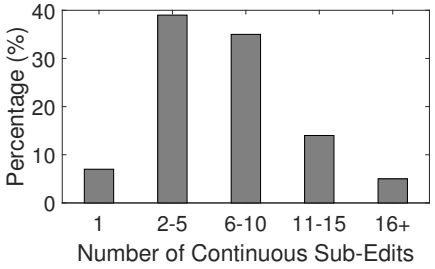


Figure 13: A real-world example of file-edit locality. The number of continuous sub-edits is highly clustered.

of f' matches the j-th chunk of f, the $(i+1)$-th chunk of f' is highly likely to match the $(j+1)$-th chunk of f. Therefore, we "simplify" the 3-level chunk searching scheme by directly comparing the MD5 checksums of the $(i+1)$-th chunk of f' and the $(j+1)$-th chunk of f. If the two chunks are identical, we simply move forward to the next chunk; otherwise, we return to the regular 3-level chunk searching scheme.

Replacing MD5 with SipHash in chunk comparison. By exploiting the locality of users' file edits as above, we manage to bypass most chunk search operations. After that, we notice that the majority of server-side computation overhead is attributed to the calculations of MD5 checksums. Thus, we wonder whether the usage of MD5 is necessary in chunk comparison. MD5 was initially designed as a cryptographic hash function for generating secure and low-collision hash codes [43], which makes it computationally expensive. In our scenario, it is not necessary to use such an expensive hash function, because our purpose is just to obtain a low collision probability. In fact, we can employ the HTTPS protocol for data exchange between the web client and server to ensure the security. Driven by this insight, we decide to replace MD5 with a lightweight pseudorandom hash function [22] in order to reduce the computational overhead.

Quite a few pseudorandom hash functions can satisfy our goal, such as Spooky [17], FNV [9], CityHash [5], SipHash [20], and Murmur3 [12]. Among them, some are very lightweight but vulnerable to collisions. For example, the computation overhead of MD5 is around 5 to 6 cycles per byte [8] while the computation overhead of CityHash is merely 0.23 cycle per byte [19], but the collision probability of CityHash is quite high. On the other hand, some pseudorandom hash functions have extremely low collision probability but are a bit slow. As listed in Table 2, SipHash seems to be a sweet spot — its computation overhead is about 1.13 cycles per byte and its collision probability is acceptably low. By replacing MD5 with SipHash in our web-based delta sync solution, we manage to reduce the computation complexity of chunk comparison by nearly 5 times.

Hash Function	Collision Probability	Cycles Per Byte
MD5	Low ($< 10^{-6}$)	5.58
Murmur3	High ($\approx 1.05 \times 10^{-4}$)	0.33
CityHash	High ($\approx 1.03 \times 10^{-4}$)	0.23
FNV	High ($\approx 1.09 \times 10^{-4}$)	1.75
Spooky	High ($\approx 9.92 \times 10^{-5}$)	0.14
SipHash	Low ($< 10^{-6}$)	1.13

Table 2: A comparison of candidate pseudorandom hash functions in terms of collision probability (on 64-bit hash values) and computation overhead (cycles per byte).

Although the collision probability of SipHash is acceptably low, it is slightly higher than that of MD5. Thus, as a fail-safe mechanism, we make a lightweight full-content hash checking (using the Spooky algorithm) in the end of a file synchronization, so as to deal with possible collisions in SipHash chunk fingerprinting. We select the Spooky algorithm because it works the fastest among all the candidate pseudorandom hash algorithms (as listed in Table 2). If the full-content hash checking fails for the synchronization of a file (with an extremely low probability), we will roll back and re-sync the file with the original MD5 chunk fingerprinting.

5.3 WebR2sync+: The Final Product

The integration of WebR2sync and the server-side optimization produces WebR2sync+. The client side of WebR2sync+ is implemented based on the HTML5 File APIs, the WebSocket protocol, an open-source implementation of SipHash-2-4 [1], and an open-source implementation of SpookyHash [11]. In total, it is written in 1700 lines of JavaScript code. The server side of WebR2sync+ is developed based on the node.js framework and a series of C processing modules. The former (written in 500 lines of node.js code) handles the user requests, and the latter (written in 1000 lines of C code) embodies the reverse delta sync process together with the server-side optimizations.

6 Evaluation

This section evaluates the performance of WebR2sync+, in comparison to WebRsync, WebR2sync and (PC client-based) `rsync` under a variety of workloads.

6.1 Experiment Setup

To evaluate different sync approaches, we set up a Dropbox-like system architecture by running the web service on a standard VM server instance (with a quad-core Intel Xeon CPU @2.5GHz and 16-GB memory) rent from Aliyun ECS, and all file content is hosted on object storage rent from Aliyun OSS. The ECS VM server and OSS storage are located at the same data center so there is no bottleneck between them. The client side of WebR2sync+ was executed in the Google Chrome

Figure 14: Experiment setup in China.

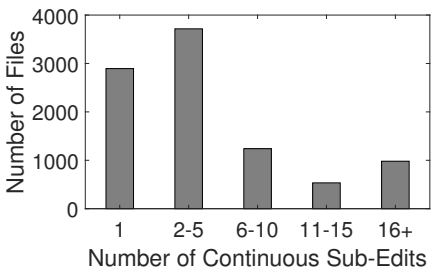


Figure 15: File-edit locality in the source files of two successive Linux kernel releases (versions 4.5 and 4.6).

browser (Windows version 56.0) running on a laptop with a quad-core Intel Core-i5 CPU @2.8GHz, 16-GB memory, and an SSD disk. The server side and client side lie in different cities (*i.e.,* Shanghai and Beijing) and different ISPs (*i.e.,* China Unicom and CERNET), as depicted in Figure 14. The network RTT is ∼30 ms and the network bandwidth is ∼100 Mbps. Therefore, the network bottleneck is kept minimal in our experiments so that the major system bottleneck lies at the server and/or client sides. If the network condition becomes much worse, the major system bottleneck might shift to the network connection.

6.2 Workloads

To evaluate the performance of WebR2sync+ under various practical usage scenarios, as compared to WebRsync, WebR2sync, and `rsync`, we generate *simple* (*i.e.,* one-shot), *regular* (*i.e.,* periodical), and *intensive* workloads. To generate simple workloads, we make random append, insert, and cut operations of different edit sizes against real-world files collected from real-world cloud storage services. The collected dataset is described in §3.2. One file is edited for only once (the so-called "one-shot"), and it is then synchronized from the client side to the server side. For an insert or cut operation, when its edit size $\geq$ 1 KB, it is first dispersed into 1–20 continuous sub-edits and then synchronized to the server.

Regular and intensive workloads are mainly employed to evaluate the service throughput of each solution. To generate regular workloads, we still make a certain type of edit to a typical file but the edit operation is executed every 10 seconds. To generate a practical intensive workload, we use a benchmark of over 8755 pairs of source files taken from two successive releases (versions 4.5 and 4.6) of the Linux kernel source trees. The average size of the source files is 23 KB and the file-edit locality is generally stronger than that in Figure 13 (as shown in Figure 15). Specifically, we first upload all the files of the old version to the server side in an FTP-like manner. Then, we synchronize all the files of the new version one by one to the server side using the target approaches (including `rsync`, WebRsync, WebR2sync, WebR2sync with SipHash, and WebR2sync+). There is no time interval between two sequential file synchronizations.

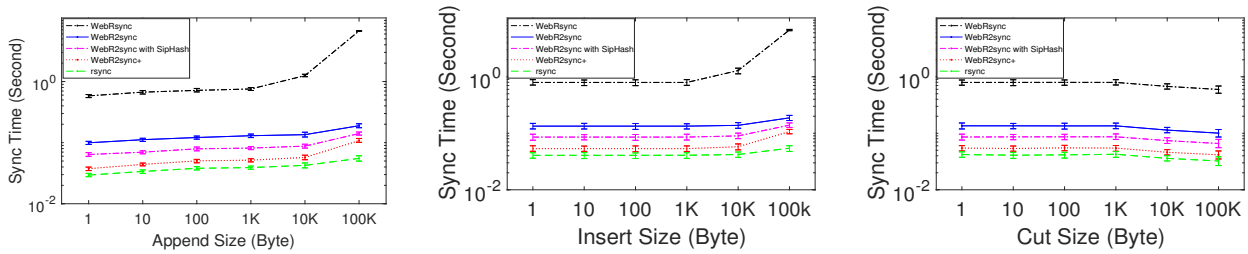
6.3 Results

This part presents our experiment results in four aspects: 1) *sync efficiency* which measures how quick a file operation is synchronized to the cloud; 2) *computation overhead* which explains the difference in sync efficiency of the studied solutions; 3) *sync traffic* which quantifies how much network traffic is saved by each solution; and 4) *service throughput* which shows the scalability of each solution using standard VM server instances.

Sync efficiency. We measure the efficiency of WebR2sync+ in terms of the time for completing the sync. Figure 16 shows the time for syncing against different types of file operations. We can see that the sync time of WebR2sync+ is substantially shorter than that of WebR2sync (by 2 to 3 times) and WebRsync (by 15 to 20 times) for every different type of operations. Note that Figure 16 is plotted with a log scale. In other words, WebR2sync+ outpaces WebRsync by around an order of magnitude, approaching the speed of PC client-based `rsync`. Furthermore, we observe that the sync time of WebR2sync with SipHash always lies between those of WebR2sync and WebR2sync+. This confirms that neither of our server-side optimizations (SipHash and locality exploiting, refer to §5.2) is indispensable.

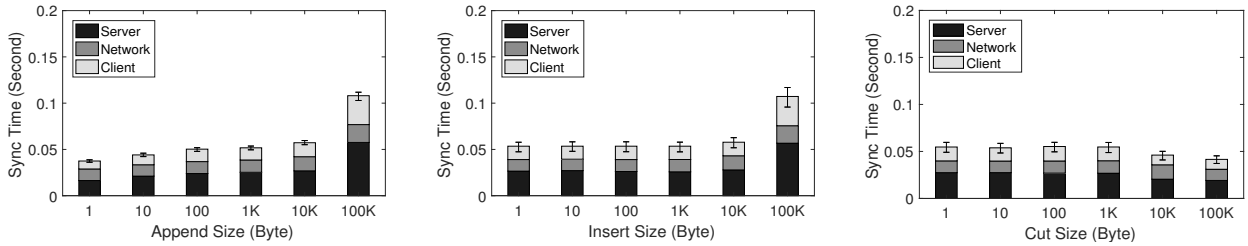
Similar as Figure 3b, we further break down the sync time of WebR2sync+ into three stages as shown in Figure 17. Comparing Figure 17 and Figure 3b, we notice that the majority of sync time is attributed to the client side for WebRsync, while it is attributed to the server side for WebR2sync+. This indicates that the computation overhead of the web browsers in WebRsync is substantially reduced in WebR2sync+, which also saves web browsers from stagnation and hanging.

Computation overhead. Moreover, we record the client-side and server-side CPU utilizations in Figure 18 and Figure 19, respectively. On the client side, WebRsync consumes the most CPU resources while



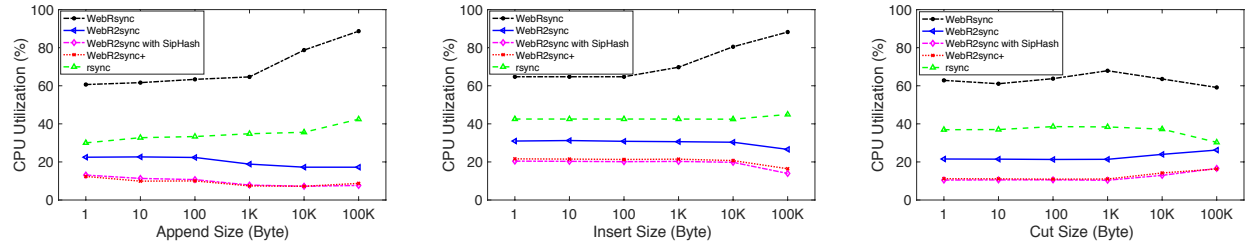
(a) Append. (b) Insert. (c) Cut.

Figure 16: Average sync time of different delta sync approaches for various sizes of file edits under a simple workload.



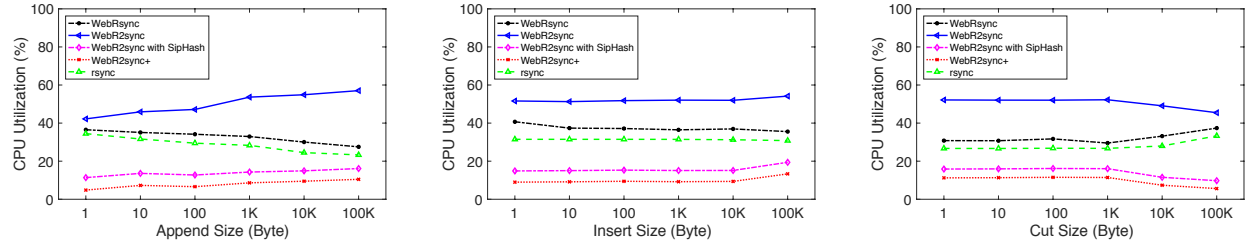
(a) Append. (b) Insert. (c) Cut.

Figure 17: Breakdown of the sync time of WebR2sync+ (shown in Figure 16) for different types of edit operations.



(a) Append. (b) Insert. (c) Cut.

Figure 18: Average **client**-side CPU utilization of different delta sync approaches under a simple workload.



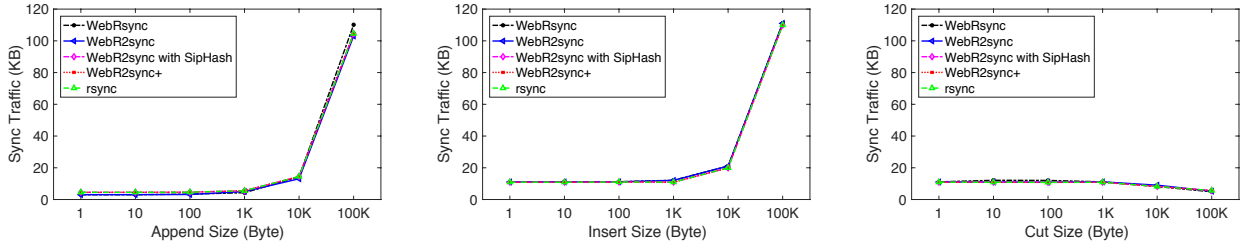
(a) Append. (b) Insert. (c) Cut.

Figure 19: Average **server**-side CPU utilization of different delta sync approaches under a simple workload.

WebR2sync+ consumes the least. PC client-based `rsync` consumes nearly a half CPU resources as compared to WebRsync, and the CPU utilization of WebR2sync lies between `rsync` and WebR2sync+. Owing to the moderate ($< 30\%$) CPU utilizations, both the clients of WebR2sync and WebR2sync+ do not exhibit stagnation.

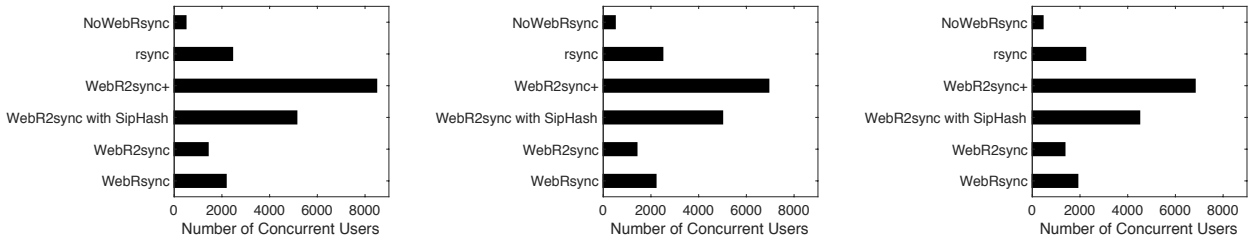
On the server side, WebR2sync consumes the most CPU resources because the most computation-intensive chunk search and comparison operations are shifted from the client to the server. On the contrary, WebR2sync+ consumes the least CPU resources, which validates the efficacy of our two-fold server-side optimizations.

Sync traffic. Figure 20 illustrates the sync traffic consumed by the different approaches. We can see that for any type of edits, the sync traffic (between 1 KB and 120 KB) is significantly less than the average file size ($\sim$1 MB), confirming the power of delta sync in improving network-level efficiency of cloud storage services.



(a) Append. (b) Insert. (c) Cut.

Figure 20: Sync traffic of different sync approaches for various sizes of file edits under a simple workload.



(a) Append. (b) Insert. (c) Cut.

Figure 21: Number of concurrent clients supported by a single VM server instance (as a measure of service throughput) under regular workloads (periodically syncing various sizes of file edits).

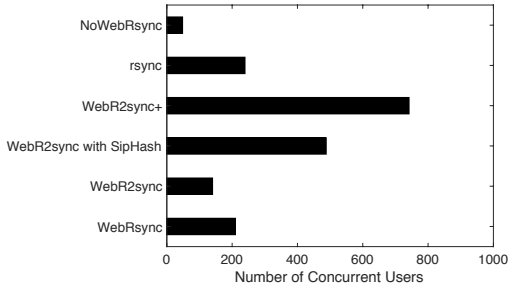


Figure 22: Number of concurrent users supported by a single VM server instance under intensive workloads (syncing two versions of Linux source trees).

For the same edit size the sync traffic of an append operation is usually less than that of an insert operation, because the former would bring more matching tokens while fewer literal bytes (refer to Figure 1). Besides, when the edit size is relatively large (10 KB or 100 KB), a cut operation consumes much less sync traffic than an append/insert operation, because a cut operation brings only matching tokens but not literal bytes.

Service throughput. Finally, we measure the service throughput of WebR2sync+ in terms of the number of concurrent clients it can support. In general, as the number of concurrent clients increases, the main burden imposed on the server comes from the high CPU utilizations in all cores. When the CPU utilizations on all cores approach 100%, we record the number of concurrent clients at that time as the service throughput. As shown in Figure 21, WebR2sync+ can simultaneously support 6800–8500 web clients' delta sync using a standard VM server instance under regular workloads. This throughput is as 3–4 times as that of WebR2sync/`rsync` and as ∼15 times as that of NoWebRsync. NoWebRsync means that no web-based delta sync is used for synchronizing file edits, *i.e.,* directly uploading the entire content of the edited file to the cloud. Also, we measure the service throughput of each solution under intensive workloads (which are mixed by the three types of edits, refer to §6.2). The results in Figure 22 indicate that even under the intensive workloads, WebR2sync+ can simultaneously support 740 web clients' delta sync using a single VM server instance.

7 Related Work

Delta sync, also known as delta encoding or delta compression, is a way of storing or transmitting data in the form of differences (deltas) between different versions of a file, rather than the complete content of the file [6]. It is particularly useful for network applications where file modifications or incremental data updates frequently happen, *e.g.,* storing multiple versions of a file, distributing consecutive user edits to a file, and transmitting video sequences [33]. In the past 4 decades, a variety of delta sync algorithms or solutions have been put forward, such as UNIX `diff` [32], `Vcdiff` [34], WebExpress [31], Optimistic Deltas [21], `rsync` [15], and content defined chunking (CDC) [35].

Due to its efficiency and flexibility, `rsync` has become the de facto delta sync protocol widely used in practice. It was originally proposed by Tridgell and Mackerras in 1996, as an algorithm for efficient remote update of data over a high-latency, low-bandwidth network link [45]. Then in 1999, Tridgell thoroughly discussed its design, implementation, and performance in [44]. Being a standard Linux utility included in all popular Linux distributions, `rsync` has also been ported to Windows, FreeBSD, NetBSD, OpenBSD, and MacOS [15].

According to a real-world usage dataset [37], the majority (84%) of files are modified by the users for at least once, thus confirming the importance of delta sync on network-level efficiency of cloud storage services. Among all mainstream cloud storage services, Dropbox was the first to adopt delta sync (more specifically, `rsync`) in around 2009 in its PC client-based file sync process [39]. Then, SugarSync, iCloud Drive, and Seafile followed the design choice of Dropbox by utilizing delta sync (`rsync` or CDC) to reduce their PC clients' and cloud servers' sync traffic. After that, two academic cloud storage systems, namely QuickSync [25] and DeltaCFS [51], further implemented delta sync (`rsync` and CDC, respectively) for mobile apps.

Drago *et al.* studied the system architecture of Dropbox and conducted large-scale measurements based on ISP-level traces of Dropbox network traffic [28]. They observed that the Dropbox traffic was as much as one third of the YouTube traffic, which strengthens the necessity of Dropbox's adopting delta sync. Li *et al.* investigated in detail the delta sync process of Dropbox through various types of controlled benchmark experiments, and found it suffers from both traffic and computation overuse problems in the presence of frequent, short data updates [39]. To this end, they designed an efficient batched synchronization algorithm called UDS (update-batched delayed sync) to reduce the traffic usage, and further extended UDS with a backwards compatible Linux kernel modification to reduce the CPU usage (recall that delta sync is computation intensive).

Despite the wide adoption of delta sync (particularly `rsync`) in cloud storage services, practical delta sync techniques are currently only available for PC clients and mobile apps rather than web browsers. To this end, we introduced the general idea of web-based delta sync with basic motivation, preliminary design, and early-stage performance evaluation using limited workloads and metrics [50]. In this paper, our work is conducted based on [50] while goes beyond it in terms of techniques, evaluations, and presentations.

8 Conclusion and Future Work

This paper presents a series of efforts towards a practical solution of web-based delta sync for cloud storage services. We first leverage the state-of-the-art techniques (including `rsync`, JavaScript, HTML5 File APIs, and WebSocket) to develop an intuitive web-based delta sync solution named WebRsync. Despite not being practically acceptable in terms of performance, WebRsync effectively helps us understand the obstacles to support web-based delta sync. Particularly, we observe that the inefficiency of JavaScript execution significantly stagnates the sync process of WebRsync. Thereby, we propose and implement WebR2sync+, a practical web-based delta sync solution by moving expensive chunk search and comparison operations from the client side to the server side. It combines with optimizations at the server side that exploit the locality of users' file edits and uses lightweight pseudorandom hash functions to replace the traditional expensive cryptographic hash function. WebR2sync+ outpaces WebRsync by an order of magnitude, and is able to simultaneously support around 6800–8500 web clients' delta sync using a standard VM server instance under a Dropbox-like system architecture.

We are investigating the following aspects as the future work. First, we are looking for a seamless way to integrate the server-side design of WebR2sync+ with the back-end of commercial cloud storage vendors (like Dropbox and iCloud Drive). Specifically, WebR2sync+ needs to cooperate with data deduplication, compression, bundling, *etc.* [23, 27]. Moreover, we would like to explore the benefits of using more fine-grained and complex delta sync protocols, such as CDC and its variants [30, 42, 49]. In addition, we envision to expand the usage of WebR2sync+ for a broader range of web service scenarios, not limited to web browsers and cloud storage services. For example, when a user wants to use a web-based app to upload a file f' to a common web server (such as Apache, Nginx, or IIS) which has already stored an old version of the file (f), web-based delta sync has the great potential to reduce network traffic and operation time. In this case, the major challenge lies in the requirement of modifying the web server implementation; minimizing the modification efforts is under investigation.

Acknowledgments

We thank the anonymous reviewers for their positive and constructive comments. Besides, we appreciate the valuable guidance and detailed suggestions from our shepherd, Vasily Tarasov, during the revision of the paper. In addition, we thank Yonghe Wang for helping with some measurements during the preparation of the paper. This work is supported by the High-Tech R&D Program of China ("863–China Cloud" Major Program) under grant 2015AA01A201, the NSFC under grants 61471217, 61432002, 61632020 and 61472337. Ennan Zhai is partly supported by the NSF under grants CCF-1302327 and CCF-1715387.

References

[1] A Javascript Implementation of SipHash-2-4. https://github.com/jedisct1/siphash-js.

[2] Aliyun ECS (Elastic Compute Service). https://www.aliyun.com/product/ECS.

[3] Aliyun OSS (Object Storage Service). https://www.aliyun.com/product/oss.

[4] asm.js, a strict subset of JavaScript that can be used as a low-level, efficient target language for compilers. http://asmjs.org.

[5] CityHash. https://opensource.googleblog.com/2011/04/introducing-cityhash.html.

[6] Delta encoding, the Wikipedia page. https://en.wikipedia.org/wiki/Delta_encoding.

[7] Dokan: An user mode file system for Windows. https://dokan-dev.github.io.

[8] eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to/results-hash.html.

[9] FNV Hash. http://www.isthe.com/chongo/tech/comp/fnv/.

[10] HTML5 Web Workers. https://www.w3schools.com/html/html5_webworkers.asp.

[11] Javascript version of SpookyHash. https://github.com/jamesruan/spookyhash-js.

[12] Murmur3 Hash Function. https://github.com/aappleby/smhasher.

[13] Native Client for Google Chrome. https://developer.chrome.com/native-client.

[14] Reading Files in JavaScript using the HTML5 File APIs. https://www.html5rocks.com/en/tutorials/file/dndfiles/.

[15] rsync Web Site. http://www.samba.org/rsync.

[16] Seafile: Enterprise file sync and share platform with high reliability and performance. https://www.seafile.com/en/home.

[17] Spookyhash: A 128-Bit Noncryptographic Hash. http://burtleburtle.net/bob/hash/spooky.html.

[18] Using files from web applications. https://developer.mozilla.org/en-US/docs/Using_files_from_web_applications.

[19] ALAKUIJALA, J., COX, B., AND WASSENBERG, J. Fast Keyed Hash/Pseudo-random Function Using SIMD Multiply and Permute. *arXiv preprint arXiv:1612.06257* (2016).

[20] AUMASSON, J.-P., AND BERNSTEIN, D. SipHash: a Fast Short-input PRF. In *Proc. of the International Conference on Cryptology in India* (2012), Springer, pp. 489–508.

[21] BANGA, G., DOUGLIS, F., RABINOVICH, M., ET AL. Optimistic Deltas for WWW Latency Reduction. In *Proc. of ATC* (1997), USENIX, pp. 289–303.

[22] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying Hash Functions for Message Authentication. In *Proc. of Crypto* (1996), Springer, pp. 1–15.

[23] BOCCHI, E., DRAGO, I., AND MELLIA, M. Personal Cloud Storage Benchmarks and Comparison. *IEEE Transactions on Cloud Computing (TCC) 5*, 4 (2015), 751–764.

[24] BOCCHI, E., DRAGO, I., AND MELLIA, M. Personal Cloud Storage: Usage, Performance and Impact of Terminals. In *Proc. of CloudNet* (2015), IEEE, pp. 106–111.

[25] CUI, Y., LAI, Z., WANG, X., DAI, N., AND MIAO, C. QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services. In *Proc. of MobiCom* (2015), ACM, pp. 592–603.

[26] DEUTSCH, P., AND GAILLY, J.-L. Zlib Compressed Data Format Specification Version 3.3. Tech. rep., RFC Network Working Group, 1996.

[27] DRAGO, I., BOCCHI, E., MELLIA, M., SLATMAN, H., AND PRAS, A. Benchmarking Personal Cloud Storage. In *Proc. of IMC* (2013), ACM, pp. 205–212.

[28] DRAGO, I., MELLIA, M., MUNAFÒ, M., SPEROTTO, A., SADRE, R., AND PRAS, A. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proc. of IMC* (2012), ACM, pp. 481–494.

[29] E, J., CUI, Y., WANG, P., LI, Z., AND ZHANG, C. CoCloud: Enabling Efficient Cross-Cloud File Collaboration based on Inefficient Web APIs. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 29*, 1 (2018), 56–69.

[30] EL-SHIMI, A., KALACH, R., KUMAR, A., OTTEAN, A., LI, J., AND SENGUPTA, S. Primary Data Deduplication – Large Scale Study and System Design. In *Proc. of ATC* (2012), USENIX, pp. 285–296.

[31] HOUSEL, B., AND LINDQUIST, D. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. In *Proc. of MobiCom* (1996), ACM, pp. 108–116.

[32] HUNT, J., AND MACILROY, M. *An Algorithm for Differential File Comparison*. Bell Laboratories New Jersey, 1976.

[33] HUNT, J., VO, K., AND TICHY, W. An Empirical Study of Delta Algorithms. *Software Configuration Management* (1996), 49–66.

[34] KORN, D., AND VO, K.-P. Engineering a Differencing and Compression Data Format. In *Proc. of ATC* (2002), USENIX, pp. 219–228.

[35] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal Content Defined Chunking for Backup Streams. In *Proc. of FAST* (2010), USENIX, pp. 239–252.

[36] LI, Z., DAI, Y., CHEN, G., AND LIU, Y. *Content Distribution for Mobile Internet: A Cloud-based Approach*. Springer, 2016.

[37] LI, Z., JIN, C., XU, T., WILSON, C., LIU, Y., CHENG, L., LIU, Y., DAI, Y., AND ZHANG, Z.-L. Towards Network-level Efficiency for Cloud Storage Services. In *Proc. of IMC* (2014), ACM, pp. 115–128.

[38] LI, Z., WANG, X., HUANG, N., KAAFAR, M., LI, Z., ZHOU, J., XIE, G., AND STEENKISTE, P. An Empirical Analysis of a Large-scale Mobile Cloud Storage Service. In *Proc. of IMC* (2016), ACM, pp. 287–301.

[39] LI, Z., WILSON, C., JIANG, Z., LIU, Y., ZHAO, B., JIN, C., ZHANG, Z.-L., AND DAI, Y. Efficient Batched Synchronization in Dropbox-like Cloud Storage Services. In *Proc. of ACM/IFIP/USENIX Middleware* (2013), Springer, pp. 307–327.

[40] LI, Z., ZHANG, Z.-L., AND DAI, Y. Coarse-grained Cloud Synchronization Mechanism Design May Lead to Severe Traffic Overuse. *Tsinghua Science and Technology 18*, 3 (2013), 286–297.

[41] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZIS, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proc. of FAST* (2009), USENIX, pp. 111–123.

[42] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A Low-bandwidth Network File System. *ACM SIGOPS Operating Systems Review 35* (2001), 174–187.

[43] RIVEST, R., ET AL. RFC 1321: The MD5 Message-digest Algorithm. *Internet activities board 143* (1992).

[44] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. 1999.

[45] TRIDGELL, A., MACKERRAS, P., ET AL. The `rsync` Algorithm. *The Australian National University* (1996).

[46] XIA, W., JIANG, H., FENG, D., DOUGLIS, F., SHILANE, P., HUA, Y., FU, M., ZHANG, Y., AND ZHOU, Y. A Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proceedings of the IEEE 104*, 9 (2016), 1681–1710.

[47] XIA, W., JIANG, H., FENG, D., AND HUA, Y. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proc. of ATC* (2011), USENIX, pp. 26–30.

[48] XIA, W., JIANG, H., FENG, D., AND HUA, Y. Similarity and Locality Based Indexing for High Performance Data Deduplication. *IEEE Transactions on Computers (TC) 64*, 4 (2015), 1162–1176.

[49] XIA, W., ZHOU, Y., JIANG, H., FENG, D., HUA, Y., HU, Y., LIU, Q., AND ZHANG, Y. FastCDC: a Fast and Efficient Content-defined Chunking Approach for Data Deduplication. In *Proc. of ATC* (2016), USENIX, pp. 101–114.

[50] XIAO, H., LI, Z., ZHAI, E., AND XU, T. Practical Web-based Delta Synchronization for Cloud Storage Services. In *Proc. of HotStorage* (2017), USENIX.

[51] ZHANG, Q., LI, Z., YANG, Z., LI, S., GUO, Y., AND DAI, Y. DeltaCFS: Boosting Delta Sync for Cloud Storage Services by Learning from NFS. In *Proc. of ICDCS* (2017), IEEE, pp. 264–275.

Stash in a Flash

Aviad Zuck[1], Yue Li[2], Jehoshua Bruck[2], Donald E. Porter[3], and Dan Tsafrir[1,4]
[1]Technion–Israel Institute of Technology [2]California Institute of Technology
[3]University of North Carolina at Chapel Hill [4]VMware Research Group
{aviadzuc,dan}@cs.technion.ac.il, {yli,bruck}@caltech.edu, porter@cs.unc.edu

Abstract

Encryption is a useful tool to protect data confidentiality. Yet it is still challenging to hide the very presence of encrypted, secret data from a powerful adversary. This paper presents a new technique to hide data in flash by manipulating the voltage level of pseudo-randomly-selected flash cells to encode two bits (rather than one) in the cell. In this model, we have one "public" bit interpreted using an SLC-style encoding, and extract a private bit using an MLC-style encoding. The locations of cells that encode hidden data is based on a secret key known only to the hiding user.

Intuitively, this technique requires that the voltage level in a cell encoding data must be (1) not statistically distinguishable from a cell only storing public data, and (2) the user must be able to reliably read the hidden data from this cell. Our key insight is that there is a wide enough variation in the range of voltage levels in a typical flash device to obscure the presence of fine-grained changes to a small fraction of the cells, and that the variation is wide enough to support reliably re-reading hidden data. We demonstrate that our hidden data and underlying voltage manipulations go undetected by support vector machine based supervised learning which performs similarly to a random guess. The error rates of our scheme are low enough that the data is recoverable months after being stored. Compared to prior work, our technique provides 24x and 50x higher encoding and decoding throughput and doubles the capacity, while being 37x more power efficient.

1 Introduction

The ability to successfully hide data is becoming increasingly important for modern computer users, who often store private and sensitive data on their personal devices. These devices are often stolen or misplaced, jeopardizing confidentiality of sensitive data [1–5]. Although encryption can hide data contents, encryption alone cannot hide the presence of encrypted data. Over time, flaws in encryption techniques can be discovered. Moreover, law enforcement agencies, intelligence agencies, and other potent adversaries are increasingly capable of forcing users to submit the decryption keys or passphrases for their devices [6–9]. Thus, for highly-sensitive data, there is value in hiding the very *presence* of the data.

Commercial forces also drive the need to hide small amounts of data within larger data sets. Economic espionage [10] is forcing companies to find ways to protect and safely circulate sensitive data. Hidden data can also be used to identify copyright infringement, using techniques such as digital watermarking [11]. Hardware validation and fingerprinting is also gaining traction as manufacturers seek cheap and efficient ways to validate products and authenticate their components so they cannot be copied and faked [12, 13]. Thus, both privacy and commercial concerns drive the need for additional data hiding tools, both for users and corporations.

This paper presents a new approach for hiding sensitive data within a larger data set on a NAND flash device. This larger data set can be public, or encrypted with a standard encrypted storage system, like Bitlocker [14] or FileVault [15]; we refer to this larger set as public data for brevity. Within this public data set, our technique encodes hidden data using small manipulations of voltage levels in a subset of the flash cells storing public data.

This paper focuses on NAND flash memory, for both practical and technical reasons. On the practical side, flash is ubiquitous in embedded systems, mobile phones, USB thumb drives, and in solid-state disks (SSDs) on personal laptops—precisely the type of devices that are most likely to be lost, stolen, or confiscated. SSDs are also significant in data centers and servers, which could also be the subject of search or seizure.

From a technical perspective, flash is well-suited for data hiding because it offers high-density, fast random access, and non-volatile storage, but with an abundance of internal randomness [16] that is typically masked by on-device firmware. Internally, flash stores data by electrically charging arrays of floating gate transistors/memory cells to a predefined voltage. To read the data back, the stored voltage levels are coarsely discretized into a one or a zero. This discretization process is noisy—the voltage levels across cells in the device vary widely. Even within one device, the charge levels in flash cells have a high variance, attributable to the inherent noisiness of the programming process, variations created in the manufacturing process, and voltage interference inherent to flash cell transistor technology (see §4). Because the flash programming process is impre-

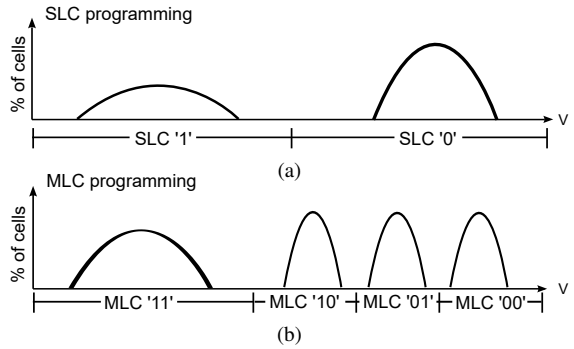


Figure 1: Typical voltage level distributions of cells in SLC (a), and MLC (b) flash memories. Leftmost curves are for programmed cells in the erased state, which are negatively charged. MLC distributions are typically narrower.

cise, flash manufacturers face a trade-off between programming time and storage density, as well as reduced lifetime [17, 18]. The opportunity we see is that there is enough natural variation to hide data in a typical flash array without leaving telltale statistical anomalies— even with an attacker powerful enough to measure the voltage level or other physical characteristics of each cell and run data analysis tools on the voltage level distributions.

The trade-off between write time and precision in flash encoding is well-known, and we leverage this in our design. By taking multiple fine-grain charging and sensing steps, one can more precisely and gradually increase the voltage to a desired level [19]. Single-level cell (SLC) flash can store only one bit selected from one of two voltage levels, whereas multi-level cell (MLC) flash uses four voltage levels and can store two bits, three-level cell (TLC) flash uses eight voltage levels, etc. [20]. Fig. 1 illustrates typical cell voltage distributions for SLC, and MLC. Devices commonly transition cells between SLC and MLC/TLC mode dynamically [21–30]. In other words, the number of bits stored in any given cell can be changed dynamically within a wider range than is commonly used—the only differences are that writing more bits is slower and one needs to know how to interpret the voltage levels of the cell when it is re-read.

In this work we store hidden data by transparently increasing the densities of select flash cells, but without creating a detectable deviation in the overall cell voltage distribution. In our model, a user can access hidden data according to normal methods; the user can hide data with a secret key, that selects certain cells to program with finer-grained variation in the voltage level. Thus, an important part of this work is measuring the expected variance in a faster and coarser charging process (e.g., SLC), and then ensuring that the result of a finer-grained charging process is within this distribution.

Our data hiding scheme, called VolTage-HIde (VT-HI), selects a small number of cells to store an extra bit, from a a larger field of cells not storing hidden data. VT-HI uses a slower charging process to more precisely charge selected cells to a voltage range that represents the logical state of a public and a hidden bit (e.g., converting from SLC to MLC). The cells not selected for hiding data are programmed using standard, widely-available programming operations to store normal data. Public data in VT-HI is assumed to be encrypted with one key, and a second key is used to locate and decrypt hidden data.

The closest related work to ours (Program-Time-HIde, or PT-HI), hides data in flash memory by encoding hidden bits using the different programming times of groups of cells. VT-HI, on the other hand, directly stores data in flash cells, by mimicking the incremental storage technique internally employed by flash vendors. Our straightforward approach has several advantages:

- Encoding is 24x faster in VT-HI and 37x more energy-efficient.
- Decoding of hidden data requires a single, non-destructive, read operation. This makes the decoding process 50x faster. Hidden data can also be read multiple times, while maintaining the integrity of public data.
- Copying hidden data without knowledge of the relevant secret key is impossible, while erasing hidden data (e.g., when in fear of device confiscation) is almost instantaneous.
- The generic nature of VT-HI makes it applicable to multiple chip models from different vendors.

VT-HI is feasible in existing flash-based devices without any hardware modifications, although firmware support would be helpful. For current devices, we approximate the required firmware support on real devices using a sequence of partial programming (PP) [16] operations, where a normal program operation is aborted midway. Using this method, the level of additional charge stored in a cell is roughly correlated with the relative time that the program operation is executed before being aborted. We note that PP steps require only standard flash interface commands [31] (i.e., PROGRAM and RESET).

Hidden data is read using a vendor-specific command that shifts the reference threshold voltage for reading. This command is used in modern flash chips by all vendors to measure voltage distributions and to improve retention [32–35]. Storing and reading public data in VT-HI requires only standard flash operations (e.g., PROGRAM and READ) in order to read data in coarse-grain voltage ranges. Notably, over time flash technology increasingly supports reading in ever finer granularities (e.g., up to four bits per cell [36, 37]).

We evaluate the effectiveness of VT-HI by measuring several issues:

1. **Does VT-HI detectably perturb the voltage levels on the device?** Using the methodology in prior data hiding work [38], we find that, under the most favorable circumstances, a Support-Vector Machine (SVM) can only achieve 50–53% accuracy, or *roughly equivalent to random*.
2. **Does VT-HI encode data faster than the current state of the art technique?** VT-HI is 24x faster and 37x more energy-efficient than PT-HI, the closest related work.
3. **Does VT-HI induce faster wear on the device?** Yes, writing hidden data amplifies writes to hidden cells by a factor of ten; this is an order-of-magnitude reduction compared to the state of the art (PT-HI requires 625). This also only applies to the small fraction of cells storing hidden data.
4. **What is the capacity of VT-HI?** Our implementation uses about 0.02% of the bits to hide data on unmodified devices; with firmware support, this could be increased to 0.2%, or double the capacity of the current state-of-the-art.

In total, these results indicate that the naturally-occurring variability in a flash device creates enough noise to form a useful substrate for data hiding techniques. As part of a larger steganographic system or watermarking system, VT-HI has the particular advantage of creating a variable number of bits; a long-standing challenge for data hiding systems is that the number of bits on a device or in a file is a zero-sum game. Moreover, although the building blocks for VT-HI are not exported to users by most flash vendors, this paper makes the case that VT-HI would be feasible in current flash controllers or firmware.

2 Related Work

Exploiting the Noisiness of Flash to Hide Data. The closest related work to ours is PT-HI [38], which creates a covert channel from the programming time of flash cells. PT-HI applies several hundreds-to-thousands of normal programming cycles to groups of cells, which in turn lengthens the programming time of some cells. Hidden data is encoded based on which cells are slower or faster to program. In other words, the technique creates subtle yet hard-to-detect variations in programming times of each group. A particular advantage of this design, not present in our proposed design, is that these variations persist even if co-located public data persists.

A particular disadvantage of PT-HI is performance: both writing hidden data and reading it requires between dozens, up to hundreds, of programming steps. Decoding in PT-HI is not only time consuming but also a destructive process that destroys any public data stored on the device, and reduces the device's overall lifetime. In addition, the error rate of the hidden payload significantly increases after only a few hundred public data Program/Erase Cycles (PEC), severely limiting the number of times a user can store hidden as well as public data on the device. When combined, these limitations potentially disqualify PT-HI as a building block for a long-lived, steganographic SSD.

Low-level variation in flash has also been used to create a unique fingerprint of flash-based devices [16, 39]. Such fingerprints can be used to authenticate a device's origin. Others suggested to use flash for approximate storage [40].

Hiding Information through Steganography. Our work continues the theme of past research in the field of steganography. Embedding hidden data unto digital objects such as image, audio, and video files is typically achieved by applying small unnoticeable distortions [41–43], abusing existing transmission protocols [44, 45], or in a visible transmission channel [46–49]. A common theme is using inherent noisiness to disguise data hidden within the noise. These solutions often face challenges with mutable data, as data like photographs are typically not expected to change.

Steganographic file systems [50–55] hide data in locations known only to the user, using a hash function on a file name and password. Plausible deniability solutions masquerade hidden data as random content visibly stored alongside regular content [56, 57].

A key limitation of many steganographic file systems is that the total number of bits is fixed. Any bits that are not available to the file system are potential telltale signs of hidden data, and require alternative explanations, like free space, that can fail to hold up if an attacker takes multiple snapshots of the device. Flash firmware can thwart such traditional solutions by leaving multiple copies of data on the device. Several works proposed to solve these and other problems on flash-based devices by openly inserting random-content, undecryptable blocks to the system as part of the system's normal operation [58–60]. However, such solutions still give away the steganographic nature of the system, which may void any claim for the user's innocence for some potent attackers (e.g., intelligence officer in an authoritative regime).

Thus, an advantage of our proposed solution (VT-HI) and PT-HI as building block for a steganographic solution is that they can create hidden bits of storage that do not necessarily reveal the presence of hidden data on the device. In our proposed work (VT-HI) in particular, changes to cells that store both hidden and public data can be excused as routine firmware maintenance (§9.2).

3 NAND Flash Background

NAND flash memories store data using floating gate (FG) cells [61]. Flash packages are divided into blocks,

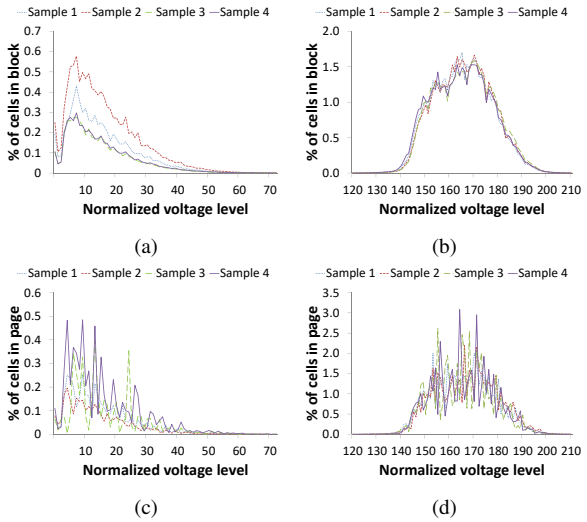


Figure 2: Voltage level distributions of charged cells in four sample 1x-nm MLC chips of the same model. Distributions exhibit significant noisiness at the block level for both non-programmed (a) and programmed (b) cells. (c) and (d) show the distributions at the page level, which exhibit even greater noisiness.

typically 256–2048 KB in size. Blocks are further divided into pages, typically 4–16 KB in size. Pages are stored on physical wordlines, which are serially connected FG cells. When data is written, the cells are electrically charged using small incremental charging steps to a predefined voltage, which traps electrons in the floating gate. The logical value of a cell is read by comparing its voltage to predefined reference threshold voltages placed between relevant voltage intervals. When the flash memory is in MLC/TLC mode, the same cell stores several logical bits by comparing to multiple, smaller voltage intervals. In such cases, several logical pages are stored in a single physical wordline.

An important constraint of flash memories is the lack of support for in-place updates. Once a cell is charged, its level of voltage can only be increased [62, 63]. Voltage is only lowered with an erase operation, which is applied at the granularity of a block (256–2048 KB). Blocks in modern MLC chips can typically endure up to 3K Program/Erase Cycles (PEC). Thus, most SSD vendors include a flash translation layer (FTL), which dynamically remaps logical addresses onto different physical pages [61]; this indirection facilitates rewriting data onto new blocks, garbage collecting old versions of data, and migrating "cold" data onto new blocks for erasure and wear leveling.

4 Flash Variability

The basis for this work is that variability in voltage level distributions of flash cells can be used to hide data. This section gives the reader a sense of the typical range and sources of variation, using measurements from a sample flash chip. The next section explains how we leverage this variability for data hiding.

The inherent variability of flash manifests in three ways relevant to our goals, described and characterized in prior work [16, 35, 64–66]. First, there is significant noise in the programming process. Second, the variability in the chip manufacturing process creates noticeable, naturally-occurring differences in the cell voltage distributions from different NAND flash samples, even from the same vendor, batch and chip model. Finally, there are significant variations in the Bit Error Rate (BER) of different hardware units. VT-HI leverages this inherent noisiness of the charge levels in flash cells, by applying tiny manipulations within the margin of naturally-occurring variations.

We measure the range of these variations in a representative 1x-nm NAND flash memory model from a major vendor (not listed because of an NDA, see §6.2 for details) , using the following procedure. First, we programmed pseudorandom data to select blocks from four flash chip samples from the same model, and measured the cell voltage distributions for each sample [35,67,68]. On each run, a new random data pattern was used. We repeated this process for 0 to 3000 PEC.

Figure 2 shows some[1] of the voltage distributions of the non-programmed/erased cell state and the full distribution of a programmed state (used to represent data bits "1" and "0", respectively) measured from four blocks (Figures 2a and 2b) and four pages (Figures 2c and 2d), each from a different sample that carries the same number of PEC. We note that 99.99% of cells are concentrated between levels [0, 70] and [120, 210], for non-programmed and programmed cells respectively. Notably, these are essentially SLC distributions. For more fine-grain distributions, such as MLC, TLC, and QLC, the voltage ranges are narrower [17, 32].

Figures 2a and 2c demonstrate a known phenomenon where non-programmed cells become partially charged due to interference from programming nearby cells [69].

In Figure 2, the long tails and general width of these curves indicate a wide range of valid voltage levels, and the nonsmoothness of the voltage distributions indicates that a uniformly random bit pattern does not generate uniform distributions of voltage levels. At the page-level the variability is even greater, due to disturbances from neighboring pages, and from having a smaller sample relative to blocks. Furthermore, there are noticeable variations in the distributions of different samples. Note

[1]The current NAND flash interface only allows measurement of positive voltage (V) in discrete normalized units (0-255 in this model), indicating that the programming process is noisy. Therefore, the distributions of erased cells that have negative voltage were not measured.

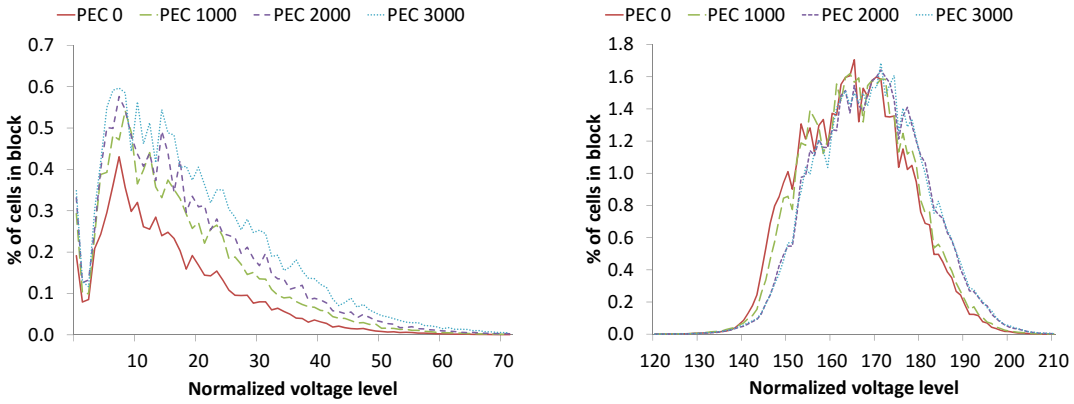


Figure 3: Voltage level distributions tend to shift to the right over the lifetime of cells. The figures show distributions for (a) non-programmed and (b) programmed cells with increasing PEC.

that our measurements were taken from blocks in different physical areas of the same chip.

Figure 3 illustrates variation in voltage levels due to aging. The figure shows the block-level voltage distributions in a flash sample after different numbers of program/erase cycles (PEC). As cells with higher PEC are more easily overprogrammed, their voltage distributions tend to have higher means compared to those of cells with lower PEC.

Finally, we measured variation in BER across hardware units in the same package, normalized to the same PEC count. Commensurate with the other results, and prior studies [65, 66], variations in BER of programmed data in flash exist regardless of PEC (as well as an expected increase in BER as PEC increases).

The measurements in this section establish the range of expected voltage levels in a flash device that is programmed with encrypted data, which should roughly appear as a uniformly-random bit pattern. VT-HI stores hidden data with a special, additional flash programming pass. If the overall voltage distribution stays indistinguishable from measurements on the same chip, there will be no telltale anomalies on the device that would indicate additional data is hidden in those cells. This section indicates that there is a wide berth for reliably hiding data within flash voltage levels.

5 Hiding Data

In this section, we describe how users utilize VT-HI to hide data, the relevant threat model and specific VT-HI techniques for a user to hide data on a flash chip; the data flow of VT-HI is illustrated in Figure 4. .

5.1 Usage Overview

Given a flash device, we model the problem as two users, normal user (NU) and hiding user (HU). These can also

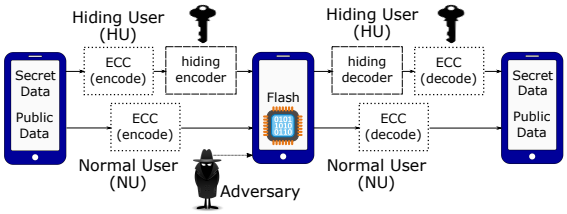


Figure 4: Flow of hiding data on flash in a mobile device.

be thought of as two "modes" or "roles" for the same human user, such as writing to a day planner in normal mode (as NU), but editing sensitive data in hidden mode (as HU). The NU wants to store her public data in flash memory. The HU wants to hide her data inside the data of the NU on the same device, and provides a private, secret, key to VT-HI, which determines the locations in the normal data device where the HU's data will be hidden as extra hidden bits in the chosen cells.

The NU need not be aware of any private keys to correctly read her data. With the secret key, HU's data can also be located and read, without altering the state of public data. Special care must be taken to avoid destroying HU data when the public NU data containing it is migrated or invalidated. The HU must either re-embed the hidden data in a new location (e.g., a page containing newly written NU data), before the old NU page containing it is permanently erased, or apply redundancy a scheme (e.g., parity encoding) to provide some protection for hidden data.

5.2 Threat Model

We assume an adversary who does not know the secret key used to select cells containing hidden data, but has access to the flash device and the capabilities to write and read flash as well as to probe the voltage levels of every cell. We assume that the adversary only gains access

Algorithm 1: Encoding algorithm for VT-HI. The main loop is repeated m times.

```
1 VT-HI (Page, Key, P, H, Vth);
    Input: Flash page number, secret key, two sets of
           bits to store, and a threshold voltage. P is
           public data and H is hidden data.
2 Use PRNG(Key, Page) to select |H|
    non-programmed public bit offsets to store
    hidden bits;
3 Program P to Page;
4 Encrypt H using Key and apply ECC;
5 repeat
6     Read cell voltage levels in Page;
7     Partial program all hidden "0" bits with
        Voltage < Vth;
8 until all hidden "0" bits have Voltage ≥ Vth;
```

to the device after the hidden payload was stored (see Figure 4). We further assume that the device is spyware-free and that the adversary cannot compare snapshots of the device state over time (we discuss multiple snapshot adversaries in §9). Probing cell voltage levels is widely supported by modern NAND flash memories [35,67,68], and was used as a tool for NAND characterization in this work. An adversary who suspects that the user is hiding data with our technique, can try to detect the existence of such data, as indicated by unexpected charge distributions in a subset of cells. We assume that the VT-HI capability is either added and removed at will by the user or is omnipresent (see §9), and therefore does not raise suspicion in itself.

However, even with perfect knowledge of charge distributions and the exact configuration parameters of VT-HI (e.g., hidden bits per page), there will be no telltale aberrations in the voltage levels indicating the presence of hidden data. In other words, judging by state of the art indicators [38] (see §7) we show it is equally plausible that a given device does or does not hold hidden data.

Finally, we assume that flash block wear in the device is not entirely equal, as is the case in many flash wear leveling policies [70–72].

5.3 Hiding Techniques

We now describe the data hiding algorithm in detail.

Normal data and hidden data are stored by two separate programming passes. The normal data is first programmed into a flash page, using standard flash operations. The hidden data will be programmed to the same pages in a second programming pass. First, a subset of the cells in a given page are selected to store hidden data, then a second encoding pass is done to store the hidden bits. Algorithm 1, as well as the following text, describe our encoding process.

Hidden cell selection. To select cells to store hidden data, we use a pseudo-random number generator (PRNG), such as SHA-256, that produces a set of random numbers based on a key—in our case, a key known only to the HU. We note that the HU does not explicitly persist the location of cells containing hidden data, but rather uses a deterministic PRNG function to calculate the map during boot time. In order to ensure an equal distribution of bit values, VT-HI encrypts hidden data, not unlike standard SSD controller data scrambling [32].

We only select non-programmed (i.e., "1") bits from the public data in a page to store hidden data. We remind the reader that flash cells typically use low voltage levels to store a "1", and raise the voltage to store a "0". We found that it is easier to reliably make small adjustments to the voltage levels of non-programmed cells than programmed cells; we believe that a flash vendor could use either type of cell in a production prototype.

In selecting a cell, the PRNG gives a page-dependent offset, such as the 3rd non-programmed bit in a specific flash page (e.g., by combining the secret key with the page number). This bit is then selected to be programmed with hidden data.

In order to store Error Correcting Codes (ECC) to tolerate bit errors, we select more cells for hidden data than the bits we wish to write.

We note that this technique spreads wear from extra programming evenly across cells over time, as which physical cell is programmed or not will vary over time, as will the output of the PRNG. We further assume public data is encrypted and bit values will be uniformly distributed. In practice, one could adopt more general wear-leveling techniques for hidden data if needed.

Storing Hidden Data. Figure 5 illustrates voltage-level encoding for hidden data in VT-HI. It starts by showing the voltage level distributions for a non-programmed ("1") cell: any voltage level less than about 127 is considered a public "1". Anything higher is a "0". We hide data by selecting a cut-off for hidden values of about 34, which is where most public voltages naturally occur.

To program a hidden "0", one must use a series of up to m partial programming (PP) steps, until the voltage is comfortably above the hidden data threshold. This PP process programs hidden data cells in an intermediate voltage level by iteratively reading and minutely incrementing the voltage level until the target threshold is reached. As with MLC or TLC flash, writes with this iterative technique are slower, but more precise.

A hidden "1" is not programmed. In the small chance that a cell should store a "1" but happens to be above the threshold, we treat this as a bit error and rely on ECC to

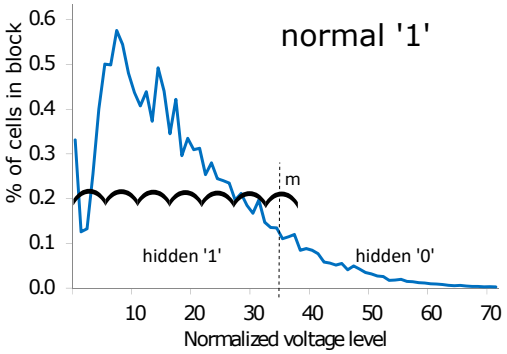


Figure 5: VT-HI hides data in the voltage level distribution of non-programmed cells, which store a normal '1'.

recover the data.

An important property of this design is that public data can be read with no awareness of hidden data or private key. This is because cells that store hidden data stay within the expected voltage levels for the public bit. To read hidden bits, the HU uses her key to calculate the indices of cells holding hidden bits, and reads them using the reference threshold voltage, which is placed in the middle of the two voltage intervals of the hidden bit states.

6 Implementation

In this section we describe the implementation of VT-HI on real hardware. We describe the hardware platform used in our experiments, and determine the configuration parameters for our hiding technique. Finally, we explain how the capacity of VT-HI can be extended through vendor support.

6.1 Experimental Hardware

Any implementation of our technique involves chip-specific configurations. To test them we used the same 1x-nm planar MLC flash used in Section §4. Each flash package has 8GB total storage capacity and contains 2048 blocks. A block consists of 128 lower pages and 128 upper pages with page size of 18048 bytes. The samples have a specified lifetime of 3000 PEC. Read, write (program), and erase latencies are 90 us, 1200 us, and 5 ms, respectively; the energy required for each operation is 50 uJ, 68 uJ, and 190 uJ, respectively.

The flash packages were operated using a commercial NAND flash tester [73]. Voltage level characterization of cells as well as the hiding algorithm were implemented as host software on a PC, which communicates with the tester via a USB interface. Throughout this section our calculations do not take into account data transfer and hardware overheads, which would be considerably lower on a production deployment. The specific voltage threshold (level 34) used for implementing our technique was determined empirically to tolerate the overshooting/underprogramming errors caused by the imprecise PP operation. We also verified that the total number of cells in the range is larger than the total number of hidden bits.

6.2 Vendor Support

Flash vendors are notoriously secretive about the internals of their devices. In order to collect the data presented in this paper, some co-authors of this paper signed a non-disclosure agreement (NDA) with a flash vendor. The NDA prohibits disclosing which vendor or the specific chips. In exchange we were given enough information to use a non-public command on the chip to measure voltage levels of cells, as well as issue partial programming (PP) commands to specific cells. To the best of our knowledge, the operations we use are generally implemented on any flash device, but the particular command encoding details vary from chip to chip, and are not made public. The NDA does not prohibit release of this data.

In principle, our prototype represents the most that a user could accomplish via reverse engineering a flash device, or using a flash device that openly published all available commands. Our results indicate the feasibility of the idea, with no changes to the flash controller. In the rest of this subsection, we explain how a few simple changes to a flash controller or the FTL firmware would improve the results we report.

First and foremost, PP is less precise than a program command issued by the controller. This is also the reason we select only non-programmed cells to store hidden data; PP is too coarse to reliably make fine-grained changes to programmed cells. We believe that an in-controller implementation of voltage hiding could likely program hidden data in fewer programming steps, saving energy and wear on the device, and opening up data hiding in both programmed and non-programmed cells.

Another feature not available to us was the ability to dynamically adjust voltage thresholds and targets [21–26]. The ability to control voltage targets and the width of voltage intervals might improve our hiding technique since narrower voltage intervals have been shown to easily fit into wider programming intervals [74] (e.g., TLC in MLC). This feature is generally available to the controller internally.

A limitation resulting from the lack of a more precise programming mechanism and the inability to adjust target voltage levels is that we found it difficult to reliably hide data in MLC or TLC modes using partial programming. We expect that a flash controller can extend our ideas to MLC or TLC, but the PP command on our test device was too coarse for this experiment to correctly store hidden data, and tended to disrupt public bits. Recall that a goal of our design is that one can read public data without any awareness of private, hidden data. Our

measurements indicate that, with more precise programming steps and/or the ability to adjust voltage thresholds slightly, our approach should extend to MLC or TLC. We note that existing flash page architectures regularly use a second fine-grained programming pass that does not significantly add interference to flash cells, and is this less detrimental to the bit rate as PP steps [32, 69]

6.3 Determining Capacity

In this subsection, we explore the potential capacity of our suggested hiding scheme, i.e., how many hidden bits we can store using VT-HI. This is a function of several concerns: over-provisioning bits to correct for errors (i.e., ECC), ensuring that the overall distribution of voltage values is not significantly perturbed (ensuring hidden data remains hidden), and minimizing the risk of inducing errors on neighboring cells or pages.

In order to keep the space overhead for ECC low, when determining configuration parameters for VT-HI we attempt to minimize the standard metric of bit-error rate (BER). We measure BER by encoding a hidden message in multiple blocks, physically located in different areas of the same chip. The message contains random content, both to emulate an encrypted hidden message, and to ensure that the charge levels in cells storing hidden bits have no anomalous effect on the overall distribution. After decoding, the message is compared with the original to determine the resulting error rate.

To find the optimal method and parameter values that minimize the hidden data BER (i.e., improve the effective data capacity) without compromising security, we systematically investigated each possible combination of three key parameters: number of partial programming steps, number of hidden bits per page, and page interval. The number of steps can be taken as a rough upper bound on write performance—fewer steps means faster hidden data writes, but more steps may be required to ensure a target voltage is reached. In setting hidden bits per page, intuitively, adding more hidden bits will push the overall distribution of voltage levels higher. Page interval is the physical distance between two cells storing hidden data; when a cell in one page is partially programmed, it may cause interference on neighboring pages. Intuitively, partially programming too many adjacent cells can cause bit flips on nearby public cells; thus, we measure this risk as a function of average physical distance of hidden bits. Our experiments were performed on a fresh chip, to avoid any interference that might stem from previous write patterns and wear. For each combination of parameters we encoded hidden data in five different blocks, and measured the average hidden data BER after each PP step.

Figure 6 shows that after roughly ten PP steps the BER converges to less than 1%. This trend holds regardless of the number of hidden bits or the page interval.

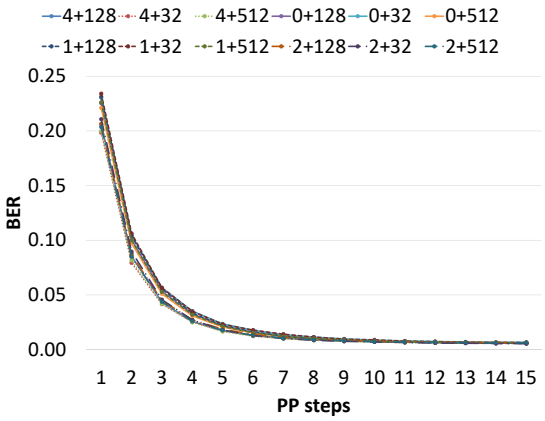


Figure 6: Hidden BER rates for VT-HI for the first fifteen steps in multiple combinations of page intervals and number of hidden bits.

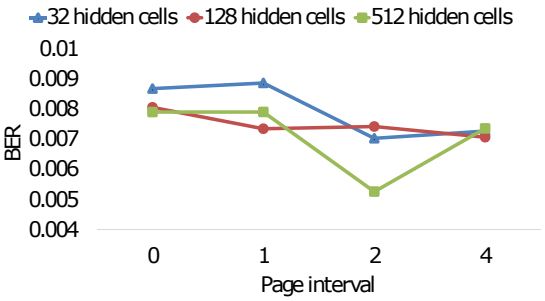


Figure 7: Hidden BER rates for VT-HI with ten PP steps. The illustrated irregularity demonstrates the effects of BER variance and program interference.

Figure 7 shows the sensitivity of BER for hidden data as a function of the number of hidden cells, using 10 PP steps to program the hidden data. Overall, the variation in bit error rate is small and generally insensitive to the number of hidden cells. There is some irregularity that is within the bounds of naturally occurring variance [32, 65, 66]. We do notice a small trend toward lower bit rates; because we only select unprogrammed cells for hiding data, any interference can flip cells that are slightly under-programmed (just short of the target threshold) to being just above the target threshold.

In this experiment, we selected 512 as an upper bound for the number of hidden bits. We measured a range of voltages for chips programmed with random data, and found that one could reliably get a minimum of 700 cells in the non-programmed state that are normally charged above our data hiding threshold. In other words, hiding more bits per page than 512 will likely leave telltale changes to the distribution of voltages.

Figure 7 indicates that, for any number of hidden bits in a page that satisfy our other constraints, the BER is small. The implication is that a small number of error-correcting hidden bits (e.g., 5%) will suffice.

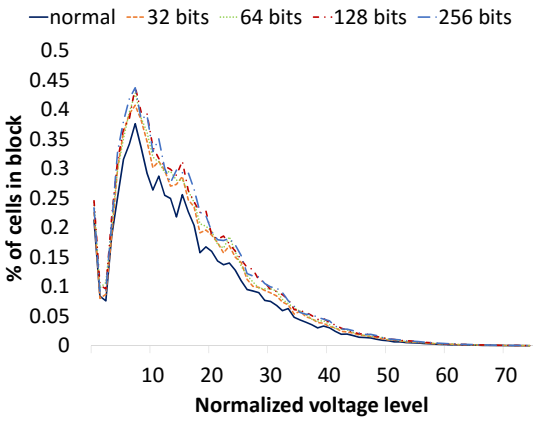


Figure 8: Average voltage level distributions for blocks after applying VT-HI. Hiding more bits creates a more noticeable shift to the right for non-programmed cells.

Figure 8 shows that hiding data using VT-HI creates only a tiny shift to the right for non-programmed cells. which can be attributed to normal voltage level distribution variability and errors, as well as small read disturbs and retention effects [32, 75]. Although we might be able to store 512 bits per page, we conservatively chose to hide 256 bits per page. In Section 7 we further explore the security of VT-HI using this configuration.

Finally, we measured the impact of page intervals on the BER for public data. Using no physical space between pages storing hidden data increased the public BER by 20%. At one physical page interval, the interference is reduced to a more acceptable 10% [64–66, 76]. Thus, subsequent experiments use a page interval of one.

7 Detectability

The primary criterion for evaluating VT-HI's success is whether an attacker with a full and detailed voltage-level analysis of the entire chip can infer whether some pages store hidden data from changes in voltage level distributions. In this section, we show that flash pages in VT-HI, with and without hidden data, cannot be distinguished.

Figure 9 illustrates the difficulty of detecting the existence of data hidden in VT-HI. The figure shows voltage level distributions from three blocks from different chips, first when they are normally programmed and then after applying VT-HI to hide data. The human eye has difficulty distinguishing which distributions come from blocks with hidden data.

SVM Analysis. Rather than rely on the human eye, we follow prior work [38] and instead use supervised machine learning to determine whether there are any detectable anomalies in the data. We use a support-vector machine (SVM) to predict whether pages and blocks contain hidden data. If VT-HI left aberrations in voltage levels that correlate with the presence of hidden data, an SVM would be able to identify these pages with better than 50% accuracy (i.e., better than flipping a coin). Our hypothesis is that the changes induced by data hiding are within normal noise.

To demonstrate this, we obtained data from three different hardware units of varying ages. We first measured the voltage level distributions and BER of three flash chips. For normal data characteristics of flash blocks and pages, we used normal programming for program/erase counts (PEC) ranging from 0 to 3000. We then hid data using VT-HI with the configuration parameters determined is §6 (threshold level 34, one page interval, 256 bits per page, ten PP steps) on all chips for blocks that were cycled to 0, 1000, and 2000 PEC.

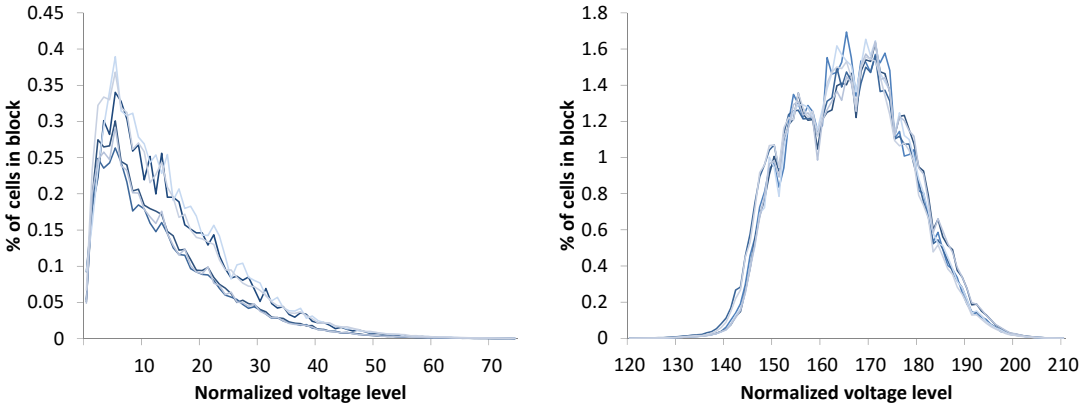
We created a training set for the SVM using datasets from two chips, and then we attempt to classify data from a third chip. For the training, we collected the voltage levels for all cells in the block with both normal and hidden data. We found that the flash chip data representativeness converged after analyzing 31 blocks. The classifier used optimal parameters obtained using grid search, and performed three-fold cross-validation for all three chips. As Wang et al. note [38], this is an unrealistically generous setup for the attacker. In reality, the attacker has to obtain knowledge of all possible PEC levels of the chip for both normal and hidden cases, and for multiple sample chips of the same vendor and model which would probably reduce the prediction accuracy.

In analyzing the voltage data we collected, the wear or number of program/erase cycles (PEC) had a first-order effect on the voltage levels.

This sensitivity to PEC is illustrated in Figure 10, which presents SVM accuracy for samples at PEC of 0, 1000, and 2000. The x-axis is the PEC of normal data. For each line, there is a range of a few hundred P/E cycles where the accuracy of the SVM is at 50% (or random). For example, consider the PEC 2000 line. At x-axis of PEC 2000 (comparing to the same wear without hidden data), the SVM does not do better than random (50%); for a few hundred cycles on either side of this point, the accuracy is still effectively 50%. At extremely about 1000 PEC, and as PEC increases the classifier's accuracy increases. Thus, we expect that, as long as the wear on the device is uniform within several hundred PEC, an SVM would not be able to reliably classify which blocks have hidden data and which do not. A similar experiment at the page-level shows similar results

We also note that this experiment deliberately places VT-HI at a disadvantage, by training the SVM on the exact chip that was storing the hidden data. We repeated our tests on a data set that includes all of the chips (from the same vendor) and all PEC levels, and this decreased the SVM accuracy to 50% in all cases.

Finally, one might be concerned that an attacker could draw inferences from changes in characteristics of pub-



(a) (b)

Figure 9: Voltage level distribution in blocks from different chips with normal distributions (light) and after applying VT-HI (dark). Results show (a) non-programmed and (b) programmed cells.

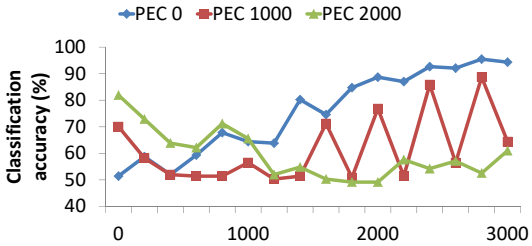


Figure 10: SVM prediction accuracy for block-level voltage distribution data for hidden blocks in three different PEC levels. Normal PEC (X axis) represents public data blocks in different PEC levels. 50% accuracy (dotted line) is equivalent to a random guess.

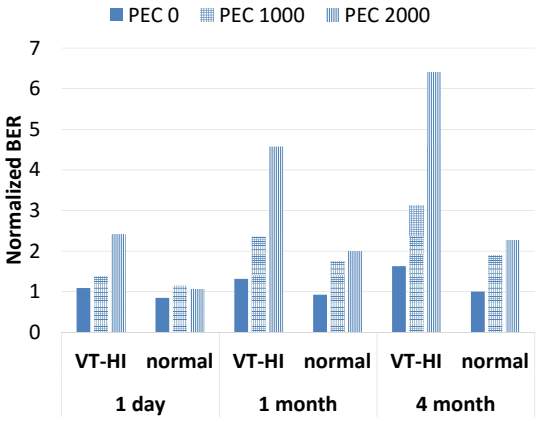


Figure 11: Normalized retention rate (versus "zero" time since programming) for data stored using VT-HI and normal data.

lic data, such as BER, mean voltage, and its standard deviation. Therefore, we performed another SVM analysis to classify blocks with and without hidden data according to these characteristics. Our results indicate that these analyses are also unsuccessful in classifying hidden data.

8 Performance and Applicability

In this section we evaluate and analyze the performance of VT-HI in terms of reliability, throughput, capacity, and energy. For each of these factors, we compare the performance of VT-HI to the most similar prior research paper, PT-HI, as summarized in Table 1. We also demonstrate that with proper configuration and vendor support we can increase the hidden data storage capacity of VT-HI by an order of magnitude. Finally, we verify the applicability of VT-HI on a chip from a second major vendor.

Reliability. As flash devices move toward smaller feature sizes, data errors will increase [77], increasing the importance of error-correcting codes and other countermeasures. Charges stored in flash memory cells gradually leak away over time, causing cell voltage to shift towards lower values. Bit errors accumulate in data as cell voltages shift across the predetermined reference threshold voltage. Here we characterize the reliability of bits hidden with VT-HI.

First, we measured the error rate for VT-HI in blocks with varying PEC levels, and find that BER is low and not affected by wear. We cycled blocks in three different chips to four distinct PEC levels. Next, we hid data using VT-HI, and measured the hidden data BER. Our results show that the BER is not affected by the age of the cells storing hidden data. For example, for PEC 0 the BER was 0.013. For other PEC the BER was roughly 0.011.

We also emulated data retention over longer periods by baking the flash chips in an oven, which accelerates the rate of charge leakage from the floating gates. Three data retention periods were used in our evaluation: 1 day, 1 month, and 4 months. The latter two periods were emulated by baking the flash [78]. Before retention, pseudorandom normal data and hidden data were first input

and stored in flash. BERs and voltage distributions were then measured after retention using the previously saved input data. We normalized the BER after the relevant retention period to the BER measured immediately after the data was stored ("zero" time). We compared the rate of changes in the BER to the equivalent rate for public data on our test chip.

The results shown in Figure 11 indicate that retention time has no significant effect on the BER of hidden data for fresh cells (PEC 0). However, the hidden data error rate does increase on older cells, at a higher rate than for public data

For example, for 2000 PEC, the BER after zero time is 0.0099, and rises to 0.063 (6.3x) after four months, while for normal data the BER rises from 0.00003 to 0.000075 (2.3x). The reason for this reduced retention is that cells with higher PEC accumulate trapped charge and become more sensitive to leakage [77]. Moreover, hidden data BER degrades faster than public data BER. The reason is that the programming technique available to VT-HI (PP steps) is not accurate enough to ensure a large buffer zone around the threshold voltage level. Such zones are typically employed to minimize BER in degraded cells in existing flash package programming schemes (see X axis in Figures 2a and 2b), which are also used for storing public data in VT-HI.

These results indicate that additional redundancy would be prudent when hiding data in older cells. Rewriting (refreshing) hidden data every several months, even only after the device reaches 1K PEC, can also significantly improve retention [79]. Finally, to provide additional protection against data loss (e.g., due to bad blocks) data can be further encoded using RAID-like schemes, similarly to normal data [80].

Throughput. Here, we calculate the expected read and write throughput from VT-HI and our closest competitor, PT-HI, using reasonable parameters from current flash chips. We find that VT-HI can deliver an order of magnitude better throughput for hidden data than the best possible configuration for PT-HI.

Under VT-HI's optimal configuration (256 hidden bits per page and 4 logical page intervals), we can estimate the time it would take to encode hidden data in a block: $(600+90)\cdot 10\cdot 64/1,000,000 = 0.44s$ with a PP time of 600 us and read time of 90 us for 10 PP and read steps, and 64 pages per block. Assuming 15,593 hidden data bits per block, this translates to a throughput of 35Kb/s. This figure takes into account a 0.5% hidden BER, which, after applying standard ECC codes, translates to 243.6 bits of data per page (i.e., ≈13 parity bits).

We repeated this calculation for PT-HI, assuming its optimal setup with a negligible hidden data error rate. We use the optimal configuration in [38] of 625 per-page PP steps and a 4-page interval for hiding data, which translates to 72Kb of hidden bits per block. The page program latency used is 1.2 ms and block erase latency is 5 ms. In this setup, the time it would take to write hidden data is $(1.2\cdot 64+5)\cdot 625/1,000 = 51.1s$ per block. Therefore, even for this ideal setup, the optimal throughput for PT-HI is only 1.4Kb/s. We note that PT-HI's performance dramatically deteriorates in setups where the device has undergone even a few hundred PEC due to its increasing BER. We also note that PT-HI wears out the device much faster than VT-HI since it requires 60x more programming steps in order to encode data.

Method	*Reliability*	*Perf.*	*Power*	*Public data integrity*	*Repeated Reads*	*Capacity*
PT-HI [38]	±	-	-	+	-	±
VT-HI	+	±	±	-	+	±

Table 1: Our contribution compared to Wang et al. [38].

Decoding hidden data that was encoded using VT-HI in a page requires only a single read operation (following a voltage reference shift command). This translates to $90\cdot 64\cdot 1/1,000,000 = 0.006s$ for decoding the data hidden in a block, and a throughput of 2.7 Mb/s. For PT-HI, 30 PP and read operations are required to decode data from a page. This translates to $(600+90)\cdot 64\cdot 30/1000000 = 1.32s$ for decoding the hidden data in a block, and a throughput of 54 kb/s.

Improved Capacity. As we explain in §6.2, our prototype can only reliably encode data in non-programmed cells, which we found to keep us to under 700 bits per page to avoid telltale disruptions in the distribution of voltages. We conjecture that, with controller-internal programming tools, we could apply the same basic idea to a larger number of cells, which should potentially increase hidden data capacity.

In this section, we evaluate the impact on the risk of detection when more non-programmed cells are used. We repeat the SVM analysis in §7. We emulate finer-grained programming by using a single PP step ($m = 1$) instead of ten, and increase the hidden bits per page by a factor of ten. We then adjust the hidden data voltage threshold to level 15 to keep the voltage levels of cells with hidden data within the expected distribution. We kept the page interval the same (1 physical page).

Figure 12 shows the SVM accuracy results for our simulated higher-capacity configuration, on block-level data. Similarly to the results for hiding 256 bits, the results are highly sensitive to PEC. If we only consider ranges where the hidden and non-hidden blocks have PEC within a few hundred cycles of each other, the accuracy is generally low (50–60%), but slightly higher than the other experiment. Some of the increased accuracy is attributable to the lack of precision in PP, especially when only a single step is used.

Hidden data BER for data in the enhanced VT-HI configuration was only 2%. After applying standard ECC

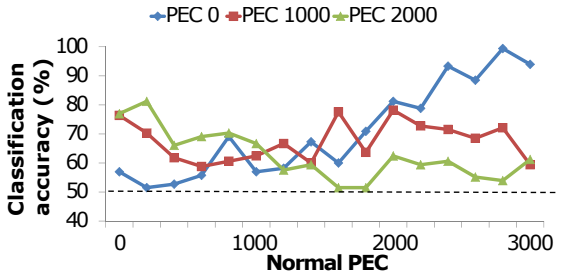


Figure 12: SVM prediction accuracy for an enhanced VT-HI configuration that hides 9x more data. Normal PEC (X axis) represents public data blocks in different PEC levels. 50% accuracy (dotted line) is equivalent to a random guess.

codes this translates to 2197 of data bits per page (14% are used for ECC). After accounting for ECC to mask the increased BER, this represents a 9x increase in usable hidden data capacity (and twice as much as PT-HI).

Energy. For our chip, we estimated the energy required for various data encoding operations using VT-HI and PT-HI (again, in an ideal setup). These include read, program and erase operations, as well as partial programming. We then used these estimated values to calculate the amount of energy required for writing a bit of hidden data. The results show that for VT-HI the energy required for hiding data is 1.1 mJ per page, as opposed to 43 mJ for PT-HI. This data indicates that, if an adversary read two snapshots of the device energy usage statistics, effectively there would not be a telltale difference for VT-HI and a system without hidden data. For instance, the energy overhead of our PP-based is less impactful than, say, extra reads from the device. With an in-controller VT-HI implementation we expect energy overheads could be reduced further.

Applicability. Finally, to verify that our method also applies to other flash chip models, we tested it on a 1x-nm 16GB MLC chip model from a different major vendor (also under a similar NDA). The flash package contains 2096 blocks, with page size of 18256 bytes. We tested our method on a fresh chip (PEC 0) and hid a 256 bit payload in relevant pages (taking into account architecture-specific page intervals). The resulting BER was 1%, similar to the one in the first model.

9 Discussion

This section discusses various applications for which VT-HI could be a useful building block.

9.1 Authentication and Provenance

One property of our approach is that erasing a block of public data on the flash device (thereby de-charging the cells) also erases any hidden payload in the cells. This property does not imply that a user cannot modify normal data; such modifications simply require the user to repeat the hiding process with the same hidden data on newly written normal data.

Many applications require some form of *proof to the trustworthiness and provenance* of their data. A number of systems find ways to embed a signature or metadata in the data file itself. VT-HI could be incorporated into these systems to embed metadata in the physical pages storing this data; only a trusted application can rewrite a file and embed hidden metadata in the device. For example, flash chip steganography enables counterfeit detection by watermarking original parts [38]. Archival storage systems authenticate the identity of data objects [81]. Embedded watermarks in storage media identify ownership of digital objects to prevent copyright infringements [11]. Secure file systems persist the keys required for accessing data to their storage media [59].

9.2 Steganography

VT-HI can also be used as a building block for implementing a steganographic system [51, 57–60]. Implementing a complete steganographic system is beyond the scope of this paper, but, in the interest of brevity, we discuss the main challenges of such a solution.

Basic Design. A VT-HI-capable system would include a publicly visible, encrypted volume, within which a user can store a hidden, encrypted data volume. To access the hidden volume, a user would input the secret key at mount time. Data can then be read and written from this volume using standard block-level operations.

The security of the hidden volume stems from the security of VT-HI. An attacker that inspects the device once, including all low-level characteristics, will not be able to differentiate flash pages that contain hidden data from those that do not, without the secret key.

Hiding VT-HI. The presence of a VT-HI-capable SSD may still raise the suspicion of an adversary that data is hidden. This problem is common to many existing systems [60], and can be mitigated in several ways. First, we can further assume that firmware update capability is available to the user via secure channels, so the VT-HI capability can be loaded whenever the user accesses hidden data and then immediately removed. Alternatively, the VT-HI capability can be included by default as an extension of open-source SSD firmware [82], allowing users to configure the firmware at will to operate with and without hiding capabilities.

Metadata Persistence and Security. VT-HI relies on configuration metadata, such as m, Vth, and the number of bits per page, which must be persisted and recovered on bootstrap. Because the metadata is small, the metadata could be included in the hidden key. Alternatively, the metadata can be encrypted and stored persistently in

predetermined locations on flash, or, similarly to the hiding firmware itself, saved and reloaded from an external source. From a security standpoint, the metadata configuration values for a specific chip model may be known to a diligent adversary. However, even with full knowledge of the configuration metadata, without the secret key the adversary is still unaware of the location of cells containing hidden bits and cannot recover them.

Other metadata persistence issues, such as recovering the hidden volume LBA for every set of pages, may require sacrificing some hidden capacity or more sophisticated mapping data structures and algorithms, which we leave as future work.

Multiple-Snapshot Adversary. A stricter threat model involves an adversary capable of comparing multiple snapshots of the device taken over time. In this case, storing hidden data while leaving the public data unchanged leaves telltale signs of voltage manipulations that prevent users from plausibly denying the existence of hidden data. To mitigate, the hiding firmware can piggyback either public data writes (similar to [58]). Alternatively, the hiding firmware can utilize wear-leveling and other SSD-internal activities [61, 79], to create the requisite cover traffic. A trade-off here is that firmware-internal bookkeeping which operates without the private key for too long will eventually damage hidden data by causing internal data movements that copy data without also copying the hidden payload. We note however that hidden data overwrites when operating the system without the hidden key is an inherent limitation of almost all existing steganographic systems [60].

Capacity. The current implementation of VT-HI can only hide a few hundred bits per flash page. We believe that many privacy-concerned users will find the strong deniability offered by VT-HI as a reasonable tradeoff for reduced capacity. Also, in §6.2 we explain how vendor support may significantly alleviate this limitation (e.g., hide data as TLC in MLC cells).

10 Conclusions

In this work we present a new method for hiding data in flash using the inherent variability in voltage level distributions of flash cells. This variation occurs naturally on flash chips, even from the same vendor and model. We manipulate the voltage levels in cells to hide data within normal voltage intervals. Our manipulations hide an additional hidden bit in cells that already store a public bit by mimicking common methods to increase flash densities. Without the hiding key, an attacker cannot detect cells with hidden data even using favorable supervised learning. In comparison with the state of the art, our method achieves respectively 24x and 50x improvement for encoding and decoding throughput of hidden data, and is 37x more power efficient. Our technique is applicable to multiple chip models, allows users to store data even on flash cells that endured significant wear, and imposes significantly less wear while doubling total hidden capacity compared with prior work.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments on earlier drafts of the work. This research was supported by Grant 2014621 from the United States-Israel Binational Science Foundation (BSF), by Grant CNS-1526707 from the United States National Science Foundation (NSF), and VMware. This work was done in part while Porter was at Stony Brook University.

References

[1] Kingston. Nearly half of organizations have lost sensitive or confidential information on USB drives in just the past two years. http://www.kingston.com/en/company/press/article/2661, 2011.

[2] Independent. BBC's panorama team loses confidential information relating to a secret british army unit. http://www.independent.co.uk/news/uk/home-news/exclusive-bbcs-panorama-team-loses-confidential-information-relating-to-a-secret-british-army-unit-9580340.html, 2014.

[3] WCSH6. USB drive containing personal information of 950 jetport workers, missing. http://www.wcsh6.com/news/local/portland/usb-drive-containing-personal-information-of-950-jetport-workers-missing/251514955, 2016.

[4] Computer World. NASA breach update: Stolen laptop had data on 10,000 users. http://www.computerworld.com/article/2493084/security0/nasa-breach-update--stolen-laptop-had-data-on-10-000-users.html, 2012.

[5] BBC News. Blackmail fear over lost raf data. http://news.bbc.co.uk/2/hi/uk/8066586.stm, 2009.

[6] The Register. Youth jailed for not handing over encryption password. http://www.theregister.co.uk/2010/10/06/jail_password_ripa/, 2010.

[7] Wikipedia. Key disclosure law. http://en.wikipedia.org/wiki/Key_disclosure_law.

[8] Denver Post. Password case reframes fifth amendment rights in context of digital world. http:

//www.denverpost.com/news/ci_19669803, 2012.

[9] PCWorld. Prepare to take your laptop to another country. http://www.pcworld.com/article/2886367/prepare-to-take-your-laptop-to-another-country.html, 2015.

[10] FBI. Economic espionage. https://www.fbi.gov/about-us/investigate/counterintelligence/economic-espionage.

[11] Ingemar J Cox, Matthew L Miller, Jeffrey Adam Bloom, and Chris Honsinger. *Digital watermarking*, volume 1558607145. Springer, 2002.

[12] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using IC fingerprinting. In *IEEE Symposium on Security and Privacy*, 2007.

[13] Malek Ben Salem. Security challenges and requirements for industrial control systems in the semiconductor manufacturing sector. NIST Workshop on Cyber-Security for Cyber-physical Devices, 2012.

[14] Microsoft Corporation. Windows BitLocker drive encryption frequently asked auestions. http://technet.microsoft.com/en-us/library/cc766200%28WS.10%29.aspx, 2009.

[15] OS X mavericks: Encrypt the information on your disk with filevault. http://support.apple.com/kb/PH13729.

[16] Yinglei Wang, Wing kei Yu, Shuo Wu, G. Malysa, G.E. Suh, and E.C. Kan. Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints. In *IEEE Symposium on Security and Privacy (SP)*, 2012.

[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, pages 2–2. USENIX Association, 2012.

[18] Laura M. Grupp, John D. Davis, and Steven Swanson. The harey tortoise: Managing heterogeneous write performance in SSDs. In *Proceedings of the USENIX Conference on Annual Technical Conference*, USENIX ATC, 2013.

[19] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, Jung-Hyuk Choi, Jang-Rae Kim, and Hyung-Kyu Lim. A 3.3 v 32 Mb NAND flash memory with incremental step pulse programming scheme. *IEEE Journal of Solid-State Circuits*, 30(11):1149–1156, Nov 1995.

[20] The Register. Good gravy, Toshiba QLC flash chips are getting closer. http://www.theregister.co.uk/2016/07/18/tosh_qlc_flash_chips_getting_closer, 2016.

[21] Taeho Kgil, D. Roberts, and T. Mudge. Improving NAND flash based disk caches. In *35th International Symposium on Computer Architecture (ISCA*, 2008.

[22] H. Nagashima, T. Tanaka, K. Kawai, and K. Quader. Nonvolatile semiconductor memory device which uses some memory blocks in multi-level memory as binary memory blocks, August 3 2006. US Patent App. 11/391,299.

[23] F. Yu, A.C. Ma, S. Chen, and Y.T. Chang. Endurance and retention flash controller with programmable binary-levels-per-cell bits identifying pages or blocks as having triple, multi, or single-level flash-memory cells, January 2 2014. US Patent App. 13/788,989.

[24] S.C. Wong and K. Johnsen. Data coding for multi-bit-per-cell memories having variable numbers of bits per memory cell, October 15 2002. US Patent 6,466,476.

[25] N.J. Wakrat and T.M. Toelkes. Dynamically allocating number of bits per cell for memory locations of a non-volatile memory, March 19 2013. US Patent 8,402,243.

[26] Seungjae Lee, Young-Taek Lee, Wook-Kee Han, Dong-Hwan Kim, Moo-Sung Kim, Seung-Hyun Moon, Hyun Chul Cho, Jung-Woo Lee, Dae-Seok Byeon, Young-Ho Lim, Hyung-Suk Kim, Sung-Hoi Hur, and Kang-Deog Suh. A 3.3 v 4 Gb four-level NAND flash memory with 90 nm CMOS technology. In *IEEE International Solid-State Circuits Conference*, ISSCC, 2004.

[27] Micron eMMC Linux enablement - SLC mode. https://prod.micron.com/~/media/documents/products/technical-note/emmc/tn5205_emmc_linux_enablement.pdf, 2012.

[28] Anandtech. Transcend announces SuperMLC: Pseudo-SLC SSDs for industrial market. http://www.anandtech.com/show/9882/transcend-announces-supermlc-pseudoslc-ssds-for-industrial-market, 2015.

[29] Electronic Design. Pseudo-SLC flash provides design flexibility. http://electronicdesign.com/site-files/electronicdesign.com/files/uploads/2013/09/FAQs-Toshiba-September.pdf, 2013.

[30] Tom's hardware. JMicron JMF670H SSD controller preview. http://www.tomshardware.com/reviews/jmicron-jmf670h-ssd-controller,4161.html, 2015.

[31] Open NAND flash interface. http://www.onfi.org. 2016.

[32] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash memory based solid-state drives. *arXiv preprint arXiv:1706.08642*, 2017.

[33] Y. Cai, Y. Luo, E. F. Haratsch, K. Mai, and O. Mutlu. Data retention in mlc nand flash memory: Characterization, optimization, and recovery. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA, 2015.

[34] Lorenzo Zuolo, Cristian Zambelli, Rino Micheloni, Marco Indaco, Stefano Di Carlo, Paolo Prinetto, Davide Bertozzi, and Piero Olivo. SSDexplorer: A virtual platform for performance/reliability-oriented fine-grained design space exploration of solid state drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1627–1638, 2015.

[35] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis, and modeling. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2013.

[36] Toshiba's 768gb 3D QLC NAND flash memory: Matching TLC at 1000 P/E cycles? Anandtech, https://www.anandtech.com/show/11590/toshiba-768-gb-3d-qlc-nand-flash-memory-1000-p-e-cycles.

[37] Western digital unveils 96-layer nand, 4-bit qlc breakthrough. Extremetech, https://www.extremetech.com/extreme/251774-western-digital-announces-new-96-layer-nand-4-bit-qlc-breakthrough.

[38] Yinglei Wang, Wing kei Yu, S.Q. Xu, E. Kan, and G.E. Suh. Hiding information in flash memory. In *IEEE Symposium on Security and Privacy (SP)*, pages 271–285, 2013.

[39] Pravin Prabhu, Ameen Akel, Laura M. Grupp, Wing-Kei S. Yu, G. Edward Suh, Edwin Kan, and Steven Swanson. Extracting device fingerprints from flash memory by exploiting physical variations. In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, pages 188–201, 2011.

[40] Amir Rahmati, Matthew Hicks, and Atul Prakash. Approximate flash storage: A feasibility study. Presented at the Workshop on Approximate Computing Across the Stack, 2016.

[41] Abbas Cheddad, Joan Condell, Kevin Curran, and Paul Mc Kevitt. Digital image steganography: Survey and analysis of current methods. *Signal Processing*, 90(3):727 – 752, 2010.

[42] Fabien A.P. Petitcolas, R.J. Anderson, and M.G. Kuhn. Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062–1078, Jul 1999.

[43] A. Swaminathan, Min Wu, and K.J.R. Liu. Digital image forensics via intrinsic fingerprints. *IEEE Transactions on Information Forensics and Security*, 3(1):101–117, March 2008.

[44] Wojciech Mazurczyk. VoIP steganography and its detection - a survey. *ACM Computing Surveys (CSUR)*, 46(2):20, 2013.

[45] Wojciech Fraczek, Wojciech Mazurczyk, and Krzysztof Szczypiorski. Hiding information in a stream control transmission protocol. *Computer Communications*, 35(2):159 – 169, 2012.

[46] Bernard B. Wu and Evgenii E. Narimanov. Analysis of stealth communications over a public fiber-optical network. *Opt. Express*, 15(2):289–301, Jan 2007.

[47] P.R. Prucnal, M.P. Fok, K. Kravtsov, and Zhenxing Wang. Optical steganography for data hiding in optical networks. In *16th International Conference on Digital Signal Processing*, ICDSP, 2009.

[48] M.P. Fok, Zhexing Wang, Yanhua Deng, and P.R. Prucnal. Optical layer security in fiber-optic networks. *IEEE Transactions on Information Forensics and Security*, 6(3):725–736, Sept 2011.

[49] Qian Wang, Kui Ren, Guancheng Li, Chenbo Xia, Xiaobing Chen, Zhibo Wang, and Qin Zou. Walls have ears! opportunistically communicating secret messages over the wiretap channel: From theory to practice. In *Proceedings of the 22Nd Conference on Computer and Communications Security*, CCS, 2015.

[50] X. Zhou, HweeHwa Pang, and K.-L. Tan. Hiding data accesses in steganographic file system. In *Proceedings of the 20th International Conference on Data Engineering*, 2004.

[51] Ross Anderson, Roger Needham, and Adi Shamir. The steganographic file system. In *Information Hiding*, volume 1525 of *Lecture Notes in Computer Science*, pages 73–82. Springer Berlin Heidelberg, 1998.

[52] Jin Han, Meng Pan, Debin Gao, and HweeHwa Pang. A multi-user steganographic file system on untrusted shared storage. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC, 2010.

[53] Kefa Rabah. Steganography-the art of hiding data. *Information Technology Journal*, 3:245–269, 2004.

[54] Andrew D. McDonald and Markus G. Kuhn. StegFS: A steganographic file system for linux. In *Information Hiding*, volume 1768 of *Lecture Notes in Computer Science*, pages 463–477. Springer Berlin Heidelberg, 2000.

[55] HweeHwa Pang, K.-L. Tan, and X. Zhou. Stegfs: a steganographic file system. In *Proceedings of the 19th International Conference on Data Engineering*, ICDE, 2003.

[56] Open Crypto audit project. `http://opencryptoaudit.org/`.

[57] Adam Skillen and Mohammad Mannan. On implementing deniable storage encryption for mobile devices. In *Network & Distributed System Security Symposium*, NDSS, 2013.

[58] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 203–214, New York, NY, USA, 2014. ACM.

[59] Timothy M Peters, Mark A Gondree, and Zachary NJ Peterson. Defy: A deniable, encrypted file system for log-structured storage. In *The Network and Distributed System Security Symposium*, NDSS, 2015.

[60] Aviad Zuck, Udi Shriki, Donald E. Porter, and Dan Tsafrir. Preserving Hidden Data with an Ever-Changing Disk. In *ACM Workshop on Hot Topics in Operating Systems*, HotOS, 2017.

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

[62] Anxiao Jiang, V. Bohossian, and J. Bruck. Rewriting codes for joint information storage in flash memories. *IEEE Transactions on Information Theory*, 56(10):5300–5313, Oct 2010.

[63] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2009.

[64] Y. Di, L. Shi, K. Wu, and C. J. Xue. Exploiting process variation for retention induced refresh minimization on flash memory. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016.

[65] Y. Pan, G. Dong, and T. Zhang. Error rate-based wear-leveling for NAND flash memory at highly scaled technology nodes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(7):1350–1354, 2013.

[66] Yeong-Jae Woo and Jin-Soo Kim. Diversifying wear index for MLC NAND flash memory to extend the lifetime of SSDs. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT, 2013.

[67] Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Thomas Mittelholzer, Evangelos Eleftheriou, Charles Camp, Thomas Griffin, Gary Tressler, and Andrew Walls. Using adaptive read voltage thresholds to enhance the reliability of MLC NAND flash memory systems. In *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*, GLSVLSI, 2014.

[68] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai. Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation. In *2013 IEEE 31st International Conference on Computer Design*, ICCD, 2013.

[69] Ki-Tae Park, Myounggon Kang, Doogon Kim, Soon-Wook Hwang, Byung Yong Choi, Yeong-Taek Lee, Changhyun Kim, and Kinam Kim. A zeroing cell-to-cell interference page architecture with temporary LSB storing and parallel msb program scheme for MLC NAND flash memories. *IEEE Journal of Solid-State Circuits*, 43(4):919–928, April 2008.

[70] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, USENIX ATC, 2008.

[71] M. Murugan and D.H.C. Du. Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead. In *IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST, 2011.

[72] Y. J. Woo and J. S. Kim. Diversifying wear index for MLC NAND flash memory to extend the lifetime of SSDs. In *Proceedings of the International Conference on Embedded Software*, EMSOFT, 2013.

[73] NAND flash memory tester (SigNASII). http://www.siglead.com/eng/innovation_signas2.html. 2016.

[74] Jong-Ho Park, Sung-Hoi Hur, Joon-Hee Leex, Jin-Taek Park, Jong-Sun Sel, Jong-Won Kim, Sang-Bin Song, Jung-Young Lee, Ji-Hwon Lee, Suk-Joon Son, Yong-Seok Kim, Min-Cheol Park, Soo-Jin Chai, Jung-Dal Choi, U-In Chung, Joo-Tae Moon, Kyeong-Tae Kim, Kinam Kim, and Byung-Il Ryu. 8 gb MLC (multi-level cell) NAND flash memory using 63 nm process technology. In *Technical Digest of IEEE International Electron Devices Meeting*, IEDM, pages 873–876, 2004.

[75] Y. Cai, Y. Luo, S. Ghose, and O. Mutlu. Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery. In *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2015.

[76] Wei Wang, Tao Xie, and Deng Zhou. Understanding the impact of threshold voltage on mlc flash memory performance and reliability. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS, 2014.

[77] Yue Li, Eyal En Gad, Anxiao Jiang, and Jehoshua Bruck. Data archiving in 1x-nm NAND flash memories: Enabling long-term storage using rank modulation and scrubbing. In *2016 IEEE 54th International Reliability Physics Symposium*, 2016.

[78] Mingzhen Xu, Changhua Tan, and MingFu Li. Extended Arrhenius law of time-to-breakdown of ultrathin gate oxides. *Applied Physics Letters*, 82(15):2482–2484, Apr 2003.

[79] Yu Cai, G. Yalcin, O. Mutlu, E.F. Haratsch, A. Cristal, O.S. Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *IEEE 30th International Conference on Computer Design*, ICCD, 2012.

[80] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tinytail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies*, FAST, 2017.

[81] Michael Factor, Ealan Henis, Dalit Naor, Simona Rabinovici-Cohen, Petra Reshef, Shahar Ronen, Giovanni Michetti, and Maria Guercio. Authenticity and provenance in long term digital preservation: Modeling and implementation in preservation aware storage. In *First Workshop on Theory and Practice of Provenance*, TAPP, 2009.

[82] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies*, FAST, 2017.

Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree

Deukyeon Hwang
UNIST

Wook-Hee Kim
UNIST

Youjip Won
Hanyang University

Beomseok Nam
UNIST

Abstract

With the emergence of byte-addressable persistent memory (PM), a cache line, instead of a page, is expected to be the unit of data transfer between volatile and non-volatile devices, but the failure-atomicity of write operations is guaranteed in the granularity of 8 bytes rather than cache lines. This granularity mismatch problem has generated interest in redesigning block-based data structures such as B+-trees. However, various methods of modifying B+-trees for PM degrade the efficiency of B+-trees, and attempts have been made to use in-memory data structures for PM.

In this study, we develop Failure-Atomic ShifT (FAST) and Failure-Atomic In-place Rebalance (FAIR) algorithms to resolve the granularity mismatch problem. Every 8-byte store instruction used in the FAST and FAIR algorithms transforms a B+-tree into another consistent state or a *transient inconsistent* state that read operations can tolerate. By making read operations tolerate transient inconsistency, we can avoid expensive copy-on-write, logging, and even the necessity of read latches so that read transactions can be non-blocking. Our experimental results show that legacy B+-trees with FAST and FAIR schemes outperform the state-of-the-art persistent indexing structures by a large margin.

1 Introduction

Recent advances in byte-addressable persistent memories (PM), such as phase change memory[48], spin-transfer torque MRAM[17], and 3D Xpoint[2] have opened up new opportunities for the applications to warrant durability or persistency without relying on legacy heavy-duty interfaces offered by the filesystem and/or by the block device [21, 24].

In legacy block devices, B+-tree has been one of the most popular data structures. The primary advantage of a B+-tree is its efficient data access performance due to its high degree of node fan-out, a balanced tree height, and dynamic resizing. Besides, the large CPU cache in modern processors[11] enables B+-tree to exhibit a good cache line locality. As a result, B+-tree shows good performance as an in-memory data structure as well. Thus, the cache-conscious variants of B+-tree including CSS-tree [36], CSB-tree [37], and FAST [19] are shown to perform better than legacy binary counterparts such as T-tree [26].

Byte-addressable PM raises new challenges in using B+-tree because legacy B+-tree operations are implemented upon the assumption that block I/O is failure atomic. In modern computer architectures, the unit of atomicity guarantee and the unit of transfer do not coincide. The unit of atomicity in memory operations corresponds to a word, e.g. 64 bits whereas the unit of transfer between the CPU and memory corresponds to a cache line, e.g. 64 Bytes. This granularity mismatch is of no concern in current memory since it is volatile. When the memory is non-volatile, unexpected system failures may cause the result of incomplete cache flush operations to be externally visible after the system recovers. To make the granularity mismatch problem even worse, modern processors often make the most use of memory bandwidth by changing the order of memory operations. Besides, recently proposed memory persistency models such as *epoch persistency* [9] even allow cache lines to be written back out of order. To prevent the reordering of memory write operations and to ensure that the written data are flushed to PM without compromising the consistency of the tree structure, B+-trees for persistent memory use explicit memory fence operations and cache flush operations to control the order of memory writes [5, 42].

The recently proposed B+-tree variants such as NV-tree [49], FP-tree [34], and wB+-tree [5] have pointed out and addressed two problems of byte-addressable persistent B+-trees. The first problem is that a large number of fencing and cache flush operations are needed to maintain the sorted order of keys. And, the other problem is that the logging demanded by tree rebalancing operations

is expensive in the PM environment.

To resolve the first problem, previous studies have proposed a way to update tree nodes in an *append-only* manner and introduced additional metadata to the B+-tree structures. With these augmentations, the size of updated memory region in a B+-tree node remains minimal. However, the additional metadata for indirect access to the keys, and the unordered entries affect the cache locality and increase the number of accessed cache lines, which can degrade search performance.

To resolve the second problem of persistent B+-trees - logging demanded by tree rebalancing operations, NVTree [49] and FP-tree [34] employed selective persistence that keeps leaf nodes in the PM but internal nodes in volatile DRAM. Although the selective persistence makes logging unnecessary, it requires the reconstruction of tree structures on system failures and makes the instant recovery impossible.

In this study, we revisit B+-trees and propose a novel approach to tolerating *transient inconsistency*, i.e., partially updated inconsistent tree status. This approach guarantees the failure atomicity of B+-tree operations without significant fencing or cache flush overhead. If transactions are made to tolerate the transient inconsistency, we do not need to maintain a consistent backup copy and expensive logging can be avoided. The key contributions of this work are as follows.

- We develop *Failure Atomic Shift (FAST)* and *Failure-Atomic In-place Rebalance (FAIR)* algorithms that transform a B+-tree through *endurable transient inconsistent states*. The endurable transient inconsistent state is the state in which read transactions can detect incomplete previous transactions and ignore inconsistency without hurting the correctness of query results. Given that all pointers in B+-tree nodes are unique, the existence of duplicate pointers enables the system to identify the state of the transaction at the time of crash and to recover the B+-tree to the consistent state without logging.
- We make read transactions non-blocking. If every store instruction transforms a B+-tree index to another state that guarantees correct search results, read transactions do not have to wait until concurrent write transactions finish changing the B+-tree nodes.

In the sense that the proposed scheme warrants consistency via properly ordering the individual operations and the inconsistency is handled by read operations, they share much of the same idea as the soft-update technique [12, 31] and *NoFS* [7].

The rest of the paper is organized as follows: In Section 2, we present the challenges in designing a B+-tree index for PM. In Section 3 and Section 4, we propose the FAST and FAIR algorithms. In Section 5, we discuss how to enable non-blocking read transactions. Section 6 evaluates the performance of the proposed persistent B+-tree and Section 7 discusses other research efforts. In Section 8 we conclude this paper.

2 B+-tree for Persistent Memory

2.1 Challenge: clflush and mfence

A popular solution to guarantee transaction consistency and data integrity is the copy-on-write technique, which updates the block in an out-of-place manner. The copy-on-write update for block-based data structures can be overly expensive because it duplicates entire blocks including the unmodified portion of the block.

The main challenges in employing the in-place update scheme in B+-trees is that we store multiple key-pointer entries in one node and that the entries must be stored in a sorted order. Inserting a new key-pointer entry in the middle of an array will shift on average half the entries. In the recent literature [42, 5, 49], this shift operation has been pointed out as the main reason why B+-trees call many cache line flush and memory fence instructions. If we do not guard the memory write operations with memory fence operations, the memory write operations can be reordered in modern processors. And, if we do not flush the cache lines properly, B+-tree nodes can be updated partially in PM because some cache lines will stay in CPU caches. To avoid such a large number of cache line flushes and memory fence operations, append-only update strategy can be employed [49, 34, 5]. However, the append-only update strategy improves the write performance at the cost of a higher read overhead because it requires additional metadata or all unsorted keys in a tree node to be read [6].

2.2 Reordering Memory Accesses

The reordering of memory write operations helps better utilize the memory bandwidth [38]. In the last few decades, the performance of modern processors has improved at a much faster rate than that of memory [10]. In order to resolve the performance gap between CPU speed and memory access time, memory is divided into multiple cache banks so that the cache lines in the banks can be accessed in parallel. As a result, memory operations can be executed out of order.

To design a failure-atomic data structure for PM, we need to consider both volatile memory order and persist order. Let us examine volatile memory order first. Memory reordering behaviors vary across architectures and memory ordering models. Some architectures such as ARM allow store instructions to be reordered with each other[8], while other architectures such as x86 prevent stores-after-stores from being reordered [40, 18], that is, *total store ordering* is guaranteed. However, most archi-

tectures arbitrarily reorder stores-after-loads, loads-after-stores, and loads-after-loads unless dependencies exist in them. The Alpha is known to be the only processor that reorders dependent loads [30]. Given that the Alpha processor has deprecated since 2003, we consider that all modern processors do not reorder dependent loads.

Memory persistency [35] is a framework that provides an interface for enforcing persist ordering constraints on PM writes. Persist order in the *strict persistency* model matches the memory order specified in the memory consistency model. However, the memory persistency model may allow the persist order to deviate from the volatile order under the *relaxed persistency* model [9, 35]. To simplify our discussion, we first present algorithms assuming the strict persistency model. Later, in Section 7, we will discuss the relaxed persistency model.

3 Failure-Atomic ShifT (FAST)

3.1 Shift and Memory Ordering

In most architectures, the order of memory access operations is not changed arbitrarily if stores and loads have dependencies. Based on this observation, we propose the *Failure-Atomic ShifT* (FAST) scheme for B+-tree nodes. FAST frees the shift operation from explicitly calling a memory fence and a cache line flush instruction without compromising the ordering requirement. The idea of FAST is simple and straightforward. Let us first examine the case where total store ordering (TSO) is guaranteed.

The process of shifting array elements is a sequence of load and store instructions that are all dependent and must be called in a cascading manner. To insert an element in a sorted array, we visit the array elements in reverse order, that is, from the rightmost element to the left ones. Until we find an element $e[i]$ that is smaller than the element we insert, we shift the elements to the right: $e[j+1] \leftarrow e[j], j = N, \ldots, i+1$. The dependency between the consecutive move operations, $e[i+1] \leftarrow e[i]$ and $e[i] \leftarrow e[i-1]$, prohibits the CPU from performing Out-Of-Order writes[38] and guarantees that the records are moved while satisfying the *prefix constraint* [45], i.e., if $e[i] \leftarrow e[i-1]$ is successful, all preceding moves, $e[j] \leftarrow e[j-1], j = i+1, \ldots, N$ have been successful.

If the array size spans across multiple cache lines, it is uncertain which cache line will be flushed first if we do not explicitly call `clflush`. Therefore, our proposed FAST algorithm calls a cache line flush instruction when we shift an array element from one cache line to its adjacent cache line. Since FAST calls cache line flushes only when it crosses the boundary of cache lines, it calls cache line flush instructions only as many times as the number of dirty cache lines in a B+-tree node.

Even if we do not call a cache line flush instruction, a dirty cache line can be evicted from CPU caches and be flushed to PM. In FAST, such a premature cache line flush does not affect the consistency of a sorted array. The only condition that FAST requires is that the dirty cache lines must be flushed in order.

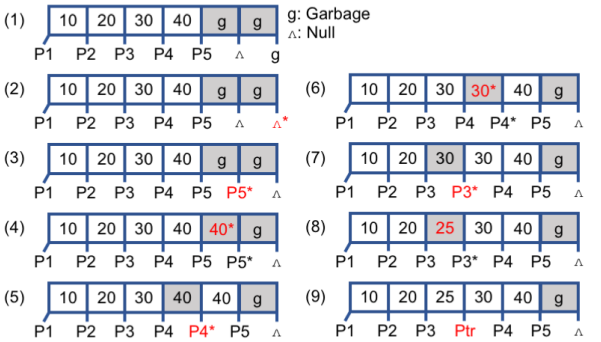


Figure 1: *FAST Inserting (25, Ptr) into B-tree node*

Suppose that an array element is shifted from one cache line to another. Since there exists a dependency between the load and the store instructions, they will not be reordered and we do not need to explicitly call a memory fence instruction. However, since cache line flush instructions can be reordered with store instructions, we call a memory fence instruction per cache line flush.

3.2 Endurable Inconsistency during Shift

The shift operation is not failure-atomic by itself because shifting an element can make it appear twice in a row. In case of a system failure, such *transient* duplicate elements can persist as shown in Figure 1, which seems at first glance unacceptably inconsistent. To resolve this issue, we use one property of B+-tree nodes; *B+-tree nodes do not allow duplicate pointers*. Therefore, a key in between duplicate pointers found during tree traversals can be ignored by transactions and considered as a tolerable *transient inconsistent* state. After finding a key of interest in a sorted array, we check if its left and right child pointers have the same addresses. If so, transactions ignore the key and continue to read the next key. This modification in traversal algorithm requires only one more compare instruction per each tree node visit, which incurs negligible overhead to transactions.

3.3 Insertion with FAST for TSO

In B+-tree nodes, the same number of keys and pointers (or keys and values) need to be shifted in tandem. We store keys and pointers as an array of structure in B+-tree nodes so that the corresponding keys and pointers are always flushed together.

The insertion algorithm is shown in Algorithm 1. Consider the example shown in Figure 1. We insert a key-value pair $(25, Ptr)$ into the B+-tree node shown in (1) of Figure 1. To make a space for 25, the two rightmost

Algorithm 1
FAST_insert(*node*, *key*, *ptr*)

```
 1: node.lock.acquire()
 2: if (sibling ← node.sibling_ptr) ≠ NULL then
 3:     if sibling.records[0].key < key then
 4:         – previous write thread has split this node
 5:         node.lock.release()
 6:         FAST_insert(sibling, key, ptr);
 7:         return
 8:     end if
 9: end if
10: if node.cnt < node_capacity then
11:     if node.search_dir_flag is odd then
12:         – if this node was updated by a delete thread, we
13:         – increase this flag to make it even so that
14:         – lock-free search scans from left to right
15:         node.search_dir_flag++;
16:     end if
17:     for i ← node.cnt − 1; i ≥ 0; i − − do
18:         if node.records[i].key > key then
19:             node.records[i+1].ptr ← node.records[i].ptr;
20:             mfence_IF_NOT_TSO();
21:             node.records[i+1].key ← node.records[i].key;
22:             mfence_IF_NOT_TSO();
23:             if &(node.records[i+1]) is at cacheline boundary
                then
24:                 clflush_with_mfence(&node.records[i+1]);
25:             end if
26:         else
27:             node.records[i+1].ptr ← node.records[i].ptr;
28:             mfence_IF_NOT_TSO();
29:             node.records[i+1].key ← key;
30:             mfence_IF_NOT_TSO();
31:             node.records[i+1].ptr ← ptr;
32:             clflush_with_mfence(&node.records[i+1]);
33:         end if
34:     end for
35:     node.lock.release()
36: else
37:     node.lock.release()
38:     FAIRsplit(node, key, ptr);
39: end if
```

keys, 30 and 40, their pointers $P4$ and $P5$, and the sentinel pointer *Null* must be shifted to the right.

First, we shift the sentinel pointer *Null* to the right as shown in (2). Next, we shift the right child pointer $P5$ of the last key 40 to the right, and then we shift key 40 to the right, as shown in (3) and (4). In step (3) and (4), we have a garbage key and a duplicate key 40 as the last key respectively. Such inconsistency can be tolerated by making other transactions *ignore the key between the same pointers* ($P5$ and $P5*$ in the example).

In step (5), we shift $P4$ by overwriting $P5$. This atomic write operation invalidates the old key 40 ($[P4, 40, P4*]$) and validates the shifted key 40 ($[P4*, 40, P5*]$). Even if a system crashes at this point, the key 40 between the redundant pointers ($[P4, 40, P4*]$) will be ignored. Next, the key 30 can be shifted to the right by overwriting the key 40 in step (6). Next, in step (7), we shift $P3$ to the right to invalidate the old key 30 ($[P3, 30, P3*]$) and make space for the key 25, that we insert. In step (8), we store 25 in the third slot. However, the key 25 is not valid because it is in between the same pointers ($[P3, 25, P3*]$). Finally, in step (9), we overwrite $P3*$ with the new pointer Ptr, which will validate the new key 25. In FAST, writing a new pointer behaves as a commit mark of an insertion.

Unlike pointers, the size of keys can vary. If a key size is greater than 8 bytes, it cannot be written atomically. However, our proposed FAST insertion makes changes to a key only if it is located between the same pointers. Therefore, even if a key size is larger than 8 bytes and even if it cannot be updated atomically, read operations ignore such partially written keys and guarantee correct search results.

In this sense, every single 8-byte write operation in the FAST insertion algorithm is failure-atomic and crash consistent because partially written tree nodes are always usable. Hence, even if a system crashes during the update, FAST guarantees recoverability as long as we flush dirty cache lines in an order.

3.4 FAST for Non-TSO Architectures

ARM processors can reorder store instructions if they do not have a dependency. I.e., if a store instruction updates a key and another store instruction updates a pointer, they can be reordered.

First, consider the case where keys are no larger than 8 bytes. Then, keys and pointers can be shifted independently because no key or pointer in either array can be in a partially updated status. Suppose that the ith and $i+1$th keys are the same and the jth and $j+1$th pointers are the same ($i \neq j$). We can simply ignore one of the redundant elements in both arrays, easily sort out which pointer is for which key's child node, and reconstruct the correct logical view of the B+-tree node.

Second, consider the case where the keys are larger than 8 bytes. One easy option is to call a memory fence instruction for each shift operation. Although this option calls a large number of memory fence instructions, it calls cache line flush instructions only as many times as the number of dirty cache lines as in TSO architectures. Although memory fence overhead is not negligible, it is much smaller than cache line flush overhead. Alternatively, we can make B+-tree store the large keys in a separate memory heap and store pointers to the keys in B+-tree nodes. This option allows us to avoid a large number of memory fence instructions. However, through experiments, we found the indirect access and the poor cache

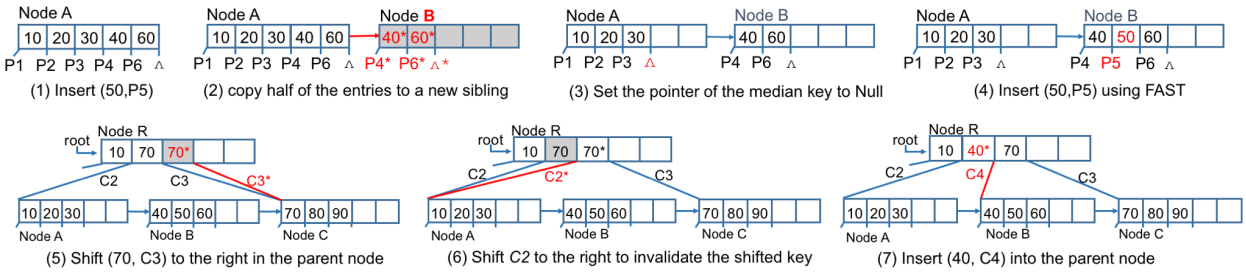


Figure 2: *FAIR Node Split*

locality significantly degrade the search performance.

3.5 Deletion with FAST

Deletion can be performed in a similar manner but in reverse order of insertion. Consider the deletion of $(25, Ptr)$ from the tree node illustrated in (9) of Figure 1. First, we overwrite Ptr with $P3$ to invalidate the key 25. Note that this atomic write operation behaves as the commit mark of the deletion. If a system crashes and a subsequent query finds the B+-tree node as shown in (8), the key 25 ($[P3, 25, P3*]$) will be considered as an invalid key because it is in between duplicate pointers.

In the subsequent steps shown in (7)∼(3), the pointer array will have adjacent duplicate pointers that invalidate one of the keys while we shift them to the left. Finally, in step (2), we shift the sentinel pointer *Null* to the left, which completes the deletion process. Similar to the insertion, we flush dirty cache lines only when we are about to make the next cache line dirty.

4 Failure-Atomic In-place Rebalancing

4.1 FAIR: Node Split

In both legacy disk-based B-trees and recently proposed B+-tree variants for PM [49, 34, 5], logging or journaling has been used when splitting or merging multiple tree nodes. If a tree node splits, (1) we create a sibling node, (2) move half of the entries from the overfull node to the new sibling node, and (3) insert a pointer to the new sibling node and its smallest key to the parent node. Since these three steps must be performed atomically, logging can be used. However, logging duplicates dirty pages. It not only increases the write traffic but also blocks the concurrent access to tree nodes.

We avoid the expensive logging by leveraging the FAST in-place update scheme and the *Failure-Atomic In-place Rebalance* (FAIR) node split algorithm described in Algorithm 2. In our B+-tree design, we store a sibling pointer not just for leaf nodes but also for internal nodes as in the B-link tree [25].

Figure 2 illustrates each step of the FAIR node split algorithm with an example. We examine every individual store operations in splitting a tree node and carefully lay them out so that we do not have to maintain an additional copy of data for crash recovery and consistency is not compromised in the event of system failure.

First, suppose there is only one node in the tree as shown in (1). If we insert a new key 50, the leaf node A will split. In (2), we create a sibling node B using a PM heap manager [33], copy half of the entries, call cache line flushes for the new node, and make the sibling pointer of node A point to the sibling node B. The sibling pointer must be written before we delete the migrated en-

Algorithm 2
FAIR_split$(node, key, ptr)$

```
1: node.lock.acquire()
2: if (sibling ← node.sibling_ptr) ≠ NULL then
3:     if sibling.records[0].key < key then
4:         node.lock.release()
5:         FAST_insert(sibling, key, ptr);
6:         return
7:     end if
8: end if
9: sibling ← nv_malloc(sizeof(node));
10: median ← node_capacity/2
11: for i ← median; i < node_capacity; i++ do
12:     FAST_insert_without_lock(sibling,
        node.records[i].key, node.records[i].ptr);
13: end for
14: sibling.sibling_ptr ← node.sibling_ptr;
15: clflush_with_mfence(sibling);
16: node.sibling_ptr ← sibling;
17: clflush_with_mfence(&node.sibling_ptr);
18: node.records[median].ptr ← NULL;
19: clflush_with_mfence(&node.records[median]);
20: – split is done. now insert (key,ptr)
21: if key < node.records[median].key then
22:     FAST_insert_without_lock(node, key, ptr)
23: else
24:     FAST_insert_without_lock(sibling, key, ptr)
25: end if
26: node.lock.release()
27: – update the parent node by traversing from the root.
28: FAST_internal_node_insert(root, node.level+
    1, sibling.records[0].key, node.sibling_ptr);
```

tries in the overfull node A. It looks as if the consistency of the tree nodes is violated because we have duplicate entries in nodes A and B; however, the right sibling node B will not be used until we delete duplicate entries from A because the smallest key in the right sibling node (40) is smaller than the largest key in the overfull node (60).

Suppose a system crashes in state (2). If a query that searches for key 35 is submitted, it will access node A following the search path from the root node because the sibling node B has not been added to the parent node yet. Since 35 is smaller than 40 (the one in node A), the search will stop without accessing node B. If the search key is 65, the query will follow the sibling pointer, access the sibling node B, and compare to see if the first key of the sibling node B is greater than the largest key in the node A. If it is, the query will keep checking the next key until it finds a matching key or a larger key than the search key. However, in this example, the search will stop after it finds that the first key 40 in the sibling node B is smaller than the largest key 60 in the node A. The basic assumption that makes this redundancy and inconsistency tolerable is that the right sibling nodes always have larger keys than the left sibling nodes. Therefore, the keys and pointers in the new right sibling node ($[P4*, 40*, P6*, 60*, Null]$) will not be accessed until we delete them in the overfull node A. We delete migrated keys and pointers from the overfull node by atomically setting the pointer of the median to NULL. This will change the tree status to (3). If we search the key 40 after deleting the migrated entries, the query will find that the largest key in node A is smaller than the search key, and follow the sibling pointer to find the key 40 in node B.

Figure 2 (5)∼(7) illustrate how we can endure transient inconsistency when we update the parent node. In step (5), node A and its sibling node B can be considered as a single virtual node. In the parent node R, we add a key-pointer pair for node B, i.e., $(40, C4)$ in the example. Since the parent node has keys greater than the key 40 that we add, we shift them using FAST as shown in (5) and (6). After we create a space in the parent node by duplicating the pointer $C2$, we finally store $(40, C4)$ as shown in (7).

4.2 FAIR: Node Merge

In B+-tree, underutilized nodes are merged with sibling nodes. If a deletion causes a node to be underutilized, we delete the underutilized node from its parent node so that the underutilized node and its left sibling node become virtually a single node. Once we detach the underutilized node from the parent node, we check whether the underutilized node and its left sibling node can be merged. If the left sibling node does not have enough entries, we merge them using FAST. If the left sibling node has enough entries, we shift some entries from the left sibling node to the underutilized node using FAST, and insert the new smallest key of the right sibling node to its parent node. This node merge algorithm is similar to the split algorithm, but it is performed in the reverse order of the split algorithm. Thus, a working example of the node merge algorithm can be illustrated by reversing the order of the steps shown in Figure 2.

5 Lock-Free Search

With the growing prevalence of many-core systems, concurrent data structures are becoming increasingly important. In the recently proposed *memory driven computing* environment that consists of a large number of processors accessing a persistent memory pool [1], concurrent accesses to shared data structures including B+-tree indexes must be managed efficiently in a thread-safe way.

As described earlier, the sequences of 8 byte store operations in FAST and FAIR algorithms guarantee that no read thread will ever access inconsistent tree nodes even if a write thread fails while making changes to the B+-tree nodes. In other words, even if a read transaction accesses a B+-tree partially updated by a suspended write thread, it is guaranteed that the read transaction will return the correct results. On the contrary, read transactions can be suspended and a write transaction can make changes to the B+-tree that the read transactions were accessing. When the read transactions wake up, they need to detect and tolerate the updates made by the write transaction. In our implementation, tree structure modifications, such as page splits or merges, can be handled by the concurrency protocol of the B-link tree [25]. However, the B-link tree has to acquire an exclusive lock to update a single tree node atomically. However, leveraging the FAST algorithm, we can design a non-blocking lock-free search algorithm to allow concurrent accesses to the same tree node as described below.

To enable a lock-free search, all queries must access a tree node in the same direction. That is, while a write thread is shifting keys and pointers to the right, read threads must access the node from left to right. If a write thread is deleting an entry using left shifts, read threads must access the node from right to left. Suppose the following example. A query accesses the tree node shown in Figure 1(1) to search for key 22. After the query reads the first two keys - 10 and 20, it is then suspended before accessing the next key 30. While the query thread is suspended, a write thread inserts 25 and shifts keys and pointers to the right. When the suspended query thread wakes up later, the tree node may be different from what it was before. If the tree node is in one of the states (1)∼(6), or (9), the read thread will follow the child pointer $P3$ without a problem. If the tree node is in state (7) or (8), the read thread will find the left and right child pointers are the same. Then, it will ignore the current key

Algorithm 3
LockFreeSearch$(node, key)$

```
1: repeat
2:   ret ← NULL;
3:   prev_switch ← node.switch;
4:   if prev_switch is even then
5:     – we scan this node from left to right
6:     for i ← 0; records[i].ptr ≠ NULL; i++ do
7:       if (k ← records[i].key) = key&&records[i].ptr ≠
         (t ← records[i+1].ptr) then
8:         if (k = records[i].key) then
9:           ret ← t;
10:          break;
11:        end if
12:      end if
13:    end for
14:  else
15:    – this node was accessed by a delete query
16:    – we have to scan this node from right to left
17:    for i ← node.cnt − 1; i >= 0; i−− do
18:      – omitted due to symmetry and lack of space
19:    end for
20:  end if
21: until prev_switch = node.switch
22: if ret = NULL && (t ← node.sibling_ptr) then
23:   if t.records[0].key ≤ key then
24:     return t;
25:   end if
26: end if
27: return ret;
```

and move on to the next key so that it follows the correct child pointer. From this example, we can see that shifting keys and pointers in the same direction does not hurt the correctness of concurrent non-blocking read queries.

If a read thread scans an array from left to right while a write thread is deleting its element by shifting to the left, there is a possibility that the read thread will miss the shifted entries. But if we shift keys and pointers in the same direction, the suspended read thread cannot miss any array element even if it may encounter the same entry multiple times. However, this does not hurt the correctness of the search results.

To guide which direction read threads scan a tree node, we use a counter flag which increases when insertions and deletions take turn. That is, the flag is an even number if a tree node has been updated by insertions and an odd number if the node has been updated by deletions. Search queries determine in which direction it scans the node according to the flag, and it double checks whether the counter flag remains unchanged after the scan. If the flag has changed, the search query must scan the node once again. A pseudo code of this lock-free search algorithm is shown in Algorithm 3.

Because of the restriction on the search direction, our lock-free search algorithm cannot employ a binary search. Although the binary search is known to perform faster than the linear search, its performance can be lower than that of the linear search when the array size is small, as we will show in Section 6.2.

5.1 Lock-Free Search Consistency Model

A drawback of a lock-free search algorithm is that a deterministic ordering of transactions cannot be enforced. Suppose a write transaction inserts two keys - 10 and 20 into a tree node and another transaction performs a range query and accesses the tree node concurrently. The range query may find 10 in the tree node but may not find 20 if it has not been stored yet. This problem can occur because the lock-free search does not serialize the two transactions. Although the lock-free search algorithm helps improve the concurrency level, it is vulnerable to the well-known *phantom reads* and *dirty reads* problems [41].

In the database community, various levels of isolation such as *serializable mode*, *non-repeatable read mode*, *read committed mode*, and *read uncommitted mode* [41] have been studied and used. Considering our lock-free search algorithm is vulnerable to dirty reads, it operates in read uncommitted mode. Although read uncommitted mode can be used for certain types of queries in OLAP systems, the scope of its usability is very limited.

To support higher isolation levels, we must resort to other concurrency control methods such as key range locks, snapshot isolation, or multi-version schemes [41]. However, these concurrency control methods may impose a significant overhead. To achieve both a high concurrency level and a high isolation level, we designed an alternative method to compromise. That is, we use read locks only for leaf nodes considering the commit operations only occur in leaf nodes. For internal tree nodes, read transactions do not use locks since they are irrelevant to phantom reads and dirty reads. Since transactions are likely to conflict more in internal tree nodes rather than in leaf nodes, read locks in leaf nodes barely affect the concurrency level as we will show in Section 6.7.

5.2 Lazy Recovery for Lock-Free Search

Since the reconstruction of a consistent logical view of a B+-tree is always possible with an inconsistent but correctable B+-tree, we perform a recovery in a lazy manner. We do not let read transactions fix tolerable inconsistency because read transactions are more latency sensitive than write transactions. In our design, instead, we make only write threads fix tolerable inconsistencies. Such a lazy recovery approach is acceptable because FAST allows at most one pair of duplicate pointers in each node. Thus, it does not waste a significant amount of memory space, and its performance impact on search

is negligible. Besides, the lazy recovery is necessary for a lock-free search. In lock-free searches, read threads and a write thread can access the same node. If read threads must fix the duplicate entries that a write thread caused, read threads will compete for an exclusive write lock. Otherwise, read threads have to check whether the node is inconsistent due to a crash or due to an inflight insert or delete, which will introduce significant complexities and latency to reads.

To fix inconsistent tree nodes, we delete the garbage key in between duplicate pointers by shifting the array to the left. For a dangling sibling node, we check if the sibling node can be merged with its left node. If not, we insert the pointer to the sibling node into the parent node.

In the FAIR scheme, the role of adding a sibling node to the parent node is not confined to the query that created the sibling node. Even if a process that split a node crashes for some reason, a subsequent process that accesses the sibling node via the sibling pointer triggers a parent node update. If multiple write queries visit a tree node via the sibling pointer, only one of them will succeed in updating the parent node and the rest of the queries will find that the parent has already been updated.

6 Experiments

We evaluate the performance of FAST and FAIR[1] against the state-of-the-art indexing structures for PM - wB+-tree with *slot+bitmap nodes* [5], FP-tree [34], WORT [23], and Skiplist [16].

wB+-tree [5] is a B+-tree variant that stores all tree nodes in PM. wB+-tree inserts data in an append-only manner. To keep the ordering of appended keys, wB+-tree employs a small metadata, called *slot-array*, which manages the index of unsorted keys in a sorted order. In order to atomically update the slot-array, wB+-tree uses a separate bitmap to mark the validness of array elements and slot-array. Because of these metadata, wB+-tree calls at least four cache line flushes when we insert data into a tree node. Besides this, wB+-tree also requires expensive logging and a large number of cache line flushes when nodes split or merge.

FP-Tree [34] is a variant of a B+-tree that stores leaf nodes in PM but internal nodes in DRAM. It proposes to reduce the CPU cache miss ratio via *finger printing* and employs hardware transactional memory to control concurrent access to internal tree nodes in DRAM. FP-Tree claims that internal nodes can be reconstructed without significant overhead. However, the reconstruction of internal nodes is not very different from the reconstruction of the whole index. We believe one of the most important benefits of using PM is the instantaneous recoverability. With FP-Tree, such an instantaneous recovery is impossible. Thus, strictly speaking, FP-Tree is not a persistent index.

WORT [23] is an alternative index for PM based upon a radix tree. Unlike B+-tree variants, the radix tree does not require key sorting nor rebalancing tree structures. Since the radix tree structure is deterministic, insertion or deletion of a key requires only a few 8 byte write operations. However, radix trees are sensitive to the key distribution and often suffers from poor cache utilization due to their deterministic tree structures. Also, radix trees are not as versatile as B+-trees as their range query performance is very poor.

SkipList [16] is a probabilistic indexing structure that avoids expensive rebalancing operations. An update of the skip list needs pointer updates in multi-level linked lists. But only the lowest level-linked list needs to be updated in a failure-atomic way. In a Log-Structured NVMM System [16], SkipList was used as a volatile address mapping tree, but SkipList shares the same goal with our B+-tree. That is, both indexing structures do not need logging, enable lock-free search, and benefit from a high degree of parallelism.

6.1 Experimental Environment

We run experiments on a workstation that has two Intel Xeon Haswell-Ex E7-4809 v3 processors (2.0 GHz, 16 vCPUs with hyper-threading enabled, and 20 MB L3 cache) that guarantee total store ordering and 64 GB of DDR3 DRAM. We compiled all implementations using g++ 4.8.2 with -O3 option.

We use a DRAM-based PM latency emulator - *Quartz*[43] as was done in [44, 3, 20]. Quartz models application-perceived PM latency by inserting stall cycles in each predefined time interval called *epoch*. In our experiments, the minimum and maximum epochs are set to 5 nsec and 10 nsec respectively. We assume that PM bandwidth is the same as that of DRAM since Quartz does not allow us to emulate both latency and bandwidth at the same time.

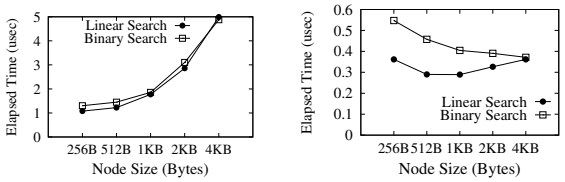
6.2 Linear Search vs. Binary Search

In the first set of experiments shown in Figure 3, we index 1 million random key-value pairs of 8 bytes each and evaluate the performance of the linear search and the binary search with varying size of B+-tree nodes. We assume the latency of PM is the same as that of DRAM.

As we increase the size of the tree nodes, the tree height decreases in log scale but the number of data to be shifted by the FAST algorithm in each node linearly increases. As a result, Figure 3(a) shows the insertion performance degrades with larger tree node sizes.

Regarding search performance, Figure 3(b) shows the binary search performance improves as we increase the tree node size because of the smaller tree height and the

[1] Codes are available at https://github.com/DICL/FAST_FAIR



(a) Insertion Time (b) Search Time

Figure 3: *Linear vs. Binary Search*

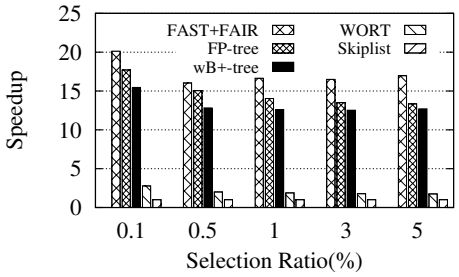


Figure 4: *Range Query Speed-Up from using SkipList with Varying Selection Ratio (AVG. of 5 Runs)*

fewer number of key comparisons. However, the binary search often stalls due to poor cache locality and the failed branch prediction. As a result, it performs slower than the linear search when the tree node size is smaller than 4 KB. Although the linear search accesses more cache lines and incurs more LLC cache misses, memory-level parallelism (MLP) helps read multiple cache lines at the same time. As a result, the number of effective LLC cache misses of the linear search is smaller than that of the binary search.

Overall, our B+-tree implementation shows good insertion and search performance when the tree node size is 512 bytes and 1 KB. Hence, we set the B+-tree node size to 512 bytes for the rest of the experiments unless stated otherwise. Note that the 512-byte tree node size occupies only eight cache lines, thus FAST requires eight cache line flushes in the worst case and four cache line flushes on average. Since the binary search makes lock-free search impossible and the linear search performs faster than binary search when the node size is small, we use the linear search for the rest of the experiments.

6.3 Range Query

A major reason to use B+trees instead of hash tables is to allow range queries. Hash indexes are known to perform well for exact match queries, but they cannot be used if a query involves the ordering of data such as *ORDER BY* sorting and *MIN/MAX* aggregation. Thus many commercial database systems use a hash index as a supplementary index of the B+-tree index.

In the experiments shown in Figure 4, we show the relative performance improvement over SkipList for range query performance with various selection ratios. The selection ratio is the proportion of selected data to the number of data in an index. We set the read latency of PM to 300 nsec and inserted 10 million 8-byte random integer keys into 1 KB tree nodes. A B+-tree with FAST and FAIR processes range queries up to 20 times faster than SkipList and consistently outperforms other persistent indexing structures (6∼27% faster than FP-tree and 25∼33% faster than wB+-tree) due to its simple structure and sorted keys.

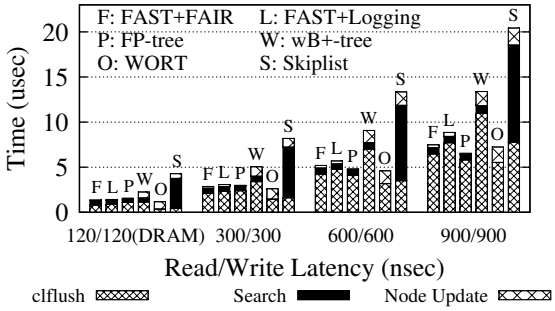


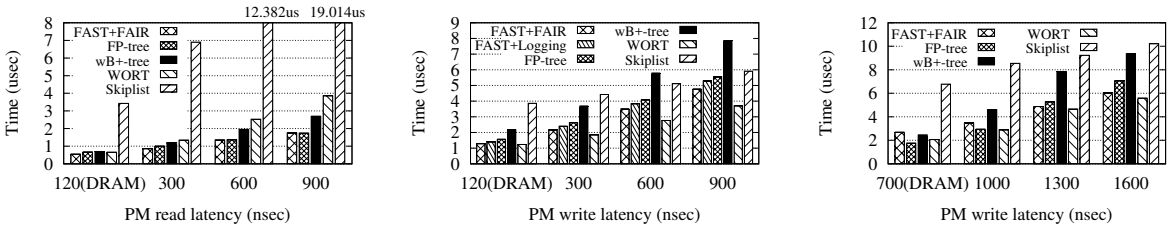
Figure 5: *Breakdown of Time Spent for B-tree Insertion (AVG. of 5 Runs)*

6.4 PM Latency Effect

In the experiment shown in Figure 5 and 6, we index 10 million 8 byte key-value pairs in uniform distribution and measure the average time spent per query while increasing the read and write latency of PM from 300 nsec. For FP-tree, we set the size of the leaf nodes and internal nodes to 1 KB and 4 KB respectively as was done in [34]. The node size of wB+-tree is fixed at 1 KB because each node can hold no more than 64 entries. Both configurations are the fastest performance settings for FP-tree and wB+-tree [34].

Figure 5 shows that FAST+FAIR, FP-tree, and WORT show comparable insertion performances and they outperform wB+-tree and SkipList by a large margin. In detail, the insertion time is composed of three parts, *Cache line flush*, *Search*, and *Node Update* times. wB+-tree calls a 1.7 times larger number of cache line flushes than FAST+FAIR. We also measured the performance of a FAST-only B+-tree with legacy tree rebalancing operations that have a logging overhead. Because of the logging overhead, FAST+Logging performs 7∼18% slower than FAST+FAIR.

The poor performance of SkipList is because of its poor cache locality. Without clustering similar keys in contiguous memory space and exploiting the cache locality, byte-addressable in-memory data structures such as SkipList, radix trees, and binary trees fail to lever-



(a) Search : Varying Read Latency

(b) Insert : Varying Write Latency on TSO Architecture

(c) Insert : Varying Write Latency on Non-TSO Architecture

Figure 6: *Performance Comparison of Single-Threaded Index (AVG. of 5 Runs)*

age memory level parallelism. Hence, block-based data structures such as B+-trees that benefit from clustered keys need to be considered not only for block device storage systems but also for byte-addressable PM. FP-tree benefits from faster access to internal nodes than FAST+FAIR and WORT, but it calls a slightly larger number of cache line flushes than FAST+FAIR (4.8 v.s 4.2) because of fingerprints and leaf-level logging.

Figure 6(a) shows how the read latency of PM affects the *exact match query* performance. FP-tree shows a slightly faster search performance than FAST+FAIR when the read latency is higher than 600 nsec because it has faster access to volatile internal nodes. When the read latency is 900 nsec, WORT spends twice as much time as FAST+FIAR to search the index. Interestingly, the average number of LLC cache misses of WORT that we measured with *perf* is 4.9 while that of FAST+FAIR is 8.3 in the experiments. Although B+-tree variants have a larger number of LLC cache misses than WORT, mostly they access adjacent cache lines and benefit from serial memory accesses, i.e., hardware prefetcher and memory level parallelism. To reflect the characteristics of modern hardware prefetcher and memory-level parallelism and to avoid overestimation of the overhead of serial memory accesses, Quartz counts the number of memory stall cycles for each LOAD request and divides it by the memory latency to count the number of serial memory accesses and estimate the appropriate read latency for them [43]. Therefore, the search performances of B+-tree variants are less affected by the increased read latency of PM compared to WORT and SkipList.

In the experiments shown in Figure 6(b), we measure the average insertion time for the same batch insertion workload while increasing only the write latency of PM. As we increase the write latency, WORT, which calls fewer cache line flushes than FAST+FAIR, outperforms all other indexes because the number of cache line flushes becomes a dominant performance factor and the poor cache locality of WORT gives less impact on the performance. FAST+FAIR consistently outperforms FP-tree, wB+-tree, and Skip List as it calls a lower number of `clflush` instructions.

6.5 Performance on Non-TSO

Although stores-after-stores are not reordered in X86 architectures, ARM processors do not preserve total store ordering. To evaluate that the performance of FAST on non-TSO architectures, we add `dmb` instruction as a memory barrier to enforce the order of store instructions and measure the insertion performance of FAST and FAIR on a smartphone, *Nexus 5*, which has a 2.26 GHz Snapdragon 800 processor and 2 GB DDR memory. To emulate PM, we assume that a particular address range of DRAM is PM and the read latency of PM is no different from that of DRAM. We emulate the write latency of PM by injecting `nop` instructions. Since the word size of Snapdragon 800 processor is 4 bytes, the granularity of failure-atomic writes is 4 bytes, and the node size of wB+-tree and FP-tree is limited to 256 bytes accordingly. Figure 6(c) shows that, when the PM latency is the same as that of DRAM, our proposed FAST+FAIR shows worse performance than FP-tree although FP-tree calls more cache line flush instructions. This is because FAST+FAIR calls more memory barrier instructions than FP-tree on ARM processors (16.2 vs. 6.6). However, as the write latency of PM increases, FAST+FAIR outperforms other indexes because the relative overhead of `dmb` becomes less significant compared to that of the cache line flushes (`dccmvac`). In our experiments, the performance of FAST+FAIR is up to 1.61 times faster than wB+-tree.

6.6 TPC-C Benchmark

In real-world applications, workloads are often mixed with writes and reads. In the experiments shown in Figure 7, we evaluate the performance using TPC-C benchmark. TPC-C benchmark is an OLTP benchmark that consists of 5 different types of queries (New-Order, Payment, Order-Status, Delivery, and Stock-Level). We varied the percentage of these queries to generate four different workloads so that the proportion of search queries increases from W1 to W4. The read and write latency of PM are both set to 300 nsec. FAST+FAIR consistently outperforms other indexes because of its good insertion performance and superior range query performance due

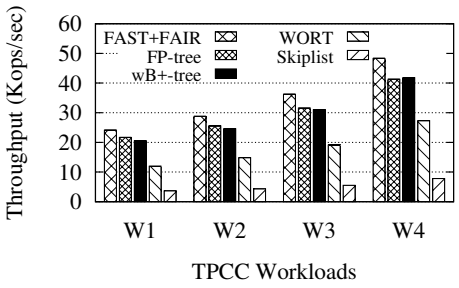


Figure 7: *TPC-C benchmark (AVG. of 5 Runs) : W1 [NewOrder 34%, Payment 43%, Status 5%, Delivery 4%, StockLevel 14%], W2 [27%, 43%, 15%, 4%, 11%], W3 [20%, 43%, 25%, 4%, 8%], W4 [13%, 43%, 35%, 4%, 5%]*

to the sorted data in its leaf nodes. While WORT shows the fastest insertion performance, it suffers from a poor range query performance in this benchmark.

6.7 Concurrency and Recoverability

In the experiments shown in Figure 8, we evaluate the performance of multi-threaded versions of FAST+FAIR, FP-tree, SkipList, and B-link tree. Although wB+-tree and WORT are not designed to handle concurrent queries, they can employ well-known concurrency control protocols such as the *crabbing protocol* [41]. However, we do not implement and evaluate multi-threaded versions of wB+-tree and WORT. Instead, we present the performance of *B-link tree* for reference because it is known that *B-link tree* outperforms the crabbing protocol [41]. Note that B-link tree is not designed to provide the failure-atomicity for byte-granularity writes in PM and B-link tree does not allow the lock-free search. We implemented the concurrent version of FP-tree using Intel's Transactional Synchronization Extension (TSX) as was done in [34]. For this experiments, the write latency of PM is set to 300 nsec and the read latency of PM is set to be equal to that of DRAM. FAST+FAIR and SkipList eliminate the necessity of read locks but they require write locks to serialize write operations on tree nodes. Our implementations of FAST+FAIR and B-link tree use `std::mutex` class in C++11 and Skiplist uses a spin lock with gcc built-in CAS function, `__sync_bool_compare_and_swap`. But they can also employ the hardware transactional memory for higher concurrency. We compiled these implementations without the -O3 optimization option because the compiler optimization can reorder instructions and it affects the correctness of lock-free algorithm.

Although these experiments are intended to evaluate the concurrency, they also show the instant recoverability of FAST and FAIR algorithms. In a physical power-off test, we need to generate a large number of partially updated transient inconsistent tree nodes and see whether read threads can tolerate such inconsistent tree nodes. In the lock-free concurrency experiments, a large number of read transactions access various partially updated tree nodes. If the read transactions can ignore such transient inconsistent tree nodes, instant recovery is possible.

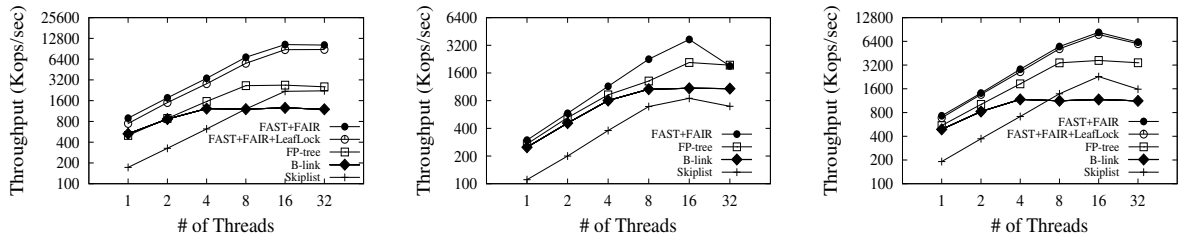
In the experiments shown in Figure 8, we run three workloads - *50M Search*, *50M Insertion*, and *Mixed*. We insert 50 million 8 byte random keys into the index and run each workload: For *50M Insertion* workload, we insert additional 50 million keys into the index. For *50M Search* we search 50 million keys. And for *Mixed* workload, each thread alternates between four insert queries, sixteen search queries, and one delete query. We use *numactl* to bind threads explicitly to a single socket to minimize the socket communication overhead and we distribute the workload across a number of threads.

Figure 8(a) shows that FAST+FAIR gains about a 11.7x faster speedup when the number of threads increases from one to sixteen. However, the speed-up saturates over 16 threads because our testbed machine has 16 vCPUs in a single socket. For FP-tree and B-link tree, the search speed-up becomes saturated when we use 8 and 4 threads respectively. When we run 8 threads, FP-tree takes advantage of the TSX and shows a throughput about 2.2x higher than B-link tree. Since SkipList also benefits from lock-free search, it scales to 16 threads but from a much lower throughput. Although FAST+FAIR+LeafLocks requires read threads to acquire read locks in leaf nodes, FAST+FAIR+LeafLocks shows a comparable concurrency level with FAST+FAIR. Note that the lock-free FAST+FAIR operates at read uncommitted mode while FAST+FAIR+LeafLocks operates at serializable mode.

In terms of write performance, FP-tree does not benefit much from the TSX as it shows a similar performance to B-link tree. It is because FP-tree performs expensive logging when a leaf node splits although it benefits from the faster TSX-enabled lock. In Figure 8(b), FAST+FAIR achieves about 12.5x higher insertion throughput when 16 threads run concurrently. In contrast, FP-tree and B-link tree achieve only a 7.7x and 4.4x higher throughput. Figure 8(c) shows that the scalability of FP-tree and B-link tree is limited because of read locks while FAST+FAIR takes advantage of the lock-free search. For the mixed workload, FAST+FAIR achieves up to a 11.44x higher throughput than a single thread.

7 Related Work

Lock-free index: In parallel computing community, various non-blocking implementations of popular data structures such as queues, lists, and search trees have been studied [4, 14, 15, 32]. Among various lock-free data structures, Braginsky et al.'s lock-free dynamic B+-tree [4] and Levandoski's Bw-tree [28] are the most rel-



(a) 50M Search (b) 50M Insert (c) 200M Search/50M Insert/12.5M Delete

Figure 8: *Performance with Varying Number of Threads (AVG. of 5 Runs)*

evant to our work. Unlike their lock-free B+-tree implementations, our current design still requires write latches for write threads because persistent B+-trees need to address durability as an additional challenge. We leave lock-free writes for persistent index as a future work.

Memory persistency: In order to resolve the ordering issues of memory writes in PM-based systems, numerous works [9, 35, 29] have lately proposed novel memory persistency models, such as *strict persistency* [35] and *epoch persistency* [9]. Strict persistency does not distinguish memory consistency from memory persistency, but *epoch persistency* [9] requires persist barriers so that persist order may deviate from the volatile memory order. These persistency models complement our work. FAST and FAIR guarantee the consistency of B+-tree under strict persistency model. If memory consistency is decoupled from memory persistency and the persist order deviates from the volatile memory order as in *relaxed persistency*, FAST and FAIR must call a persist barrier for every cache line flush instruction because they must enforce the order of cache line flushes to PM. However, within the same cache line, we do not need to call a persist barrier for each shift operation because array elements in the same cache line are guaranteed to be flushed to PM even under the relaxed persistency model. The only condition that FAST and FAIR require is that the dirty cache lines must be flushed in order. Therefore, FAST and FAIR place minimal persistence overhead under both strict and relaxed persistency models. That is, FAST calls persist barriers only as many times as the number of dirty cache lines in a B+-tree node under the relaxed persistency. On the other hand, other persistent indexes such as wB+-tree and FP-tree that employ append-only strategy need to call a persist barrier for each store instruction since their store instructions are not dependent. Due to the unavailability of PM that implements various persistency models, we leave a performance evaluation of our FAST and FAIR schemes under relaxed persistency model to our future work

Hardware transactional memory: The advent of commercially available hardware transactional memory such as the Intel's Restricted Transactional Memory (RTM) and Hardware Lock Elision (HLE) can be used to support coarse-grained atomic cache line writes [13, 27, 39, 46, 47]. Hardware transactional memories guarantee a dirty cache line remains in the write combining store buffer so that isolation can be preserved. However, memory persistency models including even strict persistency do not guarantee multiple cache lines will be flushed atomically even with the help of hardware transactional memory [39]. Hence, if a system crashes while flushing multiple cache lines, its consistency can not be guaranteed. Hence, hardware transactional memory in its current form cannot replace our FAST and FAIR algorithms as long as the tree node size is larger than a single cache line [22] and a tree needs rebalancing operations.

8 Conclusion

In this work, we have designed, implemented, and evaluated Failure-Atomic ShifT (FAST) and Failure-Atomic In-place Rebalance (FAIR) algorithms for legacy B+-trees to get the most benefit out of byte-addressable persistent memory. FAST and FAIR solves the granularity mismatch problem of PM without using logging and without modifying the data structure of B+-trees.

FAST and FAIR algorithms transform a consistent B+-tree into another consistent state or a *transient inconsistent* state that read operations can endure. By making read operations tolerate transient inconsistency, we can avoid expensive copy-on-write and logging. Besides, we can isolate read transactions, which enables non-blocking lock-free search.

Acknowledgement

We would like to give our special thanks to our shepherd Dr. Jorge Guerra and the anonymous reviewers. Their comments and suggestions helped us correct and improve this work. This research was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFCSRFC-IT1501-04. The first and the second authors contributed equally and the corresponding author is Beomseok Nam.

References

[1] HP Enterprise Lab, Memory Driven Computing. *https://www.labs.hpe.com/next-next/mdc*.

[2] Intel and Micron produce breakthrough memory technology. *https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology*.

[3] ARULRAJ, J., PAVLO, A., AND DULLOOR, S. R. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), ACM, pp. 707–722.

[4] BRAGINSKY, A., AND PETRANK, E. A lock-free B+tree. In *24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2012).

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

[6] CHI, P., LEE, W.-C., AND XIE, Y. Making B+-tree efficient in PCM-based main memory. In *Proceedings of the 2014 international symposium on Low power electronics and design* (2014), ACM, pp. 69–74.

[7] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)* (2012).

[8] CHONG, N., AND ISHTIAQ, S. Reasoning about the ARM weakly consistent memory model. In *Proceedings of the 2008 ACM SIGPLAN workshop on Memory systems performance and correctness (MSPC'08)* (2008), ACM, pp. 16–19.

[9] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B. C., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (2009).

[10] CUPPU, V., JACOB, B., DAVIS, B., AND MUDGE, T. A performance comparison of contemporary DRAM architectures. In *the 26th International Symposium on Computer Architecture* (1999).

[11] DANOWITZ, A., KELLEY, K., MAO, J., STEVENSON, J. P., AND HOROWITZ, M. Cpu db: recording microprocessor history. *Communications of the ACM 55*, 4 (2012), 55–63.

[12] DONG, M., AND CHEN, H. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference* (Santa Clara, CA, 2017), USENIX Association, pp. 719–731.

[13] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)* (2014), pp. 15:1–15:15.

[14] ELLEN, F., FATAOUROU, P., RUPPERT, E., AND VAN BREUGEL, F. Non-blocking binary search trees. In *the 29th ACM Symposium on Principles of Distributed Computing (PODC)* (2010).

[15] FOMITCHEV, M., AND RUPPERT, E. Lock-free linked lists and skiplists. In *the 23rd ACM Symposium on Principles of Distributed Computing (PODC)* (2004).

[16] HU, Q., REN, J., BADAM, A., AND MOSCIBRODA, T. Log-structured non-volatile main memory. In *Proceedings of the USENIX Annual Technical Conference* (2017).

[17] HUAI, Y. Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects. *AAPPS bulletin 18*, 6 (2008), 33–40.

[18] INTEL. Intel® 64 Architecture Memory Ordering White Paper, August 2007. SKU 318147-001.

[19] KIM, C., CHHUGANI, J., SATISH, N., SEDLAR, E., NGUYEN, A. D., KALDEWEY, T., LEE, V. W., BRANDT, S. A., AND DUBEY, P. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2010).

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

[21] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journling of journal anomaly in Android I/O: Multi-version B-tree with lazy split. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)* (2014).

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

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

[24] LEE, W., LEE, K., SON, H., KIM, W.-H., NAM, B., AND WON, Y. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the 2015 USENIX Anual Technical Conference* (2015).

[25] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems 6*, 4 (1981), 650–670.

[26] LEHMAN, T. J., AND CAREY, M. J. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB)* (1986).

[27] LEIS, V., KEMPER, A., AND NEUMANN, T. Exploiting hardware transactional memory in main-memory databases. In *Proceedings of the 30th International Conference on Data Engineering (ICDE)* (2014).

[28] LEVANDOSKI, J. J., LOMET, D. B., AND SENGUPTA, S. The Bw-Tree: a B-tree for new hardware platforms. In *Proceedings of the 29th International Conference on Data Engineering (ICDE)* (2013).

[29] LU, Y., SHU, J., AND SUN, L. Blurred persistence in transactional persistent memory. In *Proceedings of the 31st International Conference on Massive Storage Systems and Technology (MSST)* (2015).

[30] MCKENNEY, P. E. Memory ordering in modern microprocessors. *Linux Journal 136*, Aug. (2005).

[31] MCKUSICK, M. K., GANGER, G. R., ET AL. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track* (1999), pp. 1–17.

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

[33] MORARU, I., ANDERSEN, D. G., KAMINSKY, M., TOLIA, N., BINKERT, N., AND RANGANATHAN, P. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *Proceedings of the ACM Conference on Timely Results in Operating Systems (TRIOS)* (2013).

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

[35] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)* (2014), pp. 265–276.

[36] RAO, J., AND ROSS, K. A. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)* (1999).

[37] RAO, J., AND ROSS, K. A. Making B+-trees cache conscious in main memory. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2000).

[38] RIXNER, S., DALLY, W. J., KAPASI, U. J., MATTSON, P., AND OWENS, J. D. Memory access scheduling. In *ACM SIGARCH Computer Architecture News* (2000), vol. 28, ACM, pp. 128–138.

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

[40] SEWELL, P., SARKAR, S., OWENS, S., NARDELLI, F. Z., AND MYREEN, M. O. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM 53*, 7 (2010), 89–97.

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

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

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

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

[45] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Proc. of USENIX NSDI 2013* (Apr 2013).

[46] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using restricted transactional memory to build a scalable in-memory database. In *ACM SIGOPS/Eurosys European Conference on Computer Systems (EuroSys)* (2014).

[47] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (2015).

[48] WONG, H.-S. P., RAOUX, S., KIM, S., LIANG, J., REIFENBERG, J. P., RAJENDRAN, B., ASHEGHI, M., AND GOODSON, K. E. Phase change memory. *Proceedings of the IEEE 98*, 12 (2010), 2201–2227.

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

RFLUSH: Rethink the Flush

Jeseong Yeon*, Minseong Jeong*, Sungjin Lee†, Eunji Lee*‡
*Chungbuk National University, †DGIST, ‡University of Wisconsin–Madison
{jsyeon, msjeong}@oslab.cbnu.ac.kr, sungjin.lee@dgist.ac.kr, eunji@cbnu.ac.kr

Abstract

A FLUSH command has been used for decades to enforce persistence and ordering of updates in a storage device. The command forces *all* the data in the volatile buffer of the storage device to non-volatile media to achieve persistency. This *lump-sum* approach to flushing has two performance consequences. First, it slows down non-volatile materialization of the writes that actually need to be made durable. Second, it deprives the writes that do not need to be made durable of an opportunity for absorbing future writes and coalescing.

We attempt to characterize the problems of this *semantic gap* of flushing in storage devices and propose RFLUSH that allows a fine-grained control over non-volatile materialization. The RFLUSH command delivers a range of logical block addresses (LBAs) that need to be flushed and thus enables the storage device to force only a subset of data in its buffer.

We implemented this fine-grained flush command in a storage device using an open-source flash development platform and modified the F2FS file system to make use of the command in processing fsync requests as a case study. Performance evaluation using the prototype shows that the inclusion of RFLUSH improves the throughput by up to 6.5x; reduces the write traffic by up to 43%; and eliminates the long tail in the response time.

1 Introduction

Historically, storage devices have made use of a volatile buffer for various purposes. For hard disk drives (HDDs), the volatile buffer has been used for absorbing writes and minimizing seeks, while solid state drives (SSDs) have used the buffer for improving their random write performance and masking the limited endurance of the underlying non-volatile media [6, 13, 15, 19, 38, 39, 44].

The adoption of a volatile buffer, however, can bring with it data loss and improper ordering of updates in a power outage. The FLUSH command has been introduced to resolve this issue; forcing *all* the pending writes to non-volatile media, ensuring persistence and proper serialization of updates.

Unfortunately, this *lump-sum* approach to enforcing persistency has undesired performance consequences [6, 13, 38, 39, 44]. To faithfully implement the flush semantics, the storage device must empty all the dirty pages in its volatile buffer, whereas a flush request is commonly issued with less stringent requirements. As an example, consider a concurrent execution of two applications: an on-line banking application that requires to persist each transaction immediately, and a big-data analytics application that writes a large amount of intermediate results, which is a common scenario in modern complicated and multi-tenant storage platforms. In this scenario, a flush request for a committed transaction by the banking application will end up with forcing a large amount of dirty data (most of it from the analytics application, and thus irrelevant) in the storage device, which slows down what is actually needed (forcing the dirty data from the banking application).

This paper attempts to cure the performance problem of the conventional flush mechanism outlined above by refactoring the storage device interface. The refactoring is to include a command called RFLUSH (Range Flush) which allows a fine-grained control over non-volatile materialization of dirty data in the buffer. The RFLUSH command transfers a range of logical block addresses (LBAs) that specifies data to be persisted with it, helping the storage device to optimize its non-volatile materialization. This command not only speeds up the non-volatile materialization of the target LBAs but also enhances buffering and coalescing of other dirty data in the buffer.

Our work is in line with a collection of recent studies. In the past, computer systems have been built upon a standard block device interface consisting of a small set of commands over its logical address space: read, write, and flush. This abstract view of a storage device allows a host system to readily access non-volatile media in an efficient manner. However, as emerging storage media such as flash memory and other non-volatile memories (NVMs) are more commonly used, the possibility of extending the conventional block device interface to leverage the full potential of the new storage media is being actively explored [1, 4, 7, 9, 23, 24, 27, 31, 32, 36, 46].

A TRIM command has been proposed to prevent useless data from being copied around and lowering the endurance of flash memory [36]. As another example, recent storage device interfaces support atomic writes, which can be efficiently supported in flash-based storage

devices [7, 31]. Also, storage interface extensions such as those for delegating block allocation [1, 9, 27, 32, 46], multi-streamed SSDs [18, 29], host manageable storage devices [4, 23], and user programmable SSDs [35] have been studied to provide an extended functionality and/or achieve better performance in high-end storage systems.

The benefits of RFLUSH seem straightforward, but realizing it efficiently in a storage device and augmenting file systems and/or database systems to make an effective use of it are not without challenges. We implemented RFLUSH in a storage device using an open-source flash development platform [23] and modified a file system (F2FS) [22] to make use of the extended interface[1]. The modified F2FS uses the RFLUSH command in the handling of fsync (and its variants). In this way, user applications do not need to be modified since the interface (i.e., fsync) and its semantics are faithfully preserved.

The rest of this paper is organized as follows. We give our motivation for RFLUSH and briefly review the related technology trends (§2). We then present the RFLUSH command and describe its prototype implementation (§3). We present results from performance evaluation using the prototype (§4), and finally conclude (§5).

2 Motivation and Related Work

Flush Optimization using Non-volatile Memory: Many prior works have pointed out that the in-storage buffer flush is a critical contributor to performance variation and unexpected slowdown in storage devices [15, 19]. One approach to lessening the detrimental performance effects of flush is to use super capacitors for providing enough energy to force all the dirty data in the volatile buffer at the time of a power outage. SSD manufacturers incorporate super capacitors in their high-end SSD devices to make them tolerant on power outages, offering high performance and reliability at the same time [21]. As a similar approach, Xiangfeng presents a modern SSD architecture that uses non-volatile memory for a write buffer while maintaining a read cache as volatile [44]. However, both approaches intrinsically increase the manufacturing cost, resulting in lower competitiveness of the intended products. The two approaches are, however, complementary to RFLUSH in the sense that they allow the RFLUSH command to return immediately while giving priority for replacement to those dirty data that were the target of the command to make room in the buffer for future writes.

Flush Optimization in a Host: The problem of flushing mechanism also exists in a page cache between a host and a device, because the page cache adopts a flushing mechanism to ensure persistence and ordering of updates in its volatile buffer. As opposed to a storage intreface, POSIX file system interfaces provide fine-grained control over the flushing mechanism through fsync and fdatasync system calls, in addition to sync. However, the flushing activity is still costly in a larger size page cache, and thus there have been numerous studies to mitigate this problem. Likewise as on the storage side, specialized hardware such as battery-backed main memory has been considered to avoid flushing cost [5, 43]. As a software-based approach, Nightingale et al. present an externally synchronized file system called xsyncfs [30], which allows an application to avoid blocking during the long-latency synchronization. The xsyncfs allows a requesting application to immediately return from the synchronization request, but makes the updates visible when they become consistently durable, leading to improvement in responsiveness. Chidambaram et al. present a new crash-consistency protocol that decouples ordering and durability, thereby providing data consistency with high performance [6]. Instead of forcing a low-level disk promptly to flush its buffer, they allow a storage device to optimize a flushing mechanism within a time limit, while still satisfying the ordering constraints. While such optimization obtained through a trade-off between durability and performance is worthwhile to consider in storage interface extension, this paper, as an initial and fundamental approach, focuses on storage interfaces for enhancing performance without any compromising of durability.

SSD Trends: The demand to improve the flush interface is particularly high at this moment because the cost of a flush is amplified when it is combined with next-generation SSD technologies. As host interfaces such as the NVMe [10, 16] become fast, the performance bottleneck is being shifted from the host interface to the flash device. The flash memory latencies for reading, programming, and erasing are also steadily increasing although device density is improving. Therefore, the latest SSDs attempt to use an increasingly larger buffer (e.g., 512 MB to 2 GB) to compensate for flash memory's low performance and endurance [25, 33, 34, 37]. With this trend, it is obvious that cache flushing results in more serious performance degradations in the presence of a larger buffer.

Besides, there are SSDs that exploit a portion of the host memory as a dedicated in-storage buffer, which may seriously suffer from cache flushing. Such SSDs help to improve the performance while cutting off the cost by not using DRAM in the storage device [8, 33], but a tandem with a classical flush interface might incur GBs of data being flushed from the host to the storage device on a regular basis. Considering the high cost of data transfer between the host and the device, the existing flush mechanism would degrade the storage performance severely.

[1] https://github.com/jsyeon92/RFLUSH

Also, the page size of flash memory is getting bigger, which will affect the overall performance as an eager flushing forfeits the possibility of consolidation and realignment of pending writes, yielding a large number of underutilized pages [20].

High Demand on Isolation The need for improving the flush interface is also evident with respect to performance isolation. With the latest innovations of data centers, computation is rapidly being moved from standalone desktops to cloud systems. With this trend, performance isolation and accurate accounting across applications are more important than ever. Techniques for isolating storage performance on the host side have been researched extensively. IceFS isolates related data with a container-based grouping and eliminates shared physical resources or access dependencies among containers in a file system [13]. Differentiated Storage Services (DSS) [26] and IOFlow [41] propose to tag data across layers to determine which process issues a request at any given layer. Yang et al. present a split-level I/O scheduling framework that provides a set of hooks for acquiring knowledge needed for accurate accounting and fair scheduling [45].

However, not much research has been performed on the storage side to prevent interference among applications. Prior works on in-storage buffers mostly focus on the replacement policy [15, 19], and there is not much previous research on curing the inefficiency of the flush mechanism despite its huge impact on the performance and endurance of the storage device. We believe our analysis and proposal in this paper are highly timely and contribute to driving the storage interface to be in harmony with fast-advancing storage technologies.

3 Range Flush

The concept of RFLUSH is simple but there are many design issues to be addressed since it involves from the application down to the storage device. In §3.1, we discuss places where RFLUSH can be useful. Then, in §3.2, we explain how to identify data related to RFLUSH. Data associated with RFLUSH is not limited to user data but includes metadata. In §3.3, we discuss how to handle metadata for RFLUSH. We describe how to integrate RFLUSH into storage protocols in §3.4.

3.1 Where to Use RFLUSH

Since RFLUSH is more general than its counterpart FLUSH and allows finer-grained control over what to flush, there can be many use cases where it can be effective. In this paper, we focus on its use for optimizing the fsync and fdatasync system calls. (Hereafter we use fsync to denote both fsync and fdatasync.)

There are some obvious benefits in implementing fsync using RFLUSH. First, no application modifications are needed since the fsync semantics can be faithfully preserved. Second, information about the user data and metadata that are affected by the fsync is readily available. Third, there can be noticeable performance gains from isolating regions to flush by fsync.

Although we leave for future research the use of RFLUSH by the file system itself other than in the processing of the fsync, we can easily identify other potential use cases for RFLUSH. For example, many file systems use journaling for recovery purposes and they typically use write-ahead logging (WAL) [28] that requires logging be performed before the logged updates are written to their home locations. The RFLUSH command can be used to give priority to the non-volatile materialization of data in the log. The same write-ahead logging is used by almost all the database systems today and they can be equally benefited by the use of RFLUSH.

3.2 How to Identify the Associated Data

The next challenge in using RFLUSH lies in how to identify the associated data for a given fsync request. The file system needs to identify the set of pages that are associated with a file and thus has to be forced to persist. Among such pages, some are in the page cache in a dirty state. The file system can flush such pages to the storage device followed by an RFLUSH command targeting them. A problematic case is when some of the pages that need to be forced to persist have already been sent to storage, meaning that they can be either in a clean state or evicted from a page cache. Unfortunately, it is overly intricate to keep track of such data blocks, but if they are missing, the semantics of the fsync system call can be violated.

We address this challenge by specifying *whole* data blocks of a file. This approximation is made efficient by fundamental file-system design principles; most file systems allocate data blocks for a file as consecutively as possible so as to benefit from spatial locality [14]. This idea has been adopted to reduce the seek time for HDDs, but it holds true for SSDs as well since a high degree of spatial locality means better performance for SSDs because it allows for more efficient address translation and interleaving over multiple channels/chips in the SSD. With this policy, the data blocks of a file are likely to be encoded by only a few extents, which means only a small number of RFLUSH commands are needed.

However, this might not always be the case because there could be more fragmentations over time, in particular for larger files. To address this, our final design choice is to transfer the *inode number* of the target file, instead of a set of LBAs. This approach can faithfully preserve the fsync semantics, without excessive over-

head needed to specify the range of data blocks to persist with RFLUSH. The implementation details of the inode-based RFLUSH protocol will be described in Section 3.4.

3.3 How to Handle Metadata

One thing that must not be overlooked is to flush file system metadata that has a dependency on the target file of the fsync; otherwise, there is a danger of data corruption or loss on a system crash. We explain using the F2FS file system as an example. The on-disk layout of F2FS has two areas; *metadata area* and *main area*. The metadata area keeps information for file system maintenance such as block allocation bitmaps and orphan inode lists [22]. In contrast, the main area is used to store normal data blocks and file metadata including inode and indirect blocks. Upon a write request, a set of blocks needs to be updated in an atomic manner to provide crash consistency [3]. Specifically, since F2FS is a log-structured file system, it allocates and updates a new data block out-of-place, requiring the updating of related metadata (i.e., inode) and indirect blocks to properly point to the new block. In turn, the block allocation bitmap and several tables that maintain information for space management should also be updated. This behavior leads to many small random writes to blocks containing the file system metadata; encoding of those writes as a set of ranges would be complicated. To get away with this complication, we decide to encode a full range of the metadata area, which is a superset of metadata to be updated, and send it along with the RFLUSH command.

This approximation seems to have a problem when fsync requests from multiple files are interfered with each other because their metadata shares a single LBA. Consider a case in which there are two different files *A* and *B*, whose inode structures are located in a single block. When the fsync requests occur for the files concurrently, forcing the entire metadata area by one fsync request might corrupt data integrity, violating ordering constraints between data and metadata of another file (e.g., file *B*'s metadata is persisted before file B's data block).

However, this is not the case because current file systems are carefully designed so as not to let this happen. For example, F2FS logs individual inode structure on an update, instead of an entire block, thereby preventing undesired interference that can be caused by interleaved fsyncs. Ext4 resolves this issue by forcing all dependant data prior to persisting the modified metadata block. Thus, in the above example, both data A and B are flushed to non-volatile storage before the metadata block when an fsync request occurs either for file A or B.

3.4 How to Integrate into a Storage Protocol

To make use of the RFLUSH primitive, the host interface should be extended. While this extension is difficult to be incorporated into mature storage interfaces such as SATA [12] or SAS [17], it is a viable option for emerging storage interfaces like NVMe [10, 16] to add proprietary extensions. Another possibility for incorporating extensions into the standard storage API is to use the open-channel SSD architecture [4, 23]. In this architecture, the host system implements many of the functionalities needed to manage flash memory (e.g., garbage collection). Also, by utilizing veiled information behind the storage device interface, this architecture enables the management of flash memory to meet the demands of the host system. We use the latter approach since the host-manageable architecture allows easy integration of the extensions for RFLUSH.

Our prototyping system implements the inode-based RFLUSH protocol through storage interface extension and F2FS file system modification. We add the range flush protocol to BlueDBM, which is an open-channel flash development platform from MIT [23], facilitating the construction of a host-manageable storage device. Specifically, we extend the host storage interface to support the RFLUSH primitive in which the inode number is encoded. Then, we augment the in-storage buffer handler in the FTL to locate the associated data blocks and flush them selectively upon an RFLUSH request. The buffer handler maintains the pending updates in a hash table using an inode number as a key. Note that this mechanism requires a write command that also includes an inode number such that the device controller determines which file the data block belongs to. However, the open-channel SSD half of which the FTL runs on the host side can easily determine this by referencing the kernel data structure with the transferred write request, which is used in our implementation.

On the host side, F2FS, the modified file system, communicates with BlueDBM through a block device interface and makes use of the RFLUSH primitive in implementing the fsync system call. When an fsync request arrives from the application, F2FS writes all dirty pages of the requested file and the associated metadata from a page cache to a storage device. Then, F2FS issues a pair of RFLUSH commands that include the inode numbers associated with the target file and the metadata area. The RFLUSH command is forwarded to the storage device controller through the underlying block I/O layer and device driver where a host side component of BlueDBM runs. BlueDBM completes the RFLUSH request by forcing writes associated with the given inode number.

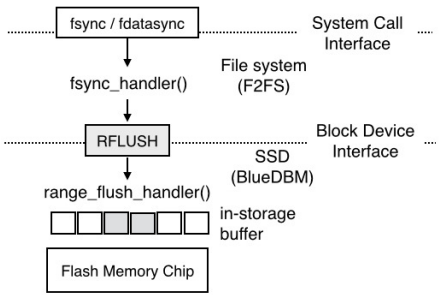


Figure 1: `RFLUSH` **protocol implementation.**

Configuration	Settings
Page / Block size	4KB / 64 Pages
Read / Write Latency	100us / 1300us
Block Erase Latency	1.5ms
Data Transfer Latency	100us (for 4KB)
Overprovisioning Ratio	3%
SSD capacity	37 GB
In-storage Buffer	256 MB / 1 GB

Table 1: **SSD platform setup.**

4 Performance Evaluation

We evaluate the proposed `RFLUSH` using a prototype implementation. The next section explains our evaluation methodology. In §4.2, we report results on the effectiveness of `RFLUSH` from experiments using both micro- and macro-benchmarks.

4.1 Methodology

We modified both the file system (F2FS) [22] and the storage device (BlueDBM) [23] to implement the `RFLUSH` protocol in Linux 4.7.2. Figure 1 shows the architecture of our experimental platform. When the user issues an `fsync` request through the system call interface, the `sync_handler` module inside the file system generates `RFLUSH` commands to BlueDBM. The `range_flush_handler` module within the FTL of BlueDBM handles the request by forcing the associated data from its volatile buffer to the non-volatile media.

Our experiments were performed on Intel Core i7 running at 3.3GHz with 64GB of DDR4 memory. The detailed configurations of BlueDBM are given in Table 1. To understand the performance consequence of the `RFLUSH` primitive, we first evaluate the prototype using a micro-benchmark based on FIO [11], which generates a synthetic workload that models a best-case scenario for `RFLUSH`. Then, we use a set of macro-benchmarks to examine the effectiveness of `RFLUSH` in a real environment. In our experiments, the storage device is accessed in a

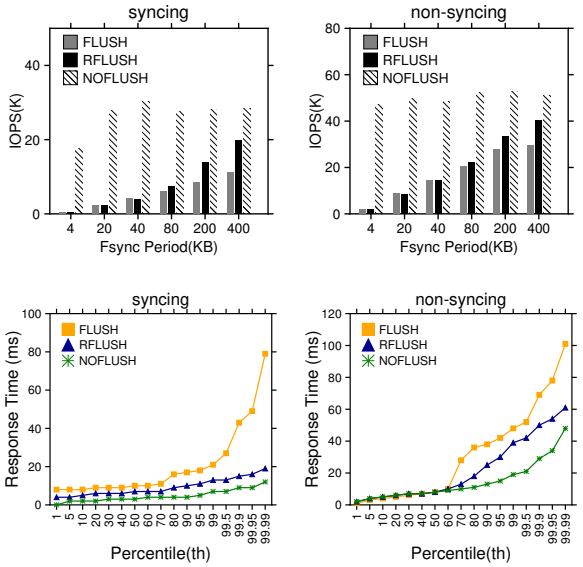


Figure 2: **IOPS and response time distributions of the micro-benchmark.** *Figures in the top row show IOPS for syncing and non-syncing threads with a 1GB storage buffer. The X-axis is the amount of data written between the invocations of fsync. The use of `RFLUSH` improves IOPS by up to 1.74x and 1.36x for the syncing and non-syncing threads, respectively, compared to using `FLUSH`. The two graphs in the bottom row give a percentile response time for both the syncing and non-syncing threads when fsync period is 400KB. The 99.99th percentile response time is reduced from 79.36us to 19.38us when `RFLUSH` is used instead of `FLUSH` in a syncting thread.*

Benchmark	Write #	Avg. Size	fsync Interval / #
Fileserver	1536K	1MB	None
TPC-C	2.2K	16KB	21373us / 13448
Linkbench	101K	16KB	10016us / 14097

Table 2: **Macro-benchmark characteristics.**

direct mode (unless otherwise specified) to observe the behavior of `RFLUSH` more clearly in a controlled environment. The performance is measured five times for each scenario and their median is reported.

4.2 Experimental Results

Micro-Benchmark: To assess the potential performance gain made possible by `RFLUSH`, we used a micro-benchmark based on FIO [11] that approximates a typical scenario where there is a mixture of asynchronous and synchronous writes. The micro-benchmark consists of both *syncing* and *non-syncing* threads. Both types of thread perform the same task except for their syncing behavior. Both write 2GB data randomly to a file with a 4KB granularity in a direct mode. The difference is a syncing thread issues an `fsync` request after writing a

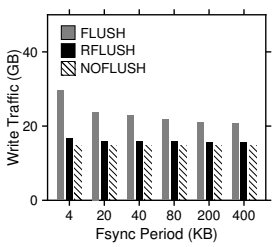


Figure 3: **Write traffic from the micro-benchmark.** *The write traffic is measured at the interface between the in-storage buffer and the flash memory when the buffer size is 1GB. RFLUSH reduces write traffic by 24% to 43% for the fsync periods we considered.*

given amount of data.

In the experiment, there were one syncing thread and 12 non-syncing threads, and we measured their performances for three possible configurations: FLUSH, RFLUSH, and NOFLUSH. The FLUSH configuration forces to flash memory all data in the volatile buffer of the storage device, while the RFLUSH configuration forces only the data in a given LBA range. In the NOFLUSH configuration, the storage device ignores all the sync requests. In all configurations, if the number of dirty pages in the buffer is above a threshold (90% here), a certain number of pages are written-back to flash memory by a background activity in the storage device. Figure 2 shows the performance of both the syncing and non-syncing threads in terms of IOPS and response time. In the figures of the top row, the X-axis is the amount of data written before the syncing thread issues an fsync request.

The results show that there is a large performance improvement for the syncing thread when RFLUSH is used instead of FLUSH. This performance improvement is mainly due to the fact that the flushing activities of the syncing thread are not interfered by the flushing of non-urgent writes from non-syncing threads when RFLUSH is used. For the same reason, RFLUSH also eliminates a long tail in the response time distribution for the syncing thread, which is critical to providing a consistent performance from a storage device.

The results also show that even the performance of non-syncing threads is improved. When RFLUSH is used, a prioritized flushing of data written by the syncing thread gives more time for the dirty data from non-syncing threads to reside in the buffer. The increased time in the buffer allows them to absorb more writes to the same LBA and also to be coalesced more with other writes, resulting in a better performance. As a result, RFLUSH reduces the write traffic significantly compared to FLUSH as Figure 3 illustrates. Its result even comes close to that of NOFLUSH. In this scenario, each three of the 12 non-syncing threads access the same file, while a

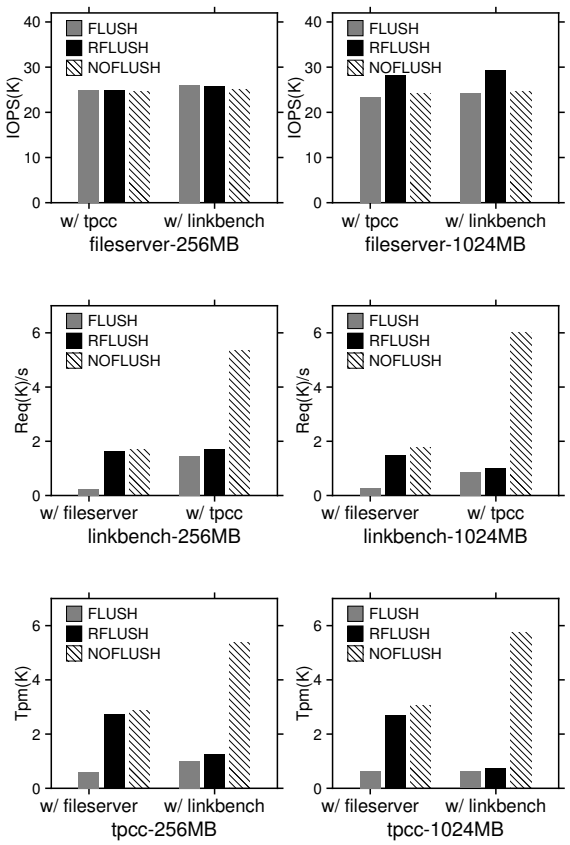


Figure 4: **Performance of mixed real workloads in a direct mode.** *These figures show IOPS for each pair of benchmarks when a storage buffer size is 256MB and 1024MB. TPC-C and Linkbench achieves 5.3x to 6.5x and 4.1x to 4.5x higher IOPS with RFLUSH when running together with Fileserver. Fileserver also delivers 20% higher IOPS with RFLUSH when executing with TPC-C and Linkbench. When TPC-C and Linkbench are mixed, their performances are improved by 1.4x to 1.6x and 1.17x to 1.3x, respectively.*

syncing thread accesses its own file. Thus, the writes of the non-syncing thread have a locality. F2FS basically updates the data in an out-of-place manner, but it allows overwrite once the data is copied for updates after the last checkpoint, unless the explicit fsync request occurs. Therefore, F2FS benefits from the enhanced buffering effect of the RFLUSH primitive in the writes of non-syncing threads.

A somewhat non-intuitive result is that when the fsync requests are issued too frequently, in some extreme cases RFLUSH even performs worse than FLUSH even though the former results in much less write traffic to the storage device. Careful analysis over the results reveals that if fsyncs are too frequent, the performance is dominated by fsyncs rather than the actual write traffic associated with them.

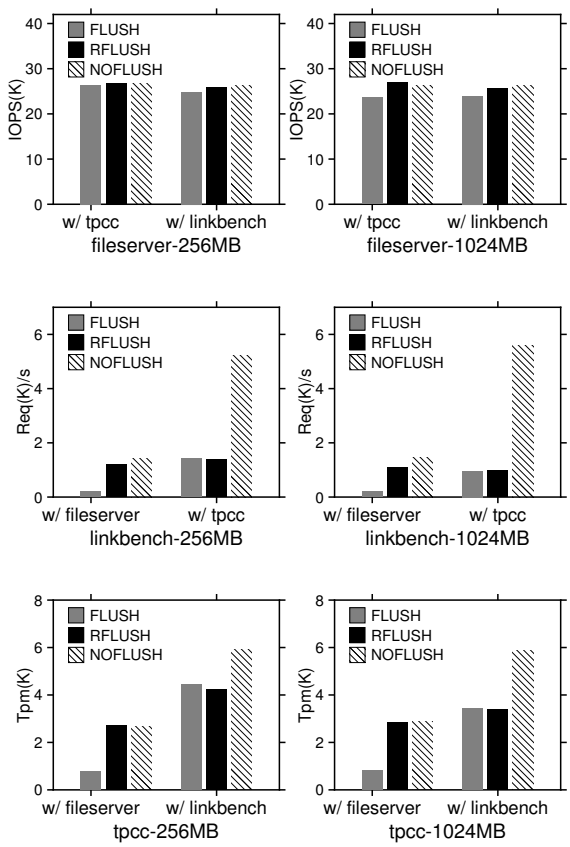


Figure 5: **Performance of mixed real workloads in a buffered mode.** *These figures report the performance with the page cache turned on. Although the absolute values are different, the results show the same general trends as in a direct mode (cf. Figure 4).*

Macro-Benchmarks: To assess the performance impact of RFLUSH in the real world, we selected three macro-benchmarks (Fileserver, Linkbench, and TPC-C) and measured their performances when a pair of them run concurrently. Fileserver generates a large number of asynchronous writes acting like a multi-streaming server [40]. Linkbench is a graph processing application based on the Facebook Social Graph, containing a few kilobytes of writes with frequent sync requests [2]. TPC-C is an on-line transaction processing benchmark which issues small-sized random writes with frequent synchronization [42]. Table 2 summarizes various statistics about the three macro-benchmarks.

Figure 4 shows the results in terms of IOPS for each pair of the three macro-benchmarks. The results show that the performance improvement by RFLUSH is most noticeable when asynchronous and synchronous workloads are mixed, as in the micro-benchmark we considered in the previous section. For example, TPC-C and Linkbench show 4.5x and 6.5x higher IOPS with RFLUSH, when they run together with Fileserver, which is consistent with the micro-benchmark results in the previous section.

The results also show that there are performance improvements even in the case where both benchmarks contain synchronous workloads. For example, when TPC-C and Linkbench are running together, RFLUSH improves performance by up to 1.4x and 1.29x in TPC-C and Linkbench, respectively. This result is due to time-multiplexed non-volatile materializations for fsyncs from the two benchmarks. One counter-intuitive observation is that an RFLUSH outperforms a NOFLUSH in a mixture of Fileserver and TPC-C/Linkbench with a 1024MB buffer. This improvement comes from that an RFLUSH replenishes free space more quickly by proactively writing back the buffered data on a synchronization request, which helps the efficent handling of the bulky writes generated from Fileserver.

We also performed the same experiments in a buffered mode (i.e., with the page cache turned on). Figure 5 reports the performance in the same format as in Figure 4. Although the absolute values are different, the results show the same general trends as in a direct mode shown in Figure 4. The performance gap between RFLUSH and FLUSH is reduced because of periodic flushing from the page cache but the difference is only marginal.

5 Conclusion

In this paper, we raised an issue about the negative performance impact of a lump-sum approach to persisting buffered data within a storage device and presented RFLUSH that allows a fine-grained persistence control. We implemented an RFLUSH prototype by modifying a file system (F2FS) in Linux 4.7.2 as well as a storage device based upon an open-source flash development platform. Performance evaluation using the prototype shows that RFLUSH increases overall I/O performance by up to 6.5x, and eliminates a long tail latency of synchronous writes.

6 Acknowledgments

We thank Ming Zhao (our shepherd) and the anonymous reviewers for their insightful comments. This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2017R1D1A1B03031494) and by the Ministry of Science, ICT & Future Planning (No. 2014R1A1A3053505 and No. NRF-2017R1E1A1A01077410).

References

[1] ANAND, A., SEN, S., KRIOUKOV, A., POPOVICI, F., AKELLA, A., ARPACI-DUSSEAU, A., ARPACI-DUSSEAU, R., AND BANERJEE, S. Avoiding file system micromanagement with range writes. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), OSDI, USENIX Association, pp. 161–176.

[2] ARMSTRONG, T. G., PONNEKANTI, V., BORTHAKUR, D., AND CALLAGHAN, M. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ICMD, ACM, pp. 1185–1196.

[3] ARPACI-DUSSEAU, R. H., AND ARPACI-DUSSEAU, A. C. *Three Easy Pieces*. Arpaci-Dusseau Books, 2015.

[4] BJØRLING, M., GONZÁLEZ, J., AND BONNET, P. Lightnvm: The linux open-channel ssd subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST, pp. 359–374.

[5] CHEN, P. M., NG, W. T., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The rio file cache: Surviving operating system crashes. *Acm Sigplan Notices 31*, 9 (1996), 74–83.

[6] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP, ACM, pp. 228–243.

[7] COBURN, J., BUNKER, T., SCHWARZ, M., GUPTA, R., AND SWANSON, S. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the 24th ACM symposium on operating systems principles* (2013), SOSP, ACM, pp. 197–212.

[8] DORGELO. Host memory buffer based ssd systems. Flash Memory Summi, 2015.

[9] DUBITZKY, Z., GOLD, I., HENIS, E., SATRAN, J., AND SHEINWALD, D. Dsf: Data sharing facility. *Technical report* (2002).

[10] ESHGHI, K., AND MICHELONI, R. Ssd architecture and pci express interface. In *Inside Solid State Drives (SSDs)*. Springer, 2013, pp. 19–45.

[11] FIO. Fio benchmark. https://github.com/axboe/fio.git, 2017.

[12] GRIMSRUD, K., AND SMITH, H. *Serial ATA Storage Architecture and Applications: Designing High-Performance, Cost-Effective I/O Solutions*. Intel press, 2003.

[13] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The harey tortoise: Managing heterogeneous write performance in ssds. In *USENIX Annual Technical Conference* (2013), ATC, pp. 79–90.

[14] HE, J., NGUYEN, D., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Reducing File System Tail Latencies with Chopper. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Santa Clara, CA, February 2015), FAST.

[15] HUANG, S.-M., AND CHANG, L.-P. Exploiting page correlations for write buffering in page-mapping multichannel ssds. *ACM Transactions on Embedded Computing Systems 15*, 1 (2016), 12.

[16] HUFFMAN, A. Nvm express: Going mainstream and whats next. Intel Developers Forum, 2014.

[17] JACKSON, M. *SAS Storage Architecture*. MindShare Press, 2005.

[18] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems* (Philadelphia, PA, 2014), HotStorage.

[19] KIM, H., AND AHN, S. Bplru: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST, pp. 1–14.

[20] KIM, M., LEE, J., LEE, S., PARK, J., AND KIM, J. Improving performance and lifetime of large-page nand storages using erase-free subpage programming. In *Proceedings of the 54th Annual Design Automation Conference 2017* (2017), DAC, ACM, p. 24.

[21] LAPEDUS, M. Sorting out next-gen memory. http://semiengineering.com/sorting-out-next-gen-memory/, 2016.

[22] LEE, C., SIM, D., HWANG, J. Y., AND CHO, S. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (2015), FAST, pp. 273–286.

[23] LEE, S., LIU, M., JUN, S. W., XU, S., KIM, J., AND ARVIND, A. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (2017), FAST, pp. 339–353.

[24] MARKS, K. An nvm express tutorial. Flash Memory Summit, 2013.

[25] MARVELL. Conservative use of dram. http://www.anandtech.com/show/9942/marvell-implements-host-memory-buffer-for-dramless-88nv1140-ssd-controller, 2016.

[26] MESNIER, M., CHEN, F., LUO, T., AND AKERS, J. B. Differentiated storage services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), SOSP, ACM, pp. 57–70.

[27] MIN, C., KANG, W.-H., KIM, T., LEE, S.-W., AND EOM, Y. I. Lightweight application-level crash consistency on transactional flash storage. In *USENIX Annual Technical Conference* (2015), ATC, pp. 221–234.

[28] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems 17*, 1 (Mar. 1992), 94–162.

[29] NAM, E. H., KIM, B. S. J., EOM, H., AND MIN, S. L. Ozone (o3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers 60*, 5 (2011), 653–666.

[30] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. *ACM Transactions on Computer Systems (TOCS) 26*, 3 (2008), 6.

[31] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND PANDA, D. K. Beyond block i/o: Rethinking traditional storage primitives. In *The 17th IEEE International Symposium on High Performance Computer Architecture* (2011), HPCA, IEEE, pp. 301–311.

[32] PRABHAKARAN, V., RODEHEFFER, T. L., AND ZHOU, L. Transactional flash. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), OSDI, pp. 147–160.

[33] SAMSUNG. Samsung ssd 850 pro. http://www.samsung.com, 2016.

[34] SANDISK. Sandisk extreme pro ssd. http://www.anandtech.com/show/8170/sandisk-extreme-pro-240gb-480gb-960gb-review, 2016.

[35] SESHADRI, S., GAHAGAN, M., BHASKARAN, M. S., BUNKER, T., DE, A., JIN, Y., LIU, Y., AND SWANSON, S. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX conference on Operating systems design and implementation* (2014), OSDI, pp. 67–80.

[36] SHU, F., AND OBR, N. Data set management commands proposal for ata8-acs2, revision 1 ed. *Microsoft Corporation, One Microsoft Way, Redmond, WA* (2012), 98052–6399.

[37] SKHYNIX. Sk hynix se3010 enterprise ssd review. http://www.tomsitpro.com/articles/sk-hynix-se3010-enterprise-ssd-review,2-977-2.html, 2016.

[38] SOLWORTH, J. A., AND ORJI, C. U. Write-only disk caches. *ACM SIGMOD Record 19*, 2 (1990), 123–132.

[39] STEIGERWALD, M. Imposing order. *Linux Magazine, May* (2007).

[40] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *USENIX; login 41* (2016).

[41] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. Ioflow: a software-defined storage architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP, ACM, pp. 182–196.

[42] TPCC. Tpcc-mysql benchmark. https://github.com/Percona-Lab/tpcc-mysql, 2017.

[43] WANG, A.-I., REIHER, P. L., POPEK, G. J., AND KUENNING, G. H. Conquest: Better performance through a disk/persistent-ram hybrid file system. In *USENIX Annual Technical Conference* (2002), ATC, pp. 15–28.

[44] XIANGFENG, L. IO Pattern based Optimization in SSD. Flash Memory Summit, 2016.

[45] YANG, S., HARTER, T., AGRAWAL, N., KOWSALYA, S. S., KRISHNAMURTHY, A., AL-KISWANY, S., KAUSHIK, R. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Split-level i/o scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles* (2015), SOSP, ACM, pp. 474–489.

[46] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based ssds with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST, pp. 1–16.

Barrier-Enabled IO Stack for Flash Storage

Youjip Won[1] Jaemin Jung[2*] Gyeongyeol Choi[1]
Joontaek Oh[1] Seongbae Son[1] Jooyoung Hwang[3] Sangyeun Cho[3]
[1]*Hanyang University* [2]*Texas A&M University* [3]*Samsung Electronics*

Abstract

This work is dedicated to eliminating the overhead required for guaranteeing the *storage order* in the modern IO stack. The existing block device adopts a prohibitively expensive approach in ensuring the storage order among write requests: interleaving the write requests with *Transfer-and-Flush*. Exploiting the cache barrier command for Flash storage, we overhaul the IO scheduler, the dispatch module, and the filesystem so that these layers are orchestrated to preserve the ordering condition imposed by the application with which the associated data blocks are made durable. The key ingredients of Barrier-Enabled IO stack are *Epoch-based IO scheduling*, *Order-Preserving Dispatch*, and *Dual-Mode Journaling*. Barrier-enabled IO stack can control the storage order without Transfer-and-Flush overhead. We implement the barrier-enabled IO stack in server as well as in mobile platforms. SQLite performance increases by 270% and 75%, in server and in smartphone, respectively. In a server storage, BarrierFS brings as much as by 43× and by 73× performance gain in MySQL and SQLite, respectively, against EXT4 via relaxing the durability of a transaction.

1 Motivation

The modern Linux IO stack is a collection of the arbitration layers; the IO scheduler, the command queue manager, and the writeback cache manager shuffle the incoming requests at their own disposal before passing them to the next layers. Despite the compound uncertainties from the multiple layers of arbitration, it is essential for the software writers to enforce a certain order in which the data blocks are reflected to the storage surface, *storage order*, in many cases such as in guaranteeing the durability and the atomicity of a database transaction [47, 26, 35], in filesystem journaling [67, 41, 65, 4], in soft-update [42, 63], or in copy-on-write or log-structure filesystems [61, 35, 60, 31]. Enforcing a storage order is achieved by an extremely expensive approach: dispatching the following request only after the data block associated with the preceding request is completely transferred to the storage device and is made durable. We call this mechanism a *Transfer-and-Flush*. For decades, interleaving the write requests with a *Transfer-and-Flush* has been the fundamental principle to guarantee the storage order in a set of requests [24, 16].

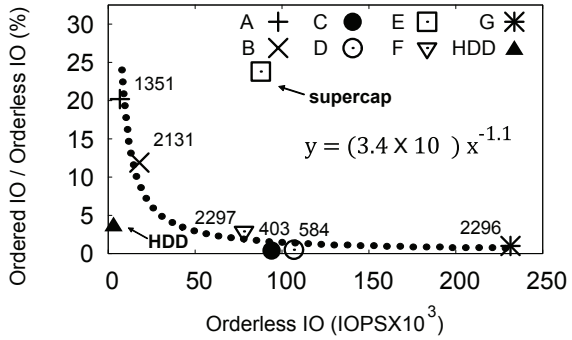


Figure 1: Ordered write vs. Orderless write, Except 'HDD', all are Flash storages; A: (1ch)/eMMC5.0, B: (1ch)/UFS2.0, C: (8ch)/SATA3.0, D: (8ch)/NVMe, E: (8ch)/SATA3.0 (supercap), F: (8ch)/PCIe, G: (32ch) Flash array, The number next to each point is the IOPS of `write()` followed by `fdatasync()`.

We observe a phenomenal increase in the performance and the capacity of the Flash storage. The performance increase owes much to the concurrency and the parallelism in the Flash storage, e.g. the multi-channel/way controller [73, 6], the large size storage cache [48], and the deep command queue [19, 27, 72]. A state of the art NVMe SSD reportedly exhibits up to 750 KIOPS random read performance [72]. It is nearly 4,000× of a HDD's performance. The capacity increase is due to the adoption of the finer manufacturing process (sub-10 nm) [25, 36], and the multi-bits per cell (MLC, TLC, and QLC) [5, 11]. Meanwhile, the time to program a Flash cell has barely improved, and is even deteriorating in some cases [22].

The Transfer-and-Flush based order-preserving mechanism conflicts with the parallelism and the concurrency in the Flash storage. It disables the parallelism and the concurrency feature of the Flash storage and exposes the raw Flash cell programming latency to the host. The

*This work was done while the author was a graduate student at Hanyang University.

overhead of the Transfer-and-Flush mechanism will become more significant as the Flash storage employs a higher degree of parallelism and the denser Flash device. Fig. 1 illustrates an important trend. We measure the sustained throughput of orderless random write (plain buffered write) and the ordered random write in EXT4 filesystem. In ordered random write, each write request is followed by `fdatasync()`. X-axis denotes the throughput of orderless write which corresponds to the rate at which the storage device services the write requests at its full throttle. This usually matches the vendor published performance of the storage device. The number next to each point denotes the sustained throughput of the ordered write. The Y-axis denotes the ratio between the two. In a single channel mobile storage for smartphone (SSD A), the performance of ordered write is 20% of that of unordered write (1351 IOPS vs. 7000 IOPS). In a thirty-two channel Flash array (SSD G), this ratio decreases to 1% (2296 IOPS vs. 230K IOPS). In SSD with supercap (SSD E), the ordered write performance is 25% of that of the unordered write. The Flash storage uses supercap to hide the flush latency from the host. Even in a Flash storage with supercap, the overhead of Transfer-and-Flush is significant.

Many researchers have attempted to address the overhead of storage order guarantee. The techniques deployed in the production platforms include non-volatile writeback cache at the Flash storage [23], `no-barrier` mount option at the EXT4 filesystem [15], and transactional checksum [56, 32, 64]. Efforts such as transactional filesystem [50, 18, 54, 35, 68] and transactional block device [30, 74, 43, 70, 52] save the application from the overhead of enforcing the storage order associated with filesystem journaling. A school of work address more fundamental aspects in controlling the storage order, such as separating the ordering guarantee from durability guarantee [9], providing a programming model to define the ordering dependency among the set of writes [20], and persisting a data block only when the result needs to be externally visible [49]. Despite their elegance, these works rely on Transfer-and-Flush when it is required to enforce the storage order. OptFS [9] relies on Transfer-and-Flush in enforcing the order between the journal commit and the associated checkpoint. Featherstitch [20] relies on Transfer-and-Flush to implement the ordering dependency between the *patchgroups*.

In this work, we revisit the issue of eliminating the Transfer-and-Flush overhead in the modern IO stack. We develop a *Barrier-Enabled IO stack*, in which the filesystem can issue the following request before the preceding request is serviced and yet the IO stack can enforce the storage order between them. The barrier-enabled IO stack consists of the cache barrier-aware storage device, the order-preserving block device layer, and the barrier enabled filesystem. For cache barrier-aware storage device, we exploit the "cache barrier" command [28]. The barrier-enabled IO stack is built upon the foundation that the host can control a certain partial order in which the cache contents are flushed. The "cache barrier" command precisely serves this purpose. For the order-preserving block device layer, the command dispatch mechanism and the IO scheduler are overhauled so that the block device layer ensures that the IO requests from the filesystem are serviced preserving a certain partial order. For the barrier-enabled filesystem, we define new interfaces, `fbarrier()` and `fdatabarrier()`, to separate the ordering guarantee from the durability guarantee. They are similar to `fsync()` and `fdatasync()`, respectively, except that they return without waiting for the associated blocks to become durable. We modify EXT4 for the order-preserving block device layer. We develop dual-mode journaling for the order-preserving block device. Based upon the dual-mode journaling, we newly implement `fbarrier()` and `fdatabarrier()` and rewrite `fsync()`.

Barrier-enabled IO stack removes the flush overhead as well as the transfer overhead in enforcing the storage order. While large body of the works have focused on eliminating the flush overhead, few works have addressed the overhead of *DMA transfer* to enforce the storage order. The benefits of the barrier-enabled IO stack include the followings;

- The application can control the storage order virtually without any overheads, including the flush overhead, DMA transfer overhead, and context switch.
- The latency of a journal commit decreases significantly. The journaling module can enforce the storage order between the journal logs and the journal commit block without interleaving them with flush or with DMA transfer.
- Throughput of the filesystem journaling improves significantly. The dual-mode journaling commits multiple transactions concurrently and yet can guarantee the durability of the individual journal commit.

By eliminating the Transfer-and-Flush overhead, the barrier-enabled IO stack successfully exploits the concurrency and the parallelism in modern Flash storage.

2 Background

2.1 Orders in IO stack

A write request travels a complicated route until the data blocks reach the storage surface. The filesystem puts the request to the IO scheduler queue. The block device driver removes one or more requests from the queue and constructs a command. It probes the device and dispatches the command if the device is available.

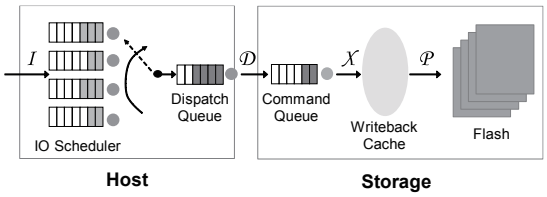


Figure 2: IO stack Organization

The device is available if the command queue is not full. The storage controller inserts the incoming command at the command queue. The storage controller removes the command from the command queue and services it (i.e. transfers the associated data block between the host and the storage). When the transfer finishes, the device signals the host. The contents of the writeback cache are committed to the storage surface either periodically or by an explicit request from the host.

We define four types of orders in the IO stack; *Issue Order*, $\mathscr{I}$, *Dispatch Order*, $\mathscr{D}$, *Transfer Order*, $\mathscr{X}$, and *Persist Order*, $\mathscr{P}$. The issue order $\mathscr{I} = \{i_1, i_2, \ldots, i_n\}$ is a set of write requests issued by the file system. The subscript denotes the order in which the requests enter the IO scheduler. The dispatch order $\mathscr{D} = \{d_1, d_2, \ldots, d_n\}$ denotes a set of the write requests dispatched to the storage device. The subscript denotes the order in which the requests leave the IO scheduler. The transfer order, $\mathscr{X} = \{x_1, x_2, \ldots, x_n\}$, is the set of transfer completions. The persist order, $\mathscr{P} = \{p_1, p_2, \ldots, p_n\}$, is a set of operations that make the data blocks in the writeback cache durable. We say that a partial order is preserved if the relative position of the requests against a designated request, *barrier*, are preserved between two different types of orders. We use the notation '=' to denote that a partial order is preserved. The partial orders between the different types of orders may not coincide due to the following reasons.

- $\mathscr{I} \neq \mathscr{D}$. The IO scheduler reorders and coalesces the IO requests subject to the scheduling principle, e.g. CFQ, DEADLINE, etc. When there is no scheduling mechanism, e.g. NO-OP scheduler [3] or NVMe [13] interface, the dispatch order may be equal to the issue order.
- $\mathscr{D} \neq \mathscr{X}$. The storage controller can freely schedule the commands in its command queue. In addition, the commands can be serviced out-of-order due to the errors, the time-outs, and the retry.
- $\mathscr{X} \neq \mathscr{P}$. The writeback cache of the storage is not FIFO. In Flash storage, persist order is governed not by the order in which the data blocks are made durable but by the order in which the associated mapping table entries are updated. The two may not coincide.

Due to all these uncertainties, the modern IO stack is said to be *orderless* [8].

2.2 Transfer-and-Flush

Enforcing a storage order corresponds to preserving a partial order between the order in which the filesystem issues the requests, $\mathscr{I}$, and the order in which the associated data blocks are made durable, $\mathscr{P}$. It is equivalent to collectively enforcing the partial orders between the pair of the orders in the adjacent layers in Fig. 2. It can be formally represented as in Eq. 1.

$$(\mathscr{I} = \mathscr{P}) \equiv (\mathscr{I} = \mathscr{D}) \wedge (\mathscr{D} = \mathscr{X}) \wedge (\mathscr{X} = \mathscr{P}) \quad (1)$$

The modern IO stack has evolved under the assumption that the host cannot control the persist order, i.e. $\mathscr{X} \neq \mathscr{P}$. This is due to the physical characteristics of the rotating media. For rotating media such as HDDs, a persist order is governed by disk scheduling algorithm. The disk scheduling is entirely left to the storage controller due to its complicated sector geometry which is hidden from outside [21]. When the host blindly enforces a certain persist order, it may experience anomalous delay in IO service. Due to this constraint of $\mathscr{X} \neq \mathscr{P}$, Eq. 1 is unsatisfiable. The constraint that the host cannot control the persist order is a fundamental limitation in modern IO stack design.

The block device layer adopts the indirect and the expensive approach to control the storage order in spite of the constraint $\mathscr{X} \neq \mathscr{P}$. First, after dispatching the write command to the storage device, the caller postpones dispatching the following command until the preceding command is serviced, i.e. until the associated DMA transfer completes. We refer to this mechanism as *Wait-on-Transfer*. Wait-on-Transfer mechanism ensures that the commands are serviced in order and to satisfy $\mathscr{D} = \mathscr{X}$. Wait-on-Transfer is expensive; it blocks the caller and interleaves the requests with DMA transfer. Second, when the preceding command is serviced, the caller issues the flush command and waits for its completion. The caller issues the following command only after the flush command returns. This is to ensure that the associated data blocks are persisted in order and to satisfy $\mathscr{X} = \mathscr{P}$. We refer to this mechanism as *Wait-on-Flush*. The modern block device layer uses Wait-on-Transfer and Wait-on-Flush in pair when it needs to enforce the storage order between the write requests. We call this mechanism as *Transfer-and-Flush*.

The cost of Transfer-and-Flush is prohibitive. It neutralizes the internal parallelism of the Flash storage controller, stalls the command queue, and exposes the caller to DMA transfer and raw cell programming delays.

2.3 Analysis: `fsync()` in EXT4

We examine how the EXT4 filesystem controls the storage order in an `fsync()`. In Ordered journaling mode (default), the data blocks are persisted before the journal

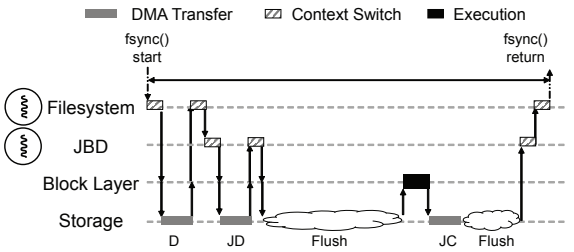


Figure 3: DMA, flush, and context switches in `fsync()`, 'D', 'JC' and 'JC' denote the DMA transfer time for *D*, *JD* and *JC*, respectively. 'Flush' denotes the time to service the flush request.

transaction. Fig. 3 illustrates the behavior of an `fsync()`. The filesystem issues the write requests for a set of dirty pages, *D*. *D* may consist of the data blocks from different files. After issuing the write requests, the application thread blocks waiting for the completion of the DMA transfer. When the DMA transfer completes, the application thread resumes and triggers the JBD thread to commit the journal transaction. After triggering the JBD thread, the application thread sleeps again. When the JBD thread makes journal transaction durable, the `fsync()` returns. It should be emphasized that the application thread triggers the JBD thread only after *D* is transferred. Otherwise, the storage controller may service the write request for *D* and the write requests for journal commit in an out-of-order manner, and the storage controller may persist the journal transaction prematurely (before *D* is transferred).

A journal transaction is usually committed using two write requests: one for writing the coalesced chunk of the journal descriptor block and the log blocks and the other for writing the commit block. In the rest of the paper, we will use *JD* and *JC* to denote the coalesced chunk of the journal descriptor and the log blocks, and the commit block, respectively. In committing a journal transaction, JBD needs to enforce the storage orders in two relations: within a transaction and between the transactions. Within a transaction, JBD needs to ensure that *JD* is made durable ahead of *JC*. Between the journal transactions, JBD has to ensure that journal transactions are made durable in order. When either of the two conditions are violated, the file system may recover incorrectly in case of unexpected failure [67, 9]. For the storage order within a transaction, JBD interleaves the write request for *JD* and the write request for *JC* with Transfer-and-Flush. To control the storage order between the transactions, JBD thread waits for *JC* to become durable before it starts committing the following transaction. JBD uses Transfer-and-Flush mechanism in enforcing both intra-transaction and inter-transaction storage order.

In earlier days of Linux, the block device layer explicitly issued a flush command in committing a journal transaction [15]. In this approach, the flush command blocks not only the caller but also the other requests in the same dispatch queue. Since Linux 2.6.37, the filesystem (JBD) implicitly issues a flush command [16]. In writing *JC*, JBD tags the write request with `REQ_FLUSH` and `REQ_FUA`. Most storage controllers have evolved to support these two flags; with these two flags, the storage controller flushes the writeback cache before servicing the command and in servicing the command it directly persists *JC* to storage surface bypassing the writeback cache. In this approach, only the JBD thread blocks and the other threads that share the same dispatch queue can proceed. Our effort can be thought as a continuation to this evolution of the IO stack. We mitigate the Transfer-and-Flush overhead by making the storage device more capable: supporting a barrier command and by redesigning the host side IO stack accordingly.

3 Order-Preserving Block Device Layer

3.1 Design

The order-preserving block device layer consists of the newly defined barrier write command, order-preserving dispatch module, and Epoch-based IO scheduler. We overhaul the IO scheduler, the dispatch module, and the write command so that they can preserve the partial order between the different types of orders, $\mathscr{I} = \mathscr{D}$, $\mathscr{D} = \mathscr{X}$, and $\mathscr{X} = \mathscr{P}$, respectively. Order-preserving dispatch module eliminates the Wait-on-Transfer overhead and the newly defined barrier write command eliminates the wait-on-flush overhead. They collectively together preserve the partial order between the issue order $\mathscr{I}$ and the persist order $\mathscr{P}$ without Transfer-and-Flush.

The order-preserving block device layer categorizes the write requests into two categories, *orderless* write and *order-preserving* write. The order-preserving requests are the ones that are subject to the storage ordering constraint. Orderless request is the one which is irrelevant to the ordering dependency and which can be scheduled freely. We distinguish the two to avoid imposing unnecessary ordering constraint in scheduling the requests. The details are to come shortly. We refer to a set of the order-preserving requests that can

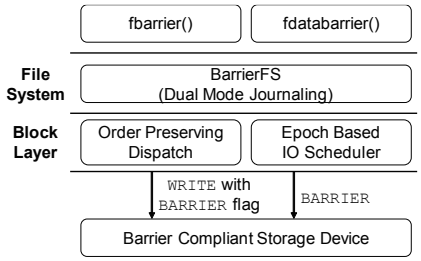


Figure 4: Organization of the barrier-enabled IO stack

be reordered with each other as an *epoch* [14]. We define a special type of order-preserving write as a *barrier* write. A barrier write is used to delimit an epoch. We introduce two new attributes `REQ_ORDERED` and `REQ_BARRIER` for the `bio` object and the `request` object to represent an order-preserving write and a barrier write. `REQ_ORDERED` attribute is used to specify the order-preserving write. Barrier write request has both `REQ_ORDERED` and `REQ_BARRIER` attributes. The order-preserving block device layer handles the request differently based upon its category. Fig. 4 illustrates the organization of Barrier-Enabled IO stack.

3.2 `Barrier Write`, the Command

The "cache barrier," or "barrier" for short, command is defined in the standard command set for mobile Flash storage [28]. With barrier command, the host can control the persist order without explicitly invoking the cache flush. When the storage controller receives the barrier command, the controller guarantees that the data blocks transferred before the barrier command becomes durable after the ones that follow the barrier command do. A few eMMC products in the market support cache barrier command [1, 2]. The barrier command can satisfy the condition $\mathscr{X} = \mathscr{P}$ in Eq. 1 which has been unsatisfiable for several decades due to the mechanical characteristics of the rotating media. The naive way of using barrier is to replace the existing flush operation [66]. This simple replacement still leaves the caller under the Wait-on-Transfer overhead to enforce the storage order.

Implementing a barrier as a separate command occupies one entry in the command queue and costs the host the latency of dispatching a command. To avoid this overhead, we define a barrier as a command flag. We designate one unused bit in the SCSI command for a barrier flag. We set the barrier flag of the write command to make itself a barrier write. When the storage controller receives a barrier write command, it services the barrier write command as if the barrier command has arrived immediately following the write command.

With reasonable complexity, the Flash storage can be made to support a barrier write command [30, 57, 39]. When the Flash storage has Power Loss Protection (PLP) feature, e.g. a supercapacitor, the writeback cache contents are guaranteed to be durable. The storage controller can flush the writeback cache fully utilizing its parallelism and yet can guarantee the persist order. In Flash storage with PLP, we expect that the performance overhead of the barrier write is insignificant.

For the devices without PLP, the barrier write command can be supported in three ways; in-order writeback, transactional writeback, or in-order recovery. In in-order writeback, the storage controller flushes the data blocks in epoch granularity. The amount of data blocks in an epoch may not be large enough to fully utilize the parallelism of the Flash storage. The in-order writeback style of the barrier write implementation can bring the performance degradation in cache flush. In transactional writeback, the storage controller flushes the writeback cache contents as a single unit [57, 39]. Since all epochs in the writeback cache are flushed together, the persist order imposed by the barrier command is satisfied. The transactional writeback can be implemented without any performance overhead if the controller exploits the spare area of the Flash page to represent a set of pages in a transaction [57]. The in-order recovery method relies on a crash recovery routine to control the persist order. When multiple controller cores concurrently write the data blocks to multiple channels, one may have to use sophisticated crash recovery protocol such as ARIES [46] to recover the storage to consistent state. If the entire Flash storage is treated as a single log device, we can use simple crash recovery algorithm used in LFS [61]. Since the persist order is enforced by the crash recovery logic, the storage controller can flush the writeback cache at the full throttle as if there is no ordering dependency. The controller is saved from performance penalty at the cost of complexity in the recovery routine.

In this work, we modify the firmware of the UFS storage device to support the barrier write command. We use a simple LFS style in-order recovery scheme. The modified firmware is loaded at the commercial UFS product of the Galaxy S6 smartphone[1]. The modified firmware treats the entire storage device as a single log structured device. It maintains an active segment in memory. FTL appends incoming data blocks to the active segment in the order in which they are transferred. When an active segment becomes full, the controller stripes the active segment across the multiple Flash chips in log-structured manner. In crash recovery, the UFS controller locates the beginning of the most recently flushed segment. It scans the pages in the segment from the beginning till it encounters the page that has not been programmed successfully. The storage controller discards the rest of the pages including the incomplete one.

Developing a barrier-enabled SSD controller is an engineering exercise. It is governed by a number of design choices and should be addressed in a separate context. In this work, we demonstrate that the performance benefit achieved by the barrier command well deserves its complexity if the host side IO stack can properly exploit it.

3.3 Order-Preserving Dispatch

Order-preserving dispatch is a fundamental innovation in this work. In order-preserving dispatch, the block device

[1] Some of the authors are firmware engineers at Samsung Electronics and have an access to the FTL firmware of Flash storage products.

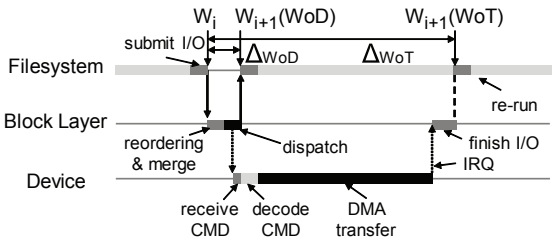


Figure 5: Wait-on-Dispatch vs Wait-on-Transfer, W_i: i^{th} write request, W_{i+1}(WoD): $(i+1)^{th}$ write request under Wait-on-Dispatch, W_{i+1}(WoT): $(i+1)^{th}$ write request under Wait-on-Transfer

layer dispatches the following command immediately after it dispatches the preceding one (Fig. 5) and yet the host can ensure that the two commands are serviced in order. We refer to this mechanism as *Wait-on-Dispatch*. The order-preserving dispatch is to satisfy the condition $\mathscr{D} = \mathscr{X}$ in Eq. 1 without Wait-on-Transfer overhead.

The dispatch module constructs a command from the requests. The dispatch module constructs the barrier write command when it encounters the barrier write request, i.e. the write request with `REQ_ORDERED` and `REQ_BARRIER` flags. For the other requests, it constructs the commands as it used to do in the legacy block device.

Implementing an order-preserving dispatch is rather simple; the block device driver sets the priority of a barrier write command as *ordered*. Then, the SCSI compliant storage device services the command satisfying the ordering constraint. The following is the reason. SCSI standard defines three command priority levels: *head of the queue*, *ordered*, and *simple* [59]. With each, the storage controller puts the incoming command at the head of the command queue, at the tail of the command queue or at an arbitrary position determined at its disposal, respectively. The default priority is *simple*. The command with *simple* priority cannot be inserted in front of the existing *ordered* or *head of the queue* command. Exploiting the command priority of existing SCSI interface, the order-preserving dispatch module ensures that the barrier write is serviced only after the existing requests in the command queue are serviced and before any of the commands that follow the barrier write are serviced.

The device can temporarily be unavailable or the caller can be switched out involuntarily after dispatching a write request. The order-preserving dispatch module uses the same error handling routine of the existing block device driver; the kernel daemon inherits the task and retries the failed request after a certain time interval, e.g. 3 msec for SCSI devices [59]. The *ordered* priority command has rarely been used in the existing block device implementations. This is because when the host cannot control the persist order, enforcing a transfer order with *ordered* priority command barely carries any meaning from the perspective of ensuring the storage order. In the emergence of the barrier write, the ordered priority plays an essential role in making the entire IO stack an order-preserving one.

The importance of order-preserving dispatch cannot be emphasized further. With order-preserving dispatch, the host can control the transfer order without releasing the CPU and without stalling the command queue. IO latency can become more predictable since there exists less chance that the CPU scheduler interferes with the caller's execution. Δ_{WoT} and Δ_{WoD} in Fig. 5 illustrate the delays between the consecutive requests in Wait-on-Transfer and Wait-on-Dispatch, respectively. In Wait-on-Dispatch, the host issues the next request $W_{i+1}(WoD)$ immediately after it issues W_i. In Wait-on-Transfer, the host issues the next request $W_{i+1}(WoT)$ only after W_i is serviced. Δ_{WoD} is an order of magnitude smaller than Δ_{WoT}.

3.4 Epoch-Based IO scheduling

Epoch-based IO scheduling is designed to preserve the partial order between the issue order and the dispatch order. It satisfies the condition $\mathscr{I} = \mathscr{D}$. It is designed with three principles; (i) it preserves the partial order between the epochs, (ii) the requests within an epoch can be freely scheduled with each other, and (iii) an orderless request can be scheduled across the epochs.

When an IO request enters the scheduler queue, the IO scheduler determines if it is a barrier write. If the request is a barrier write, the IO scheduler removes the barrier flag from the request and inserts it into the queue. Otherwise, the scheduler inserts it to the queue as is. When the scheduler inserts a barrier write to the queue, it stops accepting more requests. Since the scheduler blocks the queue after it inserts the barrier write, all order-preserving requests in the queue belong to the same epoch. The requests in the queue can be freely re-ordered and merged with each other. The IO scheduler uses the existing scheduling discipline, e.g. CFQ. The merged request will be order-preserving if one of the components is order-preserving request. The IO scheduler designates the last order-preserving request that leaves the queue as a new barrier write. This mechanism is called *Epoch-Based Barrier Reassignment*. When there are not any order-preserving requests in the queue, the IO scheduler starts accepting the IO requests again. When the IO scheduler unblocks the queue, there can be one or more orderless requests in the queue. These orderless requests are scheduled with the requests in the following epoch. Differentiating orderless requests from the order-preserving ones, we avoid imposing unnecessary ordering constraint on the irrelevant requests.

Fig. 6 illustrates an example. The circle and the rectangle that enclose the write request denote the order-preserving flag and barrier flag, respectively. An

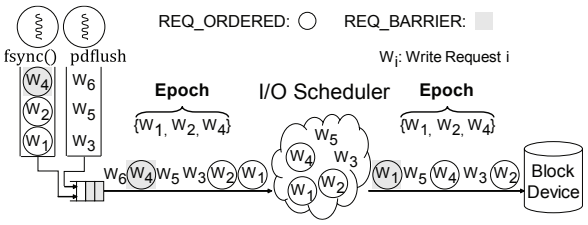


Figure 6: Epoch Based Barrier Reassignment

`fdatasync()` creates three write requests: w_1, w_2, and w_4. The barrier-enabled filesystem, which will be detailed shortly, marks the write requests as ordering preserving ones. The last request, w_4, is designated as a barrier write and an epoch, $\{w_1, w_2, w_4\}$, is established. A `pdflush` creates three write requests w_3, w_5, and w_6. They are all orderless writes. The requests from the two threads are fed to the IO scheduler as $w_1, w_2, w_3, w_5, w_4^{barrier}, w_6$. When the barrier write, w_4, enters the queue, the scheduler stops accepting the new request. Thus, w_6 cannot enter the queue. The IO scheduler reorders the requests in the queue and dispatches them as $w_2, w_3, w_4, w_5, w_1^{barrier}$ order. The IO scheduler relocates the barrier flag from w_4 to w_1. The epoch is preserved after IO scheduling.

The order-preserving block device layer now satisfies all three conditions, $\mathscr{I} = \mathscr{D}, \mathscr{D} = \mathscr{X}$ and $\mathscr{X} = \mathscr{P}$ in Eq. 1 with an Epoch-based IO scheduling, an order-preserving dispatch and a barrier write, respectively. The order-preserving block device layer successfully eliminates the Transfer-and-Flush overhead in controlling the storage order and can control the storage order with only Wait-on-Dispatch overhead.

4 Barrier-Enabled Filesystem

4.1 Programming Model

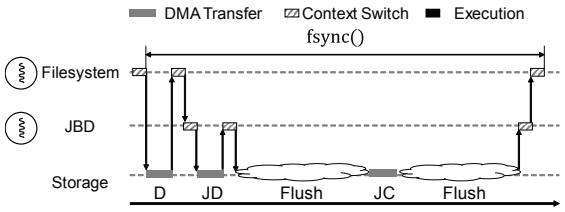
The barrier-enabled IO stack offers four synchronization primitives: `fsync()`, `fdatasync()`, `fbarrier()`, and `fdatabarrier()`. We propose two new filesystem interfaces, `fbarrier()` and `fdatabarrier()`, to separately support ordering guarantee. `fbarrier()` and `fdatabarrier()` synchronize the same set of blocks with `fsync()` and `fdatasync()`, respectively, but they return without ensuring that the associated blocks become durable. `fbarrier()` bears the same semantics as `osync()` in OptFS [9] in that it writes the data blocks and the journal transactions in order but returns without ensuring that they become durable.

`fdatabarrier()` synchronizes the modified blocks, but not the journal transaction. Unlike `fdatasync()`, `fdatabarrier()` returns without persisting the associated blocks. `fdatabarrier()` is a generic storage barrier. By interleaving the `write()` calls with `fdatabarrier()`, the application ensures that the data blocks associated with the write requests that precede `fdatabarrier()` are made durable ahead of the data blocks associated with the write requests that follow `fdatabarrier()`. It plays the same role as `mfence` for memory barrier [53]. Refer to the following codelet. Using `fdatabarrier()`, the application ensures that the "world" is made durable only after "Hello" does.

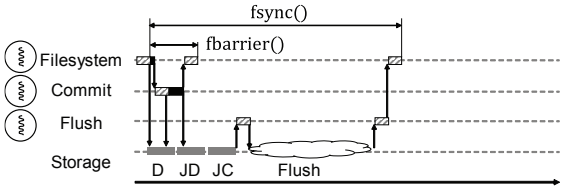
```
write(fileA, "Hello") ;
fdatabarrier(fileA) ;
write(fileA, "World") ;
```

The order-preserving block device layer is filesystem agnostic. In our work, we modify EXT4 for barrier enabled IO stack.

4.2 Dual Mode Journaling



(a) `fsync()` in EXT4; JBD writes JC with `FLUSH/FUA`. The latter 'Flush' for persisting 'JC' directly to the storage surface.



(b) `fsync()` and `fbarrier()` in BarrierFS

Figure 7: Details of `fsync()` and `fbarrier()`

Committing a journal transaction essentially consists of two saparate tasks: (i) dispatching the write commands for *JD* and *JC* and (ii) making *JD* and *JC* durable. Exploiting the order-preserving nature of the underlying block device, we physically separate the control plane activity (dispatching the write requests) and the data plane activity (persisting the associated data blocks and journal transaction) of a journal commit operation. Further, we allocate the separate threads to each task so that the two activities can proceed in parallel with minimum dependency. The two threads are called as *commit* thread and *flush* thread, respectively. We refer to this mechanism as *Dual Mode Journaling*. Dual Mode Journaling mechanism can support two journaling modes, durability guarantee mode and ordering guarantee mode, in versatile manner.

The commit thread is responsible for dispatching the write requests for *JD* and *JC*. The commit thread writes

each of the two with a barrier write so that *JD* and *JC* are persisted in order. The commit thread dispatches the write requests without any delay in between (Fig. 7(b)). In EXT4, JBD thread interleaves the write request for *JC* and *JD* with Transfer-and-Flush (Fig. 7(a)). After dispatching the write request for *JC*, the commit thread inserts the journal transaction to the committing transaction list and hands over the control to the flush thread.

The flush thread is responsible for (i) issuing the flush command, (ii) handling error and retry and (iii) removing the transaction from the committing transaction list. The behavior of the flush thread varies subject to the durability requirement of the journal commit. If the journal commit is triggered by `fbarrier()`, the flush thread returns after removing the transaction from the committing transaction list. It returns without issuing the flush command. If the journal commit is triggered by `fsync()`, the flush thread involves more steps. It issues a flush command and waiting for the completion. When the flush completes, it removes the the associated transaction from the committing transaction list and returns. BarrierFS supports all journal modes in EXT4; WRITEBACK, ORDERED and DATA.

The dual thread organization of BarrierFS journaling bears profound implications in filesystem design. First, the separate support for the ordering guarantee and the durability guarantee naturally becomes an integral part of the filesystem. Ordering guarantee involves only the control plane activity. Durability guarantee requires the control plane activity as well as data plane activity. BarrierFS partitions the journal commit activity into two independent components, control plane and data plane and dedicates separate threads to each. This modular design enables the filesystem primitives to adaptively adjust the activity of the data plane thread with respect to the durability requirement of the journal commit operation; `fsync()` vs. `fbarrier()`. Second, the filesystem journaling becomes concurrent activity. Thanks to the dual thread design, there can be multiple committing transactions in flight. In most journaling filesystems that we are aware of, the filesystem journaling is a serial activity; the journaling thread commits the following transaction only after the preceding transaction becomes durable. In dual thread design, the commit thread can commit a new journal transaction without waiting for the preceding committing transaction to become durable. The flush thread asynchronously notifies the application thread about the completion of the journal commit.

4.3 Synchronization Primitives

In `fbarrier()` and `fsync()`, BarrierFS writes *D*, *JD*, and *JC* in a piplelined manner without any delays in between (Fig. 7(b)). BarrierFS writes *D* with one or more order-preserving writes whereas it writes *JD* and *JC* with the barrier writes. In this manner, BarrierFS forms two epochs $\{D, JD\}$ and $\{JC\}$ in an `fsync()` or in an `fbarrier()` and ensures the storage order between these two epochs. `fbarrier()` returns when the filesystem dispatches the write request for *JC*. `fsync()` returns after it ensures that *JC* is made durable. Order-preserving block device satisfies prefix constraint [69]. When *JC* becomes durable, the order-preserving block device guarantees that all blocks associated with preceding epochs have been made durable. An application may repeatedly call `fbarrier()` committing multiple transactions simultaneously. By writing *JC* with a barrier write, BarrierFS ensures that these committing transactions become durable in order. The latency of an `fsync()` reduces significantly in BarrierFS. It reduces the number of flush operations from two in EXT4 to one and eliminates the Wait-on-Transfer overheads (Fig. 7).

In `fdatabarrier()` and `fdatasync()`, BarrierFS writes *D* with a barrier write. If there are more than one write requests in writing *D*, only the last one is set as a barrier write and the others are set as the order-preserving writes. An `fdatasync()` returns after the data blocks, *D*, become durable. An `fdatabarrier()` returns immediately after dispatching the write requests for *D*. `fdatabarrier()` is the crux of the barrier-enabled IO stack. With `fdatabarrier()`, the application can control the storage order virtually without any overheads: without waiting for the flush, without waiting for DMA completion, and even without the context switch. `fdatabarrier()` is a very light-weight storage barrier.

An `fdatabarrier()` (or `fdatasync()`) may not find any dirty pages to synchronize upon its execution. In this case, BarrierFS explicitly triggers the journal commit. It forces BarrierFS to issue the barrier writes for *JD* and *JC*. Through this mechanism, `fdatabarrier()` or `fdatasync()` can delimit an epoch as desired by the application even in the absence of any dirty pages.

4.4 Handling Page Conflicts

A buffer page may have been held by the committing transaction when an application tries to insert it to the running transaction. We refer to this situation as *page conflict*. Blindly inserting a conflict page into the running transaction yields its removal from the committing transaction before it becomes durable. The EXT4 filesystem checks for the page conflict when it inserts a buffer page to the running transaction [67]. If the filesystem finds a conflict, the thread delegates the insertion to the JBD thread and blocks. When the committing transaction becomes durable, the JBD thread identifies the conflict pages in the committed transaction and inserts them to the running transaction. In EXT4, there can be at most one committing transaction. The running transaction is

guaranteed to be free from page conflict when the JBD thread has made it durable and finishes inserting the conflict pages to the running transaction.

In BarrierFS, there can be more than one committing transactions. The conflict pages may be associated with different committing transactions. We refer to this situation as *multi-transaction page conflict*. As in EXT4, BarrierFS inserts the conflict pages to the running transaction when it makes a committing transaction durable. However, to commit a running transaction, BarrierFS has to scan all buffer pages in the committing transactions for page conflicts and ensure that it is free from any page conflicts. When there exists large number of committing transactions, the scanning overhead to check for the page conflict can be prohibitive in BarrierFS.

To reduce this overhead, we propose the *conflict-page list* for a running transaction. The conflict-page list represents the set of conflict pages associated with a running transaction. The filesystem inserts the buffer page to the conflict-page list when it finds that the buffer page that it needs to insert to the running transaction is subject to the page conflict. When the filesystem has made a committing transaction durable, it removes the conflict pages from the conflict-page list in addition to inserting them to the running transaction. A running transaction can only be committed when the conflict-page list is empty.

4.5 Concurrency in Journaling

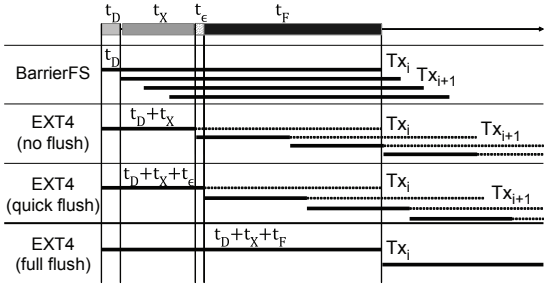


Figure 8: Concurrency in filesystem journaling under varying storage order guarantee mechanisms, t_D: dispatch latency, t_X: transfer latency, t_ε: flush latency in supercap SSD, t_F: flush latency

We examine the degree of concurrency in journal commit operation under different storage order guarantee mechanisms: BarrierFS, EXT4 with `no-barrier` option (EXT4 (no flush)), EXT4 with supercap SSD (EXT4 (quick flush)), and plain EXT4 (EXT4 (full flush)). With `no-barrier` mount option, the JBD thread omits the flush command in committing a journal transaction. With this option, the EXT4 guarantees neither durability nor ordering in journal commit operation since the storage controller may make the data blocks durable out-of-order. We examine this configuration to illustrate the filesystem journaling behavior when the flush command is removed in the journal commit operation.

In Fig. 8, each horizontal line segment represents a journal commit activity. It consists of the solid line segment and the dashed line segment. The end of the horizontal line segment denotes the time when the transaction reaches the disk surface. The end of the solid line segment represents the time when the journal commit returns. If they do not coincide, it means that the journal commit finishes before the transaction reaches the disk surface. In EXT4 (full flush), EXT4 (quick flush), and EXT4 (no flush), the filesystem commits a new transaction only after preceding journal commit finishes. The journal commit is a serial activity. In EXT4 (full flush), the journal commit finishes only after all associated blocks are made durable. In EXT4 (quick flush), the journal commit finishes more quickly than in EXT4 (full flush) since the SSD returns the flush command without persisting the data blocks. In EXT4 (no flush), the journal commit finishes more quickly than EXT (quick flush) since it does not issue the flush command. In journaling throughput, BarrierFS prevails the remainders by far since the interval between the consecutive journal commits is as small as the dispatch latency, t_D.

The concurrencies in journaling in EXT4 (no flush) and in EXT4 (quick flush) have their price. EXT4 (quick flush) requires the additional hardware component, supercap, in the SSD. EXT4 (quick flush) guarantees neither durability or ordering in the journal commit. BarrierFS commits multiple transactions concurrently and yet can guarantee the durability of the individual journal commit without the assistance of additional hardware.

The barrier enabled IO stack does not require any major changes in the existing in-memory or on-disk structure of the IO stack. The only new data structure we introduce is the "conflict-page-list" for a running transaction. Barrier enabled IO stack consists of approximately 3K LOC changes in the IO stack of the Linux kernel .

4.6 Comparison with OptFS

As the closest approach of our sort, OptFS deserves an elaboration. OptFS and barrier-enabled IO stack differ mainly in three aspects; the target storage media, the technology domain, and the programming model. First, OptFS is not designed for the Flash storage but the barrier-enabled IO stack is. OptFS is designed to reduce the disk seek overhead in a filesystem journaling; via committing multiple transactions together (delayed commit) and via making the disk access sequential (selective data mode journaling). Second, OptFS is the filesystem technique while the barrier enabled IO stack deals with the entire IO stack; the storage device, the block device layer and the filesystem. OptFS is built upon the

legacy block device layer. It suffers from the same overhead as the existing filesystems do. OptFS uses Wait-on-Transfer to control the transfer order between *D* and *JD*. OptFS relies on Transfer-and-Flush to control the storage order between the journal commit and the associated checkpoint in `osync()`. Barrier-enabled IO stack eliminates the overhead of Wait-on-Transfer and Transfer-and-Flush in controlling the storage order. Third, OptFS focuses on revising the filesystem journaling model. BarrierFS is not limited to revising the filesystem journaling model but also exports generic storage barrier with which the application can group a set of writes into an epoch.

5 Applications

To date, `fdatasync()` has been the sole resort to enforce the storage order between the write requests. The virtual disk managers for VM disk image, e.g., *qcow2*, use `fdatasync()` to enforce the storage order among the writes to the VM disk image [7]. SQLite uses `fdatasync()` to control the storage order between the undo-log and the journal header and between the updated database node and the commit block [37]. In a single insert transaction, SQLite calls `fdatasync()` four times, three of which are to control the storage order. In these cases, `fdatabarrier()` can be used in place of `fdatasync()`. In some modern applications, e.g. mail server [62] or OLTP, `fsync()` accounts for the dominant fraction of IO. In TPC-C workload, 90% of IOs are created by `fsync()` [51]. With improved `fsync()` of BarrierFS, the performance of the application can increase significantly. Some applications prefer to trade the durability and the freshness of the result for the performance and scalability of the operation [12, 17]. One can replace all `fsync()` and `fdatasync()` with ordering guarantee counterparts, `fbarrier()` and `fdatabarrier()`, respectively, in these applications.

6 Experiment

We implement a barrier-enabled IO stack on three different platforms, enterprise server (12 cores, Linux 3.10.61), PC server (4 cores, Linux 3.10.61) and smartphone (Galaxy S6, Android 5.0.2, Linux 3.10). We test three storage devices: 843TN (SATA 3.0, QD[2]=32, 8 channels, supercap), 850PRO (SATA 3.0, QD=32, 8 channels), and mobile storage (UFS 2.0, QD=16, single channel). We compare the BarrierFS against EXT4 and OptFS [9]. We refer to each of these as supercap-SSD, plain-SSD, and UFS, respectively. We implement barrier write command in UFS device. In plain-SSD and supercap SSD, we assume that the performance overhead of barrier write is 5% and none, repsectively.

[2]QD: queue depth

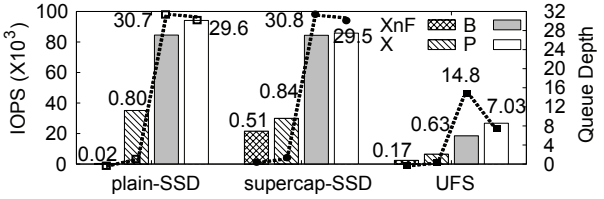
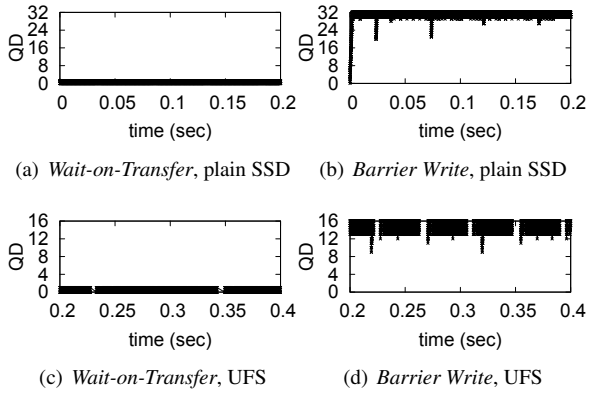


Figure 9: 4KB Randwom Write; XnF: `write()` followed by `fdatasync()`, X: `write()` followed by `fdatasync()`(`no-barrier` option), B: `write()` followed by `fdatabarrier()`, P: orderless `write()`



(a) *Wait-on-Transfer*, plain SSD (b) *Barrier Write*, plain SSD

(c) *Wait-on-Transfer*, UFS (d) *Barrier Write*, UFS

Figure 10: Queue Depth, 4KB Random Write

6.1 Order-Preserving Block Layer

We examine the performance of 4 KByte random write with different ways of enforcing the storage order: P (orderless write [i.e. plain buffered write]), B (barrier write), X (Wait-on-Transfer) and XnF (Transfer-and-Flush). Fig. 9 illustrates the result.

The overhead of Transfer-and-Flush is severe. With Transfer-and-Flush, the IO performances of the ordered write are 0.5% and 10% of orderless write in plain-SSD and UFS, respectively. In supercap SSD, the performance overhead is less significant, but is still considerable; the performance of the ordered write is 35% of the orderless write in UFS. The overhead of DMA transfer is significant. When we interleave the write requests with DMA transfer, the IO performance is less than 40% of the orderless write in each of the three storage devices.

The overhead of barrier write is negligible. When using a barrier write, the ordered write exhibits 90% performance of the orderless write in plain-SSD and super-cap SSD. For UFS, it exhibits 80% performance of the orderless write. The barrier write drives the queue to its maximum in all three Flash storages. The storage performance is closely related to the command queue utilization [33]. In Wait-on-Transfer, the queue depth never goes beyond one (Fig. 10(a) and Fig. 10(c)). In barrier write, the queue depth grows near to its maximum in all storage devices (Fig. 10(b) and Fig. 10(d)).

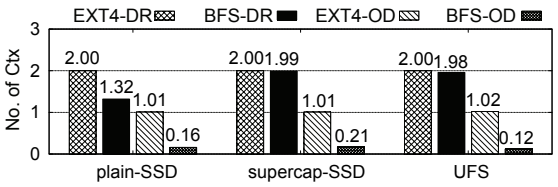
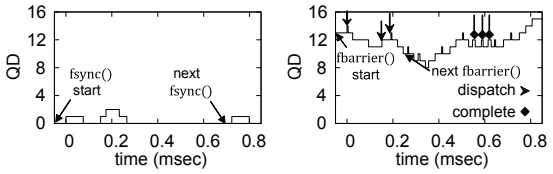


Figure 11: Average Number of Context Switches, EXT4-DR: `fsync()`, BFS-DR: `fsync()`, EXT-OD: `fsync()` with `no-barrier`, BFS-OD: `fbarrier()`, 'DR' = durability guarantee, 'OD' = ordering guarantee, 'EXT4-OD' guarantees only the transfer order, but not storage order.



(a) Durability Guarantee (b) Ordering Guarantee

Figure 12: Queue Depth in BarrierFS: `fsync()` and `fbarrier()`

6.2 Filesystem Journaling

We examine the latency, the number of context switches and the queue depth in filesystem journaling in EXT4 and BarrierFS. We use Mobibench [26]. For latency, we perform 4 KByte allocating `write()` followed by `fsync()`. With this, an `fsync()` always finds the updated metadata to journal and the `fsync()` latency properly represents the time to commit a journal transaction. For context switch and queue depth, we use 4 KByte non-allocating random write followed by different synchronization primitives.

Latency: In plain-SSD and supercap-SSD, the average `fsync()` latency decreases by 40% when we use BarrierFS against when we use EXT4 (Table 2). In UFS, the `fsync()` latency decreases by 60% in BarrierFS compared against EXT4. UFS experiences more significant reduction in `fsync()` latency than the other SSDs do.

BarrierFS makes the `fsync()` latency less variable. In supercap-SSD and UFS, the `fsync()` latencies at the 99.99^{th} percentile are 30× of the average `fsync()` latency (Table 2). In BarrierFS, the tail latencies at 99.99^{th} percentile decrease by 50%, 20%, and 70% in UFS, plain-SSD, and supercap-SSD, respectively, against EXT4.

(%)	UFS		plain-SSD		supercap-SSD	
	EXT4	BFS	EXT4	BFS	EXT4	BFS
μ	1.29	0.51	5.95	3.52	0.15	0.09
Median	1.20	0.44	5.43	3.01	0.15	0.09
99^{th}	4.15	3.51	11.41	8.96	0.16	0.10
99.9^{th}	22.83	9.02	16.09	9.30	0.28	0.24
99.99^{th}	33.10	17.60	17.26	14.19	4.14	1.35

Table 1: `fsync()` latency statistics (msec)

Context Switches: We examine the number of application level context switches in different journaling modes (Fig. 11). In EXT4, `fsync()` wakes up the caller twice: after *D* is transferred and after the journal transaction is made durable(EXT4-DR). This applies to all three storages. In BarrierFS, the number of context switches in an `fsync()` varies subject to the storage device. In UFS and supercap SSD, `fsync()` of BarrierFS wakes up the caller twice, as in the case of `fsync()` of EXT4. However, the reasons are entirely different. In UFS and supercap-SSD, the intervals between the successive write requests are much smaller than the timer interrupt interval due to small flush latency. A `write()` request rarely finds the updated metadata and an `fsync()` often resorts to an `fdatasync()`. `fdatasync()` wakes up the caller (the application thread) twice in BarrierFS: after transferring *D* and after flush completes. In plain SSD, `fsync()` of BarrierFS wakes up the caller once: after the transaction is made durable. The plain-SSD uses TLC Flash. The interval between the successive `write()`s is longer than the timer interrupt interval. The application thread blocks after triggering the journal commit and and wakes up after the journal commit operation completes.

BFS-OD manifests the benefits of BarrierFS. The `fbarrier()` rarely finds updated metadata since it returns quickly and as a result, most `fbarrier()` calls are serviced as `fdatabarrier()`. `fdatabarrier()` does not block the caller and therefore does not accompany any involuntary context switch.

Command Queue Depth: In BarrierFS, the host dispatches the write requests for *D*, *JD*, and *JC* in tandem. Ideally, there can be as many as three commands in the queue. We observe only up to two commands in the queue in servicing an `fsync()` (Fig. 12(a)). This is due to the context switch between the application thread and the commit thread. Writing *D* and writing *JD* are 160 μsec apart, but it takes 70μsec to service the write request for *D*. In `fbarrier()`, BarrierFS successfully drives the command queue to its full capacity (Fig. 12(b)).

Throughput and Scalability: The filesystem journaling is a main obstacle against building an manycore scalable system [44]. We examine the throughput of filesystem journaling in EXT4 and BarrierFS with a varying number of CPU cores in a 12 core machine. We use modified DWSL workload in `fxmark` [45]; each thread performs a 4-Kbyte allocating write followed by `fsync()`. Each thread operates on its own file. BarrierFS exhibits much more scalable behavior than EXT4 (Fig. 13). In plain-SSD, BarrierFS exhibits 2× performance against EXT4 in all numbers of cores (Fig. 13(a)). In supercap-

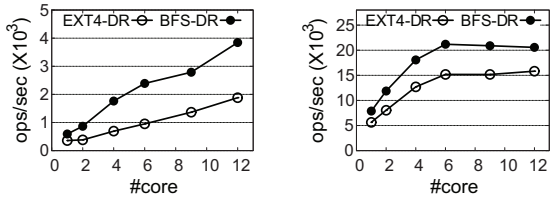
SSD, the performance saturates with six cores in both EXT4 and BarrierFS. BarrierFS exhibits 1.3× journaling throughput against EXT4 (Fig. 13(b)).

6.3 Server Workload

We run two workloads: `varmail` [71] and OLTP-insert [34]. OLTP-insert workload uses MySQL DBMS [47]. `varmail` is a metadata-intensive workload. It is known for the heavy `fsync()` traffic. There are total four combinations of the workload and the SSD (plain-SSD and supercap-SSD) pair. For each combination, we examine the benchmark performances for durability guarantee and ordering guarantee, respectively. For durability guarantee, we leave the application intact and use two filesystems, the EXT4 and the BarrierFS (EXT4-DR and BFS-DR). The objective of this experiment is to examine the efficiency of `fsync()` implementations in EXT4 and BarrierFS, respectively. For ordering guarantee, we test three filesystems, OptFS, EXT4 and BarrierFS. In OptFS and BarrierFS, we use `osync()` and `fdatabarrier()` in place of `fsync()`, respectively. In EXT4, we use `nobarrier` mount option. This experiment examines the benefit of Wait-on-Dispatch. Fig. 14 illustrates the result.

Let us examine the performances of `varmail` workload. In plain-SSD, BFS-DR brings 60% performance gain against EXT4-DR in `varmail` workload. In supercap-SSD, BFS-DR brings 10% performance gain against EXT4-DR. The experimental result of supercap-SSD case clearly shows the importance of eliminating the Wait-on-Transfer overhead in controlling the storage order. The benefit of BarrierFS manifests itself when we relax the durability guarantee. In ordering guarantee, BarrierFS achieves 80% performance gain against EXT4-OD. Compared to the baseline, EXT4-DR, BarrierFS achieves 36× performance (1.0 vs. 35.6 IOPS) when we enforce only ordering guarantee with BarrierFS (BFS-OD) in plain SSD .

In MySQL, BFS-OD prevails EXT4-OD by 12%. Compared to the baseline, EXT4-DR, BarrierFS achieves 43× performance (1.3 vs. 56.0 IOPS) when we enforce only ordering guarantee with BarrierFS (BFS-OD) in plain SSD.



(a) plain-SSD (b) supercap-SSD

Figure 13: fxmark: scalability of filesystem journaling

6.4 Mobile Workload: SQLite

We examine the performances of the libarary based embedded DBMS, SQLite, under the durability guarantee and the ordering guarantee, respectively. We examine two journal modes, PERSIST and WAL. We use 'Full Sync' and the WAL file size is set to 1,000 pages, both of which are default settings [58]. In a single `insert` transaction, SQLite calls `fdatasync()` four times. Three of them are to control the storage order and the last one is for making the result of a transaction durable.

For durability guarantee mode, We replace the first three `fdatasync()`'s with `fdatabarrier()`'s and leave the last one. In mobile storage, BarrierFS achieves 75% performance improvement against EXT4 in default PERSIST journal mode under durability guarantee (Fig. 15). In ordering guarantee, we replace all four `fdatasync()`'s with `fdatabarrier()`'s. In UFS, SQLite exhibits 2.8× performance gain in BFS-OD against EXT4-DR. The benefit of eliminating the Transfer-and-Flush becomes more dramatic as the storage controller employs higher degree of parallelism. In plain-SSD, SQLite exhibits 73× performance gain in BFS-OD against EXT4-DR (73 vs. 5300 ins/sec).

Notes on OptFS: OptFS does not perform well in our experiment (Fig. 14 and Fig. 15), unlike that in [9]. We find two reasons. First, the benefit of delayed checkpoint and selective data mode journaling becomes marginal in Flash storage. Second, in Flash storage (i.e. the storage with short IO latency) the delayed checkpoint and the selective data mode journaling negatively interact with each other and bring substantial increase in the memory pressure. The increased memory pressure severely impacts the performance of `osync()`. The `osync()` scans all dirty pages for the checkpoint at its beginning. Selective data mode journaling inserts the updated data blocks to the journal transaction. Delayed checkpoint prohibits the data blocks in the journal transaction from being checkpointed until the associated ADN arrives. As a result, `osync()` checkpoints only a small fraction of dirty pages each time it is called. The dirty pages in the journal transactions are scanned multiple times before they are checkpointed. The `osync()` shows particularly poor performance in OLTP workload (Fig. 14), where most

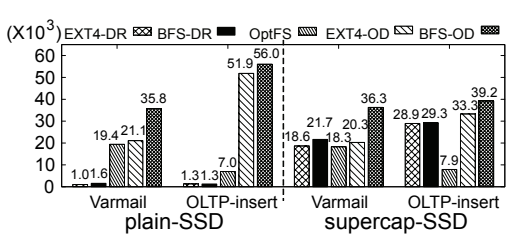
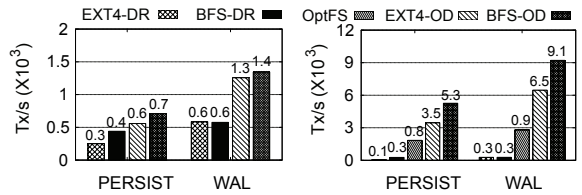


Figure 14: `varmail` (ops/s) and OLTP-insert (Tx/s)



(a) UFS (b) plain-SSD

Figure 15: SQLite Performance: ins/sec, 100K inserts

updates are subject to data mode journaling.

6.5 Crash Consistency

We test if the BarrierFS recovers correctly against the unexpected system failure. We use `CrashMonkey` for the test [40]. We modify CrashMonkey to understand the barrier write so that the CrashMonkey can properly delimit an epoch when it encounters a barrier write. We run two workloads; `rename_root_to_sub` and `create_delete`. For durability guarantee (`fsync()`), BarrierFS passes all 1,000 test cases as EXT4 does in both workloads. For ordering guarantee (`fsync()` in EXT4-OD and `fbarrier()` in BarrierFS), BarrierFS passes all 1,000 test cases whereas EXT4-OD fails in some cases. This is not surprising since EXT4 with `nobarrier` option guarantees neither the transfer orders nor the persist orders in committing the filesystem journal transaction.

Scenario	-	EXT4-DR	BFS-DR	EXT4-OD	BFS-OD
A	clean	1000	1000	547	1000
	fixed	0	0	0	0
	failed	0	0	453	0
B	clean	1000	1000	109	1000
	fixed	0	0	891	0
	failed	0	0	0	0

Table 2: Crash Consistency Test of EXT4 and BarrierFS, Scenario A: `rename_root_to_sub`, Scenario B: `create_delete`

7 Related Work

Featherstitch [20] proposes a programming model to specify the set of requests that can be scheduled together, `patchgroup`, and the ordering dependency between them, `pg_depend()`. While `xsyncfs` [49] mitigates the overhead of `fsync()`, it needs to maintain complex causal dependencies among buffered updates. `NoFS` (no order file system) [10] introduces "backpointer" to eliminate the Transfer-and-Flush based ordering in the file system. It does not support transaction.

A few works proposed to use multiple running transactions or multiple committing transactions to circumvent the Transfer-and-Flush overhead in filesystem journaling [38, 29, 55]. IceFS [38] allocates separate running transaction for each container. SpanFS [29] splits a journal region into multiple partitions and allocates committing transactions for each partition. CCFS [55] allocates separate running transactions for individual threads. In these systems, each journaling session still relies on the Transfer-and-Flush mechanism.

A number of file systems provide a multi-block atomic write feature [18, 35, 54, 68] to relieve applications from the overhead of logging and journaling. These file systems internally use the Transfer-and-Flush mechanism to enforce the storage order in writing the data blocks and the associated metadata blocks. Exploiting the order-preserving block device layer, these filesystems can use Wait-on-Dispatch mechanism to enforce the storage order between the data blocks and the metadata blocks and can be saved from the Transfer-and-Flush overhead.

8 Conclusion

The Flash storage provides the cache barrier command to allow the host to control the persist order. HDD cannot provide this feature. It is time for designing the new IO stack for the Flash storage that is free from the unnecessary constraint inherited from the old legacy that the host cannot control the persist order. We built a barrier-enabled IO stack based upon the foundation that the host can control the persist order. In the barrier-enabled IO stack, the host can dispense with *Transfer-and-Flush* overhead in controlling the storage order and can successfully saturate the underlying Flash storage. We like to conclude this work with two key observations. First, the "cache barrier" command is a necessity rather than a luxury. It should be supported in all Flash storage products ranging from the mobile storage to the high-performance Flash storage with supercap. Second, the block device layer should be designed to eliminate the DMA transfer overhead in controlling the storage order. As the Flash storage becomes quicker, the relative cost of tardy "Wait-on-Transfer" will become more substantial. To saturate the Flash storage, the host should be able to control the transfer order without interleaving the requests with DMA transfer.

We hope that this work provides a useful foundation in designing a new IO stack for the Flash storage[3].

9 Acknowledgement

We would like to thank our shepherd Vijay Chidambaram and the anonymous reviewers for their valuable feedback. We also would like to thank Jayashree Mohan for her help in CrashMonkey. This work is funded by Basic Research Lab Program (NRF, No. 2017R1A4A1015498), the BK21 plus (NRF), ICT R&D program (IITP, R7117-16-0232) and Samsung Elec.

[3]The source code for barrier enabled IO stack is available at `https://github.com/ESOS-Lab/barrieriostack`.

References

[1] emmc5.1 solution in sk hynix. https://www.skhynix.com/kor/product/nandEMMC.jsp.

[2] Toshiba expands line-up of e-mmc version 5.1 compliant embedded nand flash memory modules. http://toshiba.semicon-storage.com/us/company/taec/news/2015/03/memory-20150323-1.html.

[3] AXBOE, J. Linux block IO present and future. In *Proc. of Ottawa Linux Symposium* (Ottawa, Ontario, Canada, Jul 2004).

[4] BEST, S. JFS Overview. http://jfs.sourceforge.net/project/pub/jfs.pdf, 2000.

[5] CHANG, Y.-M., CHANG, Y.-H., KUO, T.-W., LI, Y.-C., AND LI, H.-P. Achieving SLC Performance with MLC Flash Memory. In *Proc. of DAC 2015* (San Francisco, CA, USA, 2015).

[6] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proc. of IEEE HPCA 2011* (San Antonio, TX, USA, Feb 2011).

[7] CHEN, Q., LIANG, L., XIA, Y., CHEN, H., AND KIM, H. Mitigating sync amplification for copy-on-write virtual disk. In *Proc. of USENIX FAST 2016* (Santa Clara, CA, 2016), pp. 241–247.

[8] CHIDAMBARAM, V. *Orderless and Eventually Durable File Systems*. PhD thesis, UNIVIRSITY OF WISCONSIN–MADISON, 2015.

[9] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic Crash Consistency. In *Proc. of ACM SOSP 2013* (Farmington, PA, USA, Nov 2013). https://github.com/utsaslab/optfs.

[10] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency Without Ordering. In *Proc. of USENIX FAST 2012* (San Jose, CA, USA, Feb 2012).

[11] CHO, Y. S., PARK, I. H., YOON, S. Y., LEE, N. H., JOO, S. H., SONG, K.-W., CHOI, K., HAN, J.-M., KYUNG, K. H., AND JUN, Y.-H. Adaptive multi-pulse program scheme based on tunneling speed classification for next generation multi-bit/cell NAND flash. *IEEE Journal of Solid-State Circuits(JSSC) 48*, 4 (2013), 948–959.

[12] CIPAR, J., GANGER, G., KEETON, K., MORREY III, C. B., SOULES, C. A., AND VEITCH, A. LazyBase: trading freshness for performance in a scalable database. In *Proc. of ACM EuroSys 2012* (Bern, Switzerland, Apr 2012).

[13] COBB, D., AND HUFFMAN, A. NVM express and the PCI express SSD Revolution. In *Proc. of Intel Developer Forum* (San Francisco, CA, USA, 2012).

[14] 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 *Proc. of ACM SOSP 2009* (Big Sky, MT, USA, Oct 2009).

[15] CORBET, J. Barriers and journaling filesystems. http://lwn.net/Articles/283161/, August 2010.

[16] CORBET, J. The end of block barriers. https://lwn.net/Articles/400541/, August 2010.

[17] CUI, H., CIPAR, J., HO, Q., KIM, J. K., LEE, S., KUMAR, A., WEI, J., DAI, W., GANGER, G. R., GIBBONS, P. B., ET AL. Exploiting bounded staleness to speed up big data analytics. In *Proc. of USENIX ATC 2014* (Philadelihia, PA, USA, Jun 2014).

[18] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area Cooperative Storage with CFS. In *Proc. of ACM SOSP 2001* (Banff, Canada, Oct 2001).

[19] DEES, B. Native command queuing-advanced performance in desktop storage. *IEEE Potentials Magazine 24*, 4 (2005), 4–7.

[20] FROST, C., MAMMARELLA, M., KOHLER, E., DE LOS REYES, A., HOVSEPIAN, S., MATSUOKA, A., AND ZHANG, L. Generalized File System Dependencies. In *Proc. of ACM SOSP 2007* (Stevenson, WA, USA, Oct 2007).

[21] GIM, J., AND WON, Y. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. *ACM Transactions on Storage (TOS) 6*, 2 (2010).

[22] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of nand flash memory. In *Proc.of USENIX FAST 2012* (Berkeley, CA, USA, 2012).

[23] GUO, J., YANG, J., ZHANG, Y., AND CHEN, Y. Low cost power failure protection for mlc nand flash storage systems with pram/dram hybrid buffer. In *Proc. of DATE 2013* (Alpexpo Grenoble, France, 2013), pp. 859–864.

[24] HELLWIG, C. Patchwork block: update documentation for req_flush / req_fua. https://patchwork.kernel.org/patch/134161/.

[25] HELM, M., PARK, J.-K., GHALAM, A., GUO, J., WAN HA, C., HU, C., KIM, H., KAVALIPURAPU, K., LEE, E., MOHAMMADZADEH, A., ET AL. 19.1 A 128Gb MLC NAND-Flash device using 16nm planar cell. In *Proc. of IEEE ISSCC 2014* (San Francisco, CA, USA, Feb 2014).

[26] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O Stack Optimization for Smartphones. In *Proc. of USENIX ATC 2013* (San Jose, CA, USA, Jun 2013).

[27] JESD220C, J. S. Universal Flash Storage(UFS) Version 2.1.

[28] JESD84-B51, J. S. Embedded Multi-Media Card(eMMC) Electrical Standard (5.1).

[29] KANG, J., ZHANG, B., WO, T., YU, W., DU, L., MA, S., AND HUAI, J. SpanFS: A Scalable File System on Fast Storage Devices. In *Proc. of USENIX ATC 2015* (Santa Clara, CA, USA, Jul 2015).

[30] KANG, W.-H., LEE, S.-W., MOON, B., OH, G.-H., AND MIN, C. X-FTL: Transactional FTL for SQLite Databases. In *Proc. of ACM SIGMOD 2013* (New York, NY, USA, Jun 2013).

[31] KESAVAN, R., SINGH, R., GRUSECKI, T., AND PATEL, Y. Algorithms and data structures for efficient free space reclamation in wafl. In *Proc. of USENIX FAST 2017* (Santa Clara, CA, 2017), USENIX Association, pp. 1–14.

[32] KIM, H.-J., AND KIM, J.-S. Tuning the ext4 filesystem performance for android-based smartphones. In *Proc. of ICFCE 2011* (2011), S. Sambath and E. Zhu, Eds., vol. 133 of *Advances in Intelligent and Soft Computing*, Springer, pp. 745–752.

[33] KIM, Y. An empirical study of redundant array of independent solid-state drives (RAIS). *Springer Cluster Computing 18*, 2 (2015), 963–977.

[34] KOPYTOV, A. SysBench manual. `http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf`, 2004.

[35] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A New File System for Flash Storage. In *Proc. of USENIX FAST 2015* (Santa Clara, CA, USA, Feb 2015).

[36] LEE, S., LEE, J.-Y., PARK, I.-H., PARK, J., YUN, S.-W., KIM, M.-S., LEE, J.-H., KIM, M., LEE, K., KIM, T., ET AL. 7.5 A 128Gb 2b/cell NAND flash memory in 14nm technology with tPROG=640us and 800MB/s I/O rate. In *Proc. of IEEE ISSCC 2016* (San Francisco, CA, USA, Feb 2016).

[37] LEE, W., LEE, K., SON, H., KIM, W.-H., NAM, B., AND WON, Y. WALDIO: eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proc. of USENIX ATC 2015* (Santa Clara, CA, USA, Jul 2015).

[38] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical Disentanglement in a Container-Based File System. In *Proc. of USENIX OSDI 2014* (Broomfield, CO, USA, Oct 2014).

[39] LU, Y., SHU, J., GUO, J., LI, S., AND MUTLU, O. Lighttx: A lightweight transactional design in flash-based ssds to support flexible transactions. In *Proc. of IEEE ICCD 2013*.

[40] MARTINEZ, A., AND CHIDAMBARAM, V. Crashmonkey: A framework to automatically test file-system crash consistency. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (Santa Clara, CA, 2017). `https://github.com/utsaslab/crashmonkey`.

[41] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proc. of Linux symposium 2007* (Ottawa, Ontario, Canada, Jun 2007).

[42] MCKUSICK, M. K., GANGER, G. R., ET AL. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proc. of USENIX ATC 1999* (Monterey, CA, USA, Jun 1999).

[43] MIN, C., KANG, W.-H., KIM, T., LEE, S.-W., AND EOM, Y. I. Lightweight application-level crash consistency on transactional flash storage. In *Proc. of USENIX ATC 2015* (Santa Clara, CA, USA, Jul 2015).

[44] MIN, C., KASHYAP, S., MAASS, S., AND KIM, T. Understanding Manycore Scalability of File Systems. In *Proc. of USENIX ATC 2016* (Denver, CO, USA, Jun 2016).

[45] MIN, C., KASHYAP, S., MAASS, S., AND KIM, T. Understanding manycore scalability of file systems. In *Proc.of USENIX ATC 2016* (Denver, CO, 2016), pp. 71–85.

[46] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems(TODS) 17*, 1 (1992), 94–162.

[47] MYSQL, A. Mysql 5.1 reference manual. *Sun Microsystems* (2007).

[48] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write Off-loading: Practical Power Management for Enterprise Storage. *ACM Transactions on Storage(TOS) 4*, 3 (2008), 10:1–10:23.

[49] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the Sync. In *Proc. of USENIX OSDI 2006* (Seattle, WA, USA, Nov 2006).

[50] OKUN, M., AND BARAK, A. Atomic writes for data integrity and consistency in shared storage devices for clusters. In *Proc. of ICA3PP 2002* (Beijing, China, Oct 2002).

[51] OU, J., SHU, J., AND LU, Y. A high performance file system for non-volatile main memory. In *Proc. of ACM EuroSys 2016* (London, UK, Apr 2016).

[52] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND PANDA, D. K. Beyond block I/O: Rethinking traditional storage primitives. In *Proc. of IEEE HPCA 2011* (San Antonio, TX, USA, Feb 2011).

[53] PALANCA, S., FISCHER, S. A., MAIYURAN, S., AND QAWAMI, S. Mfence and lfence micro-architectural implementation method and system, July 5 2016. US Patent 9,383,998.

[54] PARK, S., KELLY, T., AND SHEN, K. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proc. of ACM EuroSys 2013* (Prague, Czech Republic, Apr 2013).

[55] PILLAI, T. S., ALAGAPPAN, R., LU, L., CHIDAMBARAM, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Application crash consistency and performance with ccfs. In *Proc.of USENIX FAST 2017* (Santa Clara, CA, 2017), pp. 181–196.

[56] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON File Systems. In *Proc. of ACM SOSP 2005* (Brighton, UK, Oct 2005).

[57] PRABHAKARAN, V., RODEHEFFER, T. L., AND ZHOU, L. Transactional flash. In *Proc. of USENIX OSDI 2008* (Berkeley, CA, USA, 2008), pp. 147–160.

[58] PUROHITH, D., MOHAN, J., AND CHIDAMBARAM, V. The dangers and complexities of sqlite benchmarking. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (New York, NY, USA, 2017), APSys '17, ACM, pp. 3:1–3:6.

[59] REV, H. SCSI Commands Reference Manual. `http://www.seagate.com/files/staticfiles/support/docs/manual/Interface%20manuals/100293068h.pdf/`, Jul 2014. Seagate.

[60] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013).

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

[62] SEHGAL, P., TARASOV, V., AND ZADOK, E. Evaluating Performance and Energy in File System Server Workloads. In *Proc. of USENIX FAST 2010* (San Jose, CA, USA, Feb 2010).

[63] SELTZER, M. I., GANGER, G. R., MCKUSICK, M. K., SMITH, K. A., SOULES, C. A., AND STEIN, C. A. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proc. of USENIX ATC 2000* (San Diego, CA, USA, Jun 2000).

[64] SHILAMKAR, G. Journal Checksums. `http://wiki.old.lustre.org/images/4/44/Journal-\checksums.pdf`, May 2007.

[65] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the xfs file system. In *Proc. of USENIX ATC 1996* (Berkeley, CA, USA, 1996).

[66] TS'O, T. Using Cache barrier in liue of `REQ_FLUSH`. `http://www.spinics.net/lists/linux-ext4/msg49018.html`, September 2015.

[67] TWEEDIE, S. C. Journaling the linux ext2fs filesystem. In *Proc.of The Fourth Annual Linux Expo* (Durham, NC, USA, May 1998).

[68] VERMA, R., MENDEZ, A. A., PARK, S., MANNARSWAMY, S., KELLY, T., AND MORREY, C. Failure-Atomic Updates of Application Data in a Linux File System. In *Proc. of USENIX FAST 2015* (Santa Clara, CA, USA, Feb 2015).

[69] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 357–370.

[70] WEISS, Z., SUBRAMANIAN, S., SUNDARARAMAN, S., TALAGALA, N., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *Proc. of USENIX FAST 2015* (Santa Clara, CA, USA, Feb 2015).

[71] WILSON, A. The new and improved FileBench. In *Proc. of USENIX FAST 2008* (San Jose, CA, USA, Feb 2008).

[72] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proc. of ACM SYSTOR 2015* (Haifa, Israel, May 2015).

[73] Y. PARK, S., SEO, E., SHIN, J. Y., MAENG, S., AND LEE, J. Exploiting Internal Parallelism of Flash-based SSDs. *IEEE Computer Architecture Letters(CAL) 9*, 1 (2010), 9–12.

[74] ZHANG, C., WANG, Y., WANG, T., CHEN, R., LIU, D., AND SHAO, Z. Deterministic crash recovery for NAND flash based storage systems. In *Proc. of ACM/EDAC/IEEE DAC 2014* (San Francisco, CA, USA, Jun 2014).

High-Performance Transaction Processing in Journaling File Systems

Yongseok Son, Sunggon Kim, Heon Young Yeom, and Hyuck Han†
Seoul National University, †Dongduk Women's University

Abstract

Journaling file systems provide crash-consistency to applications by keeping track of uncommitted changes in the journal area (journaling) and writing committed changes to their original area at a certain point (checkpointing). They generally use coarse-grained locking to access shared data structures and perform I/O operations by a single thread. For these reasons, journaling file systems often face the problem of lock contention and underutilization of I/O bandwidth on multi-cores with high-performance storage. To address these issues, we design journaling and checkpointing schemes that enable concurrent updates on data structures and parallelize I/O operations. We implement our schemes in EXT4/JBD2 and evaluate them on a 72-core machine with a high-performance NVMe SSD. The experimental results show that our optimized file system improves the performance by up to about 2.2x and 1.5x compared to the existing EXT4 file system and a recent scalable file system, respectively.

1 Introduction

A transaction in file systems is a group of file system modifications that must be atomic and durable [6, 12, 23]. To support transaction processing, many file systems have adopted a journaling technique to guarantee the atomicity and durability. Journaling logs modified metadata and data to the journal area in a transaction before updating the original area. After the transaction is committed, the committed transaction is written into the original area by checkpointing (write-ahead logging). With journaling, the file systems can provide crash-consistency to applications by recovering the committed transactions in case of crashes [5, 23, 24].

Although crash-consistency is supported using journaling file systems [20, 26, 29, 31], journaling can face a performance bottleneck on multi-cores and high-performance storage [15, 16, 21]. The performance bottleneck mainly arises from (1) data structures for transaction processing protected by non-scalable locks and (2) serialized I/O operations by a single thread. For example, in EXT4/JBD2, multiple application threads insert their own buffer into the running transaction by using coarse-grained locking which can negatively affect scalability on multi-cores. In addition, a single application thread performs checkpoint I/O operations which can underutilize high-performance storage.

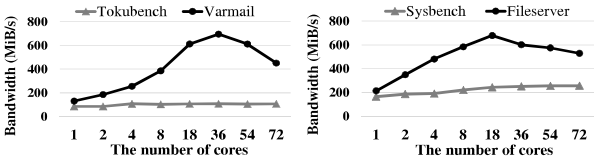
To handle these issues, previous studies [16, 21] investigated the locking and I/O operations of file systems. SpanFS [16] consists of a collection of micro file system services called domains. It distributes files and directories among the domains and delegates an I/O operation to the corresponding domain to reduce the lock contention and exploit the device parallelism. Min et al. [21] observed that file systems are hidden scalability bottlenecks in many I/O-intensive applications. They designed and implemented a benchmark to evaluate the scalability of file systems and found unexpected scalability behaviors. Our study is in line with these approaches [16, 21] in terms of investigating the locking and I/O operations of file systems. In contrast, we focus on internal operations of shared data structures and I/O processing in transaction processing.

In this paper, we propose a transaction processing with two main schemes to achieve high-performance I/O as follows: (1) We use lock-free data structures and operations to reduce the lock contention. This scheme allows multiple threads to access the data structures (e.g., linked lists) concurrently. (2) We propose a parallel I/O scheme that performs I/O operations by multiple threads in a parallel and cooperative manner. This scheme allows multiple threads to cooperate in I/O processing and issue/complete the I/Os in parallel while not sacrificing the consistency of the file system. We apply and implement the techniques to transaction processing (i.e., running, committing, checkpointing, and recovery) on EXT4/JBD2 in Linux kernel 4.9.1.

We evaluate the existing and our optimized file systems on a 72-core machine with an Intel P3700 NVMe SSD [14] using metadata and data-intensive workloads in the ordered and data journaling modes. The experimental results show that the optimized file system improves the performance by up to about 2.2x and 2.1x in the ordered mode and the data journaling mode, respectively, compared to the existing file system. Also, the optimized file system improves the performance by up to about 1.5x compared to SpanFS, a recently developed file system for multi-core scalability.

The contributions of our work are as follows:

- We analyze the locking and I/O operations in transaction processing of EXT4/JBD2 in terms of its procedures.
- We design and implement a transaction processing with concurrent lock-free updates on data structures



(a) Ordered mode (b) Data journaling mode

Figure 1: Motivational evaluation (the number of threads is the same as that of the cores, and the detailed experimental environment is described in Section 4.)

and parallel I/O processing in a cooperative manner.

- We demonstrate that our optimized file system improves the performance compared to the existing file system and a recent scalable file system.

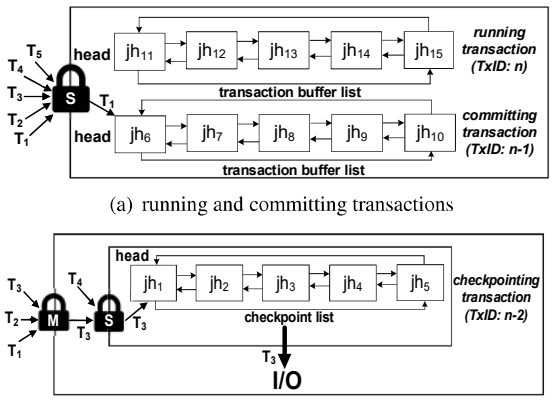
The rest of this paper is organized as follows: Section 2 describes the background and motivation. Section 3 presents the design and implementation of the proposed schemes. Section 4 shows the experimental results. Section 5 discusses the related works. Section 6 concludes this paper.

2 Background and Motivation

This paper focuses on the EXT4 journaling mechanism since EXT4 is the most widely used file system in Linux and more general than other file systems [16, 17]. EXT4 uses a fork of the journaling block device (JBD) called JBD2 [32] which performs transaction processing by using a variant of write-ahead logging (WAL) [11]. JBD is a file system-independent interface that can also be attached to other file systems, such as EXT3 and OCFS2 [10]. EXT4/JBD2 provides three journaling modes: write-back, ordered, and data journaling. The detailed description of each journaling mode can be found in previous studies [2, 16, 23, 25].

JBD2 adopts a single compound transaction. There is only one running transaction that absorbs all updates and one committing transaction at any time. An application thread starts a running transaction for each update and associates the update with the transaction. The running transaction has a linked list, which has the pointers to the modified blocks. When a periodic commit operation is invoked or an `fsync()` is called, a journal thread changes the state of the transaction to *committing*, and writes the blocks associated with the transaction into the journal area. After the transaction commits, the transaction is marked as checkpoint. The committed blocks are written back to the original area by an application thread during the checkpoint operation. Then, the journal area is reclaimed via the checkpoint operation.

We focus on the locking and I/O operations in transaction processing. Figure 1 shows a motivational evaluation using metadata and data-intensive workloads in the ordered and data journaling modes, respectively. As



(a) running and committing transactions

(b) checkpointing transaction

Figure 2: Examples of existing locking and I/O operations (T: thread, TxID: transaction ID, jh: journal_head, S: spin lock (`j_list_lock`), M: mutex lock (`j_checkpoint_mutex`))

shown in the figure, the performance does not scale well or decreases as the number of cores grows. Based on our analysis and other studies [15, 16], it is due to the lock contention on shared data structures and the serialization of I/O operations. For example, as shown in Figure 2(a) and 2(b), a spin lock (`j_list_lock`) is used to ensure the correct list operations for journal heads (`jhs`)[1] in the journaling lists (transaction buffer and checkpoint lists) [16], which are circular doubly linked lists. However, in multi-cores, this locking can incur contention on the shared data structures and limit the scalability. In addition, only a single thread performs the journal and checkpoint I/Os. For example, as shown in Figure 2(b), T_3 performs I/O operations for checkpointing by acquiring a mutex lock (`j_checkpoint_mutex`). Such serialized I/O operations can limit the I/O parallelism on high-performance storage. We will explain the transaction processing in terms of locking and I/O operations.

Running transaction. When application threads perform some file operations (e.g., `create()`), they start a transaction to handle the modifications. To process the transaction, the threads first check if a running transaction is available or not. If a running transaction is available, the threads join the running transaction by increasing the number of updates (`t_updates`) in the transaction under the state lock (`j_state_lock`) which is a read-write lock; the `t_updates` variable indicates the number of current threads that join the transaction. Otherwise, a new transaction is created, or the threads are blocked if the transaction cannot be newly created. When a running transaction needs to be committed while a previous transaction is committing, the threads which try

[1] Journal head (`jh`) is a structure that associates the buffer (buffer_head (`bh`)) with the respective transaction [13]. The operations on the `bh` are protected by a spin lock (`jbd_lock_bh_state`) per `bh`.

to get a running transaction are blocked until the running transaction is available. It is because there are only one running transaction and one committing transaction at any time in the compound transaction scheme [16, 23].

After getting the running transaction, the threads modify their own buffer and then try to insert it into a transaction buffer list by using the `jh` of the buffer (`bh`). To insert the `jh`, the threads try to acquire a list lock (`j_list_lock`) which is a spin lock. A thread, which acquires the list lock, associates the `jh` with the running transaction and inserts the `jh` into the tail of the list. Then, the thread releases the list lock and finishes the insert operation. Finally, the thread completes its own transaction processing by decreasing the number of updates.

When application threads perform some file operations, such as `truncate()`, the threads can invalidate buffers that are already associated with a transaction. In this case, by acquiring the state lock and the list lock, a thread removes the `jh` from the transaction buffer or checkpoint lists and disassociates the `jh` from the running or checkpoint transactions if it is associated with the running or checkpoint transactions, respectively. If the `jh` is associated with a committing transaction, the thread sets the `jh` as *freed*; both the `jh` and its buffer will be freed later during the commit procedure. As discussed above, EXT4/JBD2 ensures correct updates on the transaction state and the transaction buffer list by the state lock and the list lock, respectively.

Committing transaction. To commit a transaction, a journal thread wakes up and processes a commit procedure. The journal thread changes the running transaction to a committing transaction and its state to *committing* by acquiring the state lock. Then, the journal thread waits for other application threads to complete their transaction processing by checking the `t_updates` variable. If the `jh` is already associated with a running transaction, the `jh` must be moved to a committing transaction. Meanwhile, the committing transaction does not accept any new modifications, and the next modification triggers the creation of a new running transaction. With the committing transaction, the journal thread prepares for journal I/Os by creating a wait list, which is used to wait for the completion of I/Os. Then, the journal thread fetches the `jh` from the head (`t_buffers`) of the transaction buffer list and creates a copy of its buffer called frozen buffer (`frozen_bh`) to preserve the contents of the buffer. The journal thread removes the `jh` from the list by updating the head of the list to the next of the head and inserts the `jh` into the shadow list under the list lock. The shadow list (`t_shadow`) includes the frozen buffers.

To perform a batched journal I/O, the journal thread aggregates the frozen buffer by inserting it into a write buffer (`wbuf`) and the wait list. If the number of inserted buffers (`bufs`) is higher than the pre-defined threshold, the journal thread issues I/Os to the journal area by calling `submit_bh()` and prepares for the next I/Os. After issuing all the I/O requests for journaling, the journal thread waits for the completion of I/Os. And then removes the `jh` from the shadow list and inserts it into the forget list under the list lock. The forget list (`t_forget`) includes both the frozen buffers from the shadow list and buffers to be freed. After all the I/Os are completed, the journal thread writes the commit block for the transaction atomicity; if a crash occurs, the file system can replay or discard the transaction according to the existence of the commit block of the transaction. Then, the journal thread makes a checkpoint list with the buffers that are not freed and still dirty in the forget list under the list lock. Finally, the committed transaction is inserted into the tail of a checkpoint transaction list for checkpointing by acquiring the state and list locks.

Checkpointing transaction. When a transaction needs to be checkpointed, application threads try to acquire a checkpoint mutex lock (`j_checkpoint_mutex`) and perform a batched I/O operation. A winner thread, which acquires the mutex lock, performs the checkpoint I/O operations while other threads are blocked until the I/O operations are completed. Then, the thread tries to acquire the list lock to get the transaction and access its checkpoint list. The list lock is used since other threads can access the checkpoint list to remove the `jh`s when they free the buffers of the `jh`s, which do not need to be checkpointed.

Under the mutex and list locks, the winner thread aggregates the buffers by fetching the `jh`s from the checkpoint list and inserting the fetched buffers into a checkpoint buffer (`j_chkpt_bhs`) to issue the I/Os in a batched manner. Similar to the commit procedure, the `jh` is removed and re-inserted into a checkpoint io list, which is used for I/O completion. If the number of aggregated buffers (`batch_count`) is higher than the pre-defined threshold, the thread releases the list lock and issues the I/Os. Then, the thread prepares for the next I/Os by acquiring the list lock. After issuing all the I/Os, the thread completes them one by one by fetching the `jh` from the checkpoint io list. When fetching the `jh`, the thread uses the list lock. After then, the thread sets the next transaction to be checkpointed in the checkpoint transaction list under the list lock. Finally, the checkpointed transaction is freed, which denotes the end of a life cycle of the transaction, and the list lock and the mutex lock are released.

3 Design and Implementation

To achieve higher I/O performance on multi-cores with high-performance storage, we aim to reduce the lock contention and maximize I/O parallelism in transaction

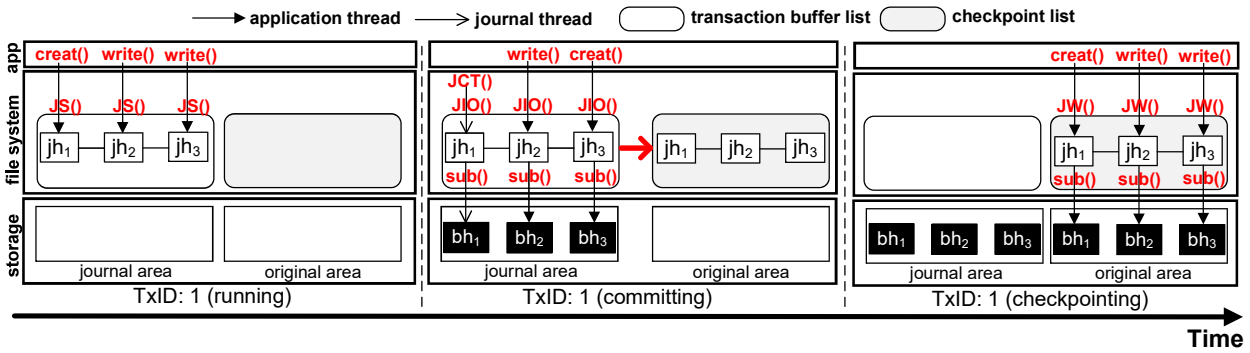


Figure 3: Overall architecture (app: application, jh: journal_head, bh: buffer_head, TxID: transaction ID, JS(): jbd2_journal_start(), JCT(): jbd2_journal_commit_transaction(), JIO(): journal_io_start(), JW(): jbd2_log_wait_for_space(), sub(): submit_bh())

processing. To do this, we propose a transaction processing with two main schemes that enable concurrent updates on shared data structures and cooperatively parallelize I/O operations. We apply these schemes to the transaction processing in EXT4/JBD2.

We maintain the compound transaction scheme of EXT4/JBD2 to exploit its advantages [23]. For example, it provides a better performance when the same metadata or data is frequently updated within a short period of time. With this advantage, we implement our schemes in the compound transaction. We also preserve the existing ordering of write operations and transactions, such as the ordering of journal blocks and a commit block, committing and checkpointing, and checkpoints. Therefore, our schemes do not sacrifice the consistency of the file system.

Furthermore, we do not optimize all locking operations in transaction processing but focus on the list lock for management of journal heads and the checkpoint mutex lock for serialized I/O operations. Compared to the list lock and the mutex lock, other locks (e.g., state lock) do not incur a significant overhead according to our evaluation, as well as other works [16]. However, such locks can be a performance bottleneck in a massive number of cores, which is beyond this paper; therefore, we leave the latent performance issue as a future work.

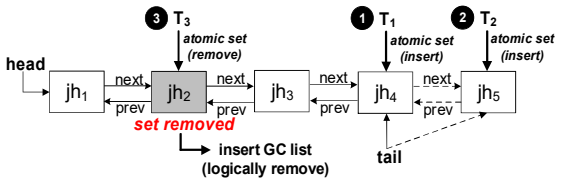
3.1 Design

Figure 3 shows an overall architecture of proposed schemes. When application threads update the metadata and data by calling system calls, such as `creat()` and `write()`, they start a transaction (i.e., TxID: 1) by calling `jbd2_journal_start()`. They insert their own modified buffer into the transaction buffer list concurrently. When the transaction needs to commit, the journal thread begins the commit process by calling `jbd2_journal_commit_transaction()` and starts journal I/Os. Application threads, which cannot create nor join a running transaction, join and perform the journal I/Os with the journal thread by calling `journal_io_start()`. They fetch the buffers in the transaction buffer list concurrently and write them to the journal area in parallel by calling `submit_bh()`. Then, the threads concurrently insert the committed buffers into the checkpoint list. When the space for journaling is not enough, application threads start to perform the checkpoint I/Os by calling `jbd2_log_wait_for_space()`. They fetch the committed buffers in the checkpoint list concurrently and write them to the original area in parallel by calling `submit_bh()`.

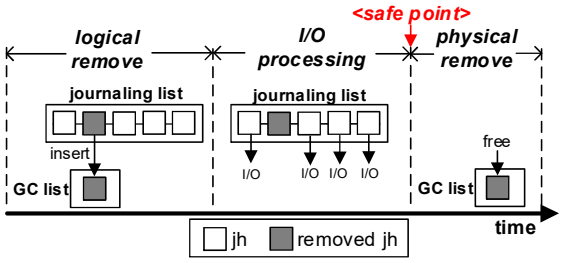
3.1.1 Concurrent updates on data structures

We manage the linked lists for transaction processing in a lock-free manner as shown in Figure 4. To this end, instead of the existing circular doubly linked lists, we use non-circular doubly linked lists and add the `tail` to the lists to enable lock-free operations. In the circular doubly linked list, when an item is inserted into the list, the multiple pointers that link the item, head, and tail are updated, which makes the atomic insert operation difficult. Instead, we add the tail and set the tail's next item as a constant *NULL* variable [1], which allows us to identify the last element of the list and insert the item into the tail atomically.

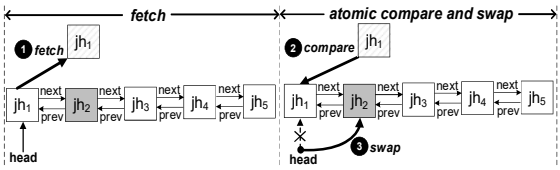
INSERT. We provide a concurrent insert operation to add an item to a list. In the existing transaction processing, the items are inserted into the tail of the list in the incoming order. Similar to the existing scheme but without locking, we concurrently update the tail by the incoming items using an atomic set instruction. In an example shown in Figure 4(a), before jh_5 is inserted into a journaling list (e.g., transaction buffer list or checkpoint list), the journaling list consists of four `jh`s, and the tail points jh_4 which is inserted by T_1. When T_2 inserts jh_5, the thread atomically updates the tail and the jh_5's previous item by jh_5 and jh_4, respectively, by executing the atomic set operation. By updating the previous item (jh_4) of jh_5 atomically, the next item of jh_4 is decided as



(a) Insert and remove operations in a lock-free manner (T: thread)



(b) Two-phase removal (GC: garbage collection)



(c) Fetch operations in a lock-free manner

Figure 4: Concurrent updates on data structures (jh: journal_head)

jh_5. This insert operation allows multiple threads to add their item concurrently by updating the tail and linking atomically.

REMOVE. We provide a concurrent remove operation to delete an item from a list. When items are removed from the list concurrently without locking, the *invalid reference* problem [22] can occur. To address this issue, we propose a two-phase remove operation that marks an item as "removed" (logical remove) and then frees the item (physical remove) at a *safe point* when no other threads hold any references to the transaction and logically removed items. This scheme ensures safe access to the items of the list, and thus, threads can perform appropriate operations for the items. For the safe garbage collection (GC) of the logically removed items, we additionally maintain a GC list per transaction.

For example, as shown in Figure 4(a), when a thread (T_3) tries to remove the `jh` (jh_2), the thread marks the `jh` as *removed* atomically by executing the atomic set instruction. Then, the thread inserts the `jh` into the GC list using our concurrent insert operation as shown in Figure 4(b). And then, the threads perform I/O for the valid `jh` or bypass the I/O for the logically removed `jh` while traversing the list safely. When the transaction arrives at the *safe point*, all items in the GC list are reclaimed. The *safe point* is the point when a transaction is checkpointed. At this point, no other threads reference the logically removed `jh`s in the transaction nor insert any logically removed `jh`s into the GC list of the transaction since all the transaction processing is over. Therefore, we can free all the logically "removed" `jh`s at the *safe point*.

FETCH. Finally, we provide a concurrent fetch operation to get an item while traversing a list. In the existing transaction processing, the list traversal occurs when no threads insert any items into the list (e.g., journal and checkpoint I/O processing). This ensures that all threads see a consistent view of the list, including valid next pointers of all items. Under this condition, we can simply enable the concurrent fetch operation by using an atomic compare and swap (CAS) instruction. In the example shown in Figure 4(c), a thread first fetches the current head (jh_1). Then, the thread compares the fetched jh_1 with the current head and changes the head to jh_1's next item by using the CAS operation. If the thread fails the CAS operation, it repeats the procedure above. This fetch operation allows multiple threads to extract individual items concurrently by updating the head atomically. Consequently, through our concurrent update scheme, multiple threads can insert/remove/fetch their items in the lists concurrently and safely without the existing list lock.

3.1.2 Parallel I/O in a cooperative manner

We provide a parallel I/O in a cooperative manner to maximize the I/O parallelism. In the existing transaction processing, application threads can be blocked while the serialized I/O operations (e.g., journal and checkpoint I/O) are performed. On the other hand, in our scheme, we allow the application threads to perform the I/O operations by not blocking but joining them to the I/O operations. For example, in the case of journal I/O, we allow the threads that cannot get a running transaction to join the I/Os by not blocking them. In the case of checkpoint I/O, we allow the threads to join the I/Os by eliminating the mutex lock. By joining the multiple threads to the I/O processing, they fetch buffers from the shared linked lists (e.g., journaling lists), issue the I/Os of the buffers, and complete them in parallel. For better parallelism, we use our concurrent fetch operation and per-thread wait list, which is a linked list used to wait for the I/O completion in parallel.

As shown in Figure 5, each thread fetches the `jh` concurrently by executing the atomic CAS instruction. Then, each thread issues the I/O of the buffer (i.e., `bh`) associated with the `jh` and inserts the buffer into its own wait list. After all the I/Os are issued, each thread completes its own I/O using its own wait list. Meanwhile, if the fetched `jh` was removed logically, the thread (T_2) does not perform the I/O for the `jh` but fetches the next `jh`. Using this scheme, multiple threads can cooperate

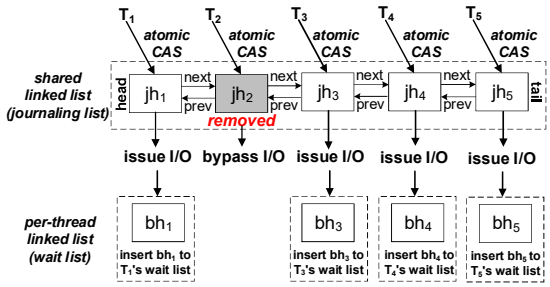


Figure 5: Parallel I/O in a cooperative manner (T: thread, jh: journal_head, bh: buffer_head)

in I/O processing by issuing/completing I/Os in parallel. This can make a commit and checkpoint procedure faster by increasing the I/O parallelism and minimizing the blocking time. We note that our parallel I/O operations can change the I/O ordering between buffers inside a transaction. However, such a change does not sacrifice the atomicity since we write the commit block after all journal blocks are written, which will be described in Section 3.2.2.

The optimized file system with our two schemes preserves the consistency of the file system by satisfying the following properties: (1) Every block associated with a transaction is written to the journal area at a commit procedure. (2) A transaction is committed or uncommitted (atomicity) according to the commit block. (3) Committed transaction N-1 is checkpointed prior to committed transaction N. We will explain how to apply our schemes to transaction processing and how to satisfy the properties in detail.

3.2 Implementation

3.2.1 Running transaction

We enable multiple application threads to insert/remove the journal heads into/from the transaction buffer list concurrently. Similar to the existing procedure, when the threads start a transaction, they get a running transaction and increase the number of updates in the transaction (Procedure 1, lines 3-4 and 31-39). Meanwhile, in our running procedure, we allow the application threads to cooperate in I/O processing for journal I/Os by calling `journal_io_start()` (lines 32-33), which will be described in Section 3.2.2.

After getting the running transaction, we insert the `jh` into the transaction buffer list by using our concurrent insert operation (lines 5-6 and 44-51)[2]. First, the threads associate their `jh` with the running transaction. Then, they update the tail (`t_buffers_tail`) by their `jh` and the `jh`'s previous item by the old tail by executing the `atomic_set` instruction[3]. This instruction updates the tail and returns the old tail atomically. Then, the threads

PROCEDURE 1 C-like pseudo-code of our running transaction

```
1:  create(dir, ...){
2:      /* create a new file */
3:      handle = jbd2_journal_start(journal, ...);
4:      transaction = handle->transaction;
5:      add_buffer(bh->jh, transaction,
6:          transaction->t_buffers, transaction->t_buffers_tail);
7:      jbd2_journal_stop(handle);
8:  }

9:  truncate(dentry, ...){
10:     /* truncate a file */
11:     journal_unmap_buffer(journal, bh);
12: }

13: journal_unmap_buffer(journal, bh){
14:     /* invalidate a buffer */
15:     write_lock(journal->j_state_lock);
16:     transaction = bh->jh->transaction;
17:     if(!bh->jh->cp_transaction){
18:         head = jh->cp_transaction->gc_head;
19:         tail = jh->cp_transaction->gc_tail;
20:         del_buffer(jh, transaction, head, tail);
21:     }else if(transaction == journal->j_committing_transaction){
22:         set_buffer_free(bh);
23:         atomic_set(jh->removed, removed);
24:     }else if(transaction == journal->j_running_transaction){
25:         head = journal->j_running_transaction->gc_head;
26:         tail = journal->j_running_transaction->gc_tail;
27:         del_buffer(jh, transaction, head, tail);
28:     }
29:     write_unlock(journal->j_state_lock);
30: }

31: jbd2_journal_start(journal, ...){
32:     if(j_running_transaction is not available)
33:         /*create a new transaction or call journal_io_start(journal)*/
34:     read_lock(journal->j_state_lock);
35:     handle->transaction = journal->j_running_transaction;
36:     atomic_add(transaction->t_updates, 1);
37:     read_unlock(journal->j_state_lock);
38:     return handle;
39: }

40: jbd2_journal_stop(handle){
41:     /* complete a transaction */
42:     atomic_sub(handle->transaction->t_updates, 1);
43: }

44: add_buffer(jh, transaction, head, tail) {
45:     jh->transaction = transaction;
46:     jh->prev = atomic_set(tail, jh);
47:     if(jh->prev == NULL)
48:         head = jh;
49:     else
50:         jh->prev->next = jh;
51: }

52: del_buffer(jh, transaction, head, tail) {
53:     atomic_set(jh->removed, removed);
54:     jh->gc_prev = atomic_set(tail, jh);
55:     if(jh->gc_prev == NULL)
56:         head = jh;
57:     else
58:         jh->gc_prev->gc_next = jh;
59:     bh->jh = jh->bh = NULL; /* unlink the bh from the jh */
60:     jh->transaction = NULL;
61: }
```

[2]The `jh` is inserted into a transaction buffer list or checkpoint list by using the `prev/next/transaction` or `cpprev/cpnext/cp_transaction` fields of the `jh`, respectively.

[3]__sync_lock_test_and_set(type *ptr, type value): This built-in function performs an atomic exchange operation. It writes the value into *ptr and returns the previous contents of *ptr [28].

check whether the old tail exists or not. If it does not exist, the head (t_buffers) of the list is updated by the inserted jh, which becomes the first item in the list. Otherwise, the next item of the old tail is updated by the inserted jh.

For remove operations, we use our two-phase remove operation. When the threads remove their jh, they get the GC list of the transaction if the jh is associated with running or checkpoint transactions (lines 17-20 and 24-27). For the logical remove operation (lines 52-61), the thread marks the jh as *removed* by executing the atomic_set instruction and inserts the jh into the GC list atomically by using gc_prev/next fields of the jh. Then, the bh is unlinked from the removed jh (line 59), and the jh's transaction or cp_transaction field is set to NULL in the case of running or checkpointing transaction, respectively (line 60). This means that the jh is not associated with the bh and the transaction any longer. Thus, the jh becomes an obsolete structure, and the bh gets freed at this point. This operation on the bh is performed safely since the operation is protected by a spin lock (jbd_lock_bh_state) per bh as same as the existing scheme. Meanwhile, in the case of committing transaction, the thread only marks the jh as *removed* (line 23), and both bh and jh will be freed during the commit procedure.

3.2.2 Committing transaction

During the existing commit procedure, the journal thread updates the lists under the list lock and performs journal I/O operations by a single thread. On the other hand, in our commit procedure, we update the lists by using our concurrent update operations and parallelize the I/O operations in a cooperative manner.

To commit a transaction, the journal thread gets a committing transaction similar to the existing procedure (Procedure 2, lines 3-9). Then, the journal thread starts the parallel I/O by setting the journal_io variable (line 10). This informs application threads that the I/O processing is initiated. Note that in the existing procedure, application threads are blocked when a running transaction is not available and cannot be newly created. Instead of blocking the threads, we enable the threads to perform the I/O processing along with the journal thread by calling journal_io_start() (Procedure 1, line 33, Procedure 2, line 11, and Procedure 3, line 2). Thus, the threads can join the I/O processing if it is initiated by the journal thread (Procedure 3, lines 5-6).

To handle the joined threads, we record the number of threads by executing atomic_add/sub instructions[4] (Procedure 3, lines 7 and 20) and create the per-thread wait list for the parallel I/O completion (line 8). Then, we allow each thread to fetch the jh from the transaction buffer list by using our concurrent fetch operation, which executes the atomic_cas instruction[5] (lines 9-17). If the fetched jh was logically removed, the thread bypasses and retries to fetch the next jh. Otherwise, each thread

PROCEDURE 2 C-like pseudo-code of our committing transaction (1)

```
1:  /*the journal thread commits a transaction*/
2:  jbd2_journal_commit_transaction(journal){
3:      commit_transaction = journal->j_running_transaction;
4:      write_lock(journal->j_state_lock);
5:      journal->j_committing_transaction = commit_transaction;
6:      journal->j_running_transaction = NULL;
7:      while(atomic_read(transaction->t_updates)){...}
8:      write_unlock(journal->j_state_lock);
9:      transaction = journal->j_committing_transaction;
10:     atomic_set(transaction->journal_io, start);
11:     journal_io_start(journal);
12:     while(atomic_read(transaction->num_io_threads) != 0);
13:     <issue and complete a commit block>
14:     write_lock(journal->j_state_lock);
15:     <insert the committed transaction into a checkpoint transaction list
16:         (journal->j_checkpoint_transactions) using our concurrent insert>
17:     write_unlock(journal->j_state_lock);
18:     atomic_set(transaction->cp_io, start);
19: }
```

PROCEDURE 3 C-like pseudo-code of our committing transaction (2)

```
1:  /*the journal thread performs journal I/Os with application threads*/
2:  journal_io_start(journal){
3:      if((transaction = journal->j_committing_transaction) == NULL)
4:          return;
5:      if(atomic_read(transaction->journal_io) == stop)
6:          return;
7:      atomic_add(transaction->num_io_threads, 1);
8:      create_wait_list(local_wait_list); // create a local wait list per thread
9:      while((jh = transaction->t_buffers) != NULL){
10:         if(atomic_cas(transaction->t_buffers, jh, jh->next) != jh)
11:             continue;
12:         if(atomic_read(jh->removed) == removed)
13:             continue;
14:         <make a frozen buffer (frozen_bh)>
15:         submit_bh(WRITE, jh->frozen_bh);
16:         add_wait_list(local_wait_list, jh->frozen_bh);
17:     }
18:     atomic_set(transaction->journal_io, stop);
19:     wait_journal_io(wait_list);
20:     atomic_sub(transaction->num_io_threads, 1);
21: }

22: wait_journal_io(local_wait_list){
23:     while(!wait_list_empty(local_wait_list){
24:         frozen_bh = list_entry(local_wait_list.next, ...);
25:         wait_on_buffer(frozen_bh);
26:         jh = frozen_bh->bh->jh;
27:         jh->transaction = NULL;
28:         if(atomic_read(jh->removed) != removed && jbddirty(jh->bh))
29:             add_buffer(jh, transaction, transaction->t_checkpoint_list,
30:                 transaction->t_checkpoint_list_tail);
31:     }
32: }
```

[4] __sync_add/sub_and_fetch(type *ptr, type val): These built-in functions atomically add/subtract the value of val to/from the variable that *ptr points to. The functions return the new value of the variable that *ptr points to [28].

[5] __sync_val_compare_and_swap(type *ptr, type oldval, type newval): This built-in function performs an atomic compare and swap operation. If the current value of *ptr is oldval, then write newval into *ptr. Otherwise, no operation is performed. The function returns the contents of *ptr before the operation [28].

creates a frozen buffer, submits the I/O of the buffer to the journal area, and inserts the buffer into its own wait list in parallel.

After all the I/Os are issued, we stop new upcoming threads from joining the I/O processing by unsetting the `journal_io` variable (line 18). Then, the joined threads complete the I/O by using their own wait list (lines 19 and 22-32). Through the procedure above, the parallel I/O is completed by writing all the buffers to the journal area. This procedure satisfies the following property.

Property 1. *Every block associated with a transaction is written to the journal area at a commit procedure.*

Every application thread increases `t_update` *before inserting its* `jh` *(Procedure 1, line 36) and decreases* `t_update` *after inserting its* `jh` *(Procedure 1, line 42). Before the journal thread starts the parallel I/O processing by setting* `journal_io` *(Procedure 2, line 10), the thread waits until* `t_update` *becomes 0 (Procedure 2, line 7). This prevents application threads from starting and finishing the I/O processing before all the* `jh`*s are inserted into the transaction buffer list. Thus, it ensures that all the buffers associated with the transaction are written to the journal area even if the parallel I/O is enabled.* □

While completing the I/Os (Procedure 3, lines 22-32), the threads insert the `jh`s into a checkpoint list if the `jh`s are not removed logically and their buffers are still dirty. In this processing, for simplicity and efficiency, we make the checkpoint list while completing the I/Os before the commit block is written. However, the list is not used for checkpointing until the commit procedure is finished to preserve the ordering of committing and checkpointing.

In addition, we use the wait lists instead of the shadow list and include all the frozen buffers in the wait lists. Instead of the forget list, we use the GC list and insert the `jh`s which are associated with buffers to be freed to the GC list. After completing all the I/Os, the journal thread waits until all the journal I/Os are finished by using the number of joined threads before writing the commit block (Procedure 2, lines 12-13). This procedure satisfies the following property.

Property 2. *A transaction is committed or uncommitted (atomicity) according to the commit block.*

Every application thread that joins the I/O processing increases `num_io_threads` *before issuing I/O (Procedure 3, line 7) and decreases* `num_io_threads` *after completing I/O (Procedure 3, line 20). The journal thread waits until* `num_io_threads` *becomes 0 before the journal thread writes the commit block (Procedure 2, line 12). This means that all the journal blocks are written before the commit block is written to the journal area. Thus, it ensures the atomicity of the transaction by preserving the ordering between the journal blocks and the commit block.* □

Finally, the journal thread inserts the committed transaction into the checkpoint transaction list by using the state lock (`j_state_lock`) and our concurrent insert operation, and sets the `cp_io` variable to start the checkpoint I/O (lines 14-18).

3.2.3 Checkpointing transaction

In the existing procedure, when a transaction needs to be checkpointed, an application thread performs checkpoint I/O operations by acquiring a checkpoint mutex lock (`j_checkpoint_mutex`). Meanwhile, other application threads, which fail to acquire the lock, are blocked until the checkpoint is finished, which can underutilize the I/O parallelism.

To enable a parallel checkpoint I/O, we allow the threads to join the I/O processing instead of using the mutex lock and the checkpoint buffer. However, even with the parallel I/O, the I/O issue/complete operations are still inefficient since the list lock is used to fetch/insert the `jh`s from/into the checkpoint/checkpoint io lists. Thus, we fetch the `jh`s by using our concurrent fetch operation, issue the I/Os, and complete the I/Os by using the per-thread wait list in parallel instead of the global checkpoint io list.

When a checkpoint is triggered, application threads get a transaction to be checkpointed if the transaction is available (Procedure 4, lines 2-3). Then, the threads check whether the transaction can be checkpointed or not by using the `cp_io` variable (lines 4-5). Similar to our commit procedure, we record the number of joined threads, and each thread creates its own wait list (lines 6-7). For the concurrent and parallel I/O issue, each thread concurrently fetches the `jh` from the checkpoint list, submits the I/O of the buffer associated with the `jh` to the original area, and inserts the buffer into the wait list of each thread in parallel (lines 8-15). If the fetched `jh` was removed logically, the thread retries to fetch the next `jh`. After issuing all the I/Os, we stop new upcoming threads from joining the I/O processing by unsetting the `cp_io` variable (line 16). Then, the joined threads disassociate the `jh`s from the transaction while completing the I/Os (lines 17 and 28-34).

After completing all the I/Os, we find the last remaining thread by decreasing the number of joined threads (line 18). The last thread sets the next transaction to be checkpointed by updating the head of the checkpoint transaction list to the next of the head using the atomic CAS operation (lines 19-20). This procedure satisfies the following property.

Property 3. *Committed transaction N-1 is checkpointed prior to committed transaction N.*

PROCEDURE 4 C-like pseudo-code of our checkpointing transaction

```
 1: jbd2_log_wait_for_space(journal){
 2:     if((transaction = journal->j_checkpoint_transactions) == NULL)
 3:         return;
 4:     if(atomic_read(transaction->cp_io) == stop)
 5:         return;
 6:     atomic_add(transaction->cp_num_io_threads, 1);
 7:     create_wait_list(local_wait_list); // create a local wait list per thread
 8:     while((jh = transaction->t_checkpoint_list) != NULL){
 9:         if(atomic_cas(transaction->t_checkpoint_list, jh, jh->next) != jh))
10:             continue;
11:         if(atomic_read(jh->removed) == removed)
12:             continue;
13:         submit_bh(WRITE, jh->bh);
14:         add_wait_list(local_wait_list, jh->bh);
15:     }
16:     atomic_set(transaction->cp_io, stop);
17:     wait_cp_io(local_wait_list);
18:     if(atomic_sub(transaction->cp_num_io_threads, 1) == 0){
19:         <set the next transaction to be checkpointed
20:                 in the checkpoint transaction list using atomic_cas>
21:         while((jh = transaction->gc_head) != NULL){
22:             transaction->gc_head = jh->gc_next;
23:             free(jh);
24:         }
25:         free(transaction);
26:     }
27: }

28: wait_cp_io(local_wait_list){
29:     while(!wait_list_empty(local_wait_list){
30:         bh = list_entry(local_wait_list.next, ...);
31:         wait_on_buffer(bh);
32:         bh->jh->cp_transaction = NULL;
33:     }
34: }
```

A committed transaction is inserted into the tail of the checkpoint transaction list in the committed order (Procedure 2, lines 15-16). The last thread sets the next transaction to be checkpointed in the checkpoint transaction list in the committed order (Procedure 4, lines 19-20). This means that if transaction N-1 is committed prior to transaction N, transaction N is not checkpointed prior to transaction N-1. Thus, it ensures that all the buffers in the transaction are written to the original area in the committed order. Consequently, our optimized file system preserves the consistency of the file system by satisfying Property 1, 2, and 3. □

Then, the last thread physically removes all the obsolete `jh`s in the GC list of the transaction (lines 21-24). At this point, we can reclaim the `jh`s safely. It is because all the transaction processing is ended: (1) No other threads reference the logically removed `jh`s in the transaction since all the I/O processing is ended. (2) No other threads insert any logically removed `jh`s into the GC list of the transaction since all the `jh`s in the transaction are disassociated from the transaction. Finally, the last thread frees the checkpointed transaction (line 25).

3.2.4 Recovery

In the existing recovery procedure, a single-threaded process (i.e., mount process) performs the recovery operation. To optimize the recovery procedure, we create multiple threads to perform scan and replay I/O operations in parallel and do not use any additional lock. When multiple threads start the scan operation, each thread makes a local wait list to complete the I/Os in parallel. Then, it atomically gets its own offset which is the logical position in the journal area by executing the `atomic_add` instruction. Each thread gets its own buffer based on the offset, issues the read request for the buffer in parallel, and inserts the buffer into own wait list. This process is repeated for all the buffers which need to be scanned. After the threads complete the I/Os for the scan operation in parallel, they concurrently insert the buffers included in their own wait list into a global list for the replay operation.

In the case of the replay operation, each thread makes a local wait list similar to the case of the scan operation and concurrently fetches the buffers to be replayed from the list. Then, it issues the write request in parallel and inserts the buffer into its own wait list. After issuing all the I/Os for the replay operation, the threads complete the I/Os in parallel. This recovery scheme makes the recovery procedure faster and more efficient.

4 Evaluation

4.1 Experimental setup

We perform all of the experiments on a 72-core machine with four Intel Xeon E7-8870 processors (without hyperthreading), 16 GiB DRAM, and PCI 3.0 interface. For storage, the machine has an 800 GiB Intel P3700 NVMe SSD [14], which has 18 channels. The machine runs Ubuntu 16.04.1 LTS distribution with a Linux kernel 4.9.1. We evaluate the existing EXT4 and fully optimized EXT4 (O-EXT4) file systems in the ordered (default) and data journaling modes. To present a performance breakdown, we also evaluate an optimized EXT4 with our parallel I/O scheme (P-EXT4). In P-EXT4, we allow the application threads to perform the I/O operations by not blocking but joining them to the journal and checkpoint I/Os in a parallel and cooperative manner. However, we still update the data structures using `j_list_lock`. Through this evaluation, we compare the performance of our two schemes. We run metadata and data-intensive workloads, such as tokubench [9], sysbench [18], and filebench [34] with the parameters shown in Table 1. We vary the number of cores from 1 to 72, and the number of threads is equal to that of the cores. We run each test ten times and report the average.

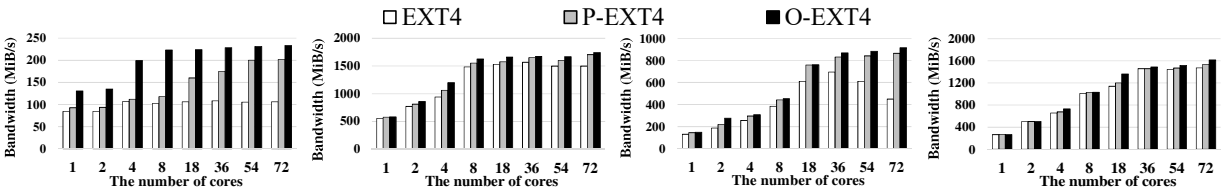
4.2 Performance results

4.2.1 Ordered mode

We present the performance results in the ordered mode as shown in Figure 6. In the case of tokubench as

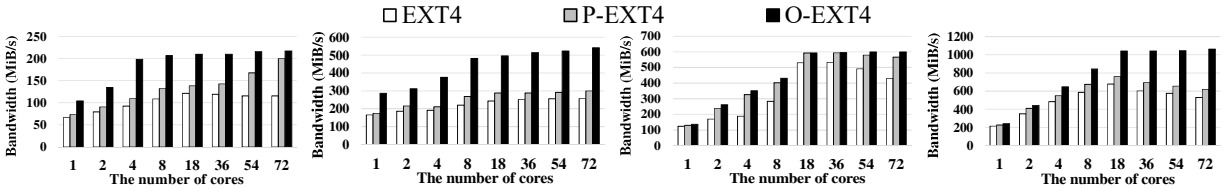
Benchmarks	Descriptions	Parameters
Tokubench (micro benchmark)	Metadata-intensive workload (file creation)	Files: 30,000,000, I/O sizes: 4KiB
Sysbench (micro benchmark)	Data-intensive workload (random write)	Files: 72, Each file size: 1GiB, I/O sizes: 4KiB
Filebench Varmail (macro benchmark)	Metadata-intensive workload (read/write ratio = 1:1)	Files: 300,000, Directory width: 10,000
Filebench Fileserver (macro benchmark)	Data-intensive workload (read/write ratio = 1:2)	Files: 1,000,000, Directory width: 10,000

Table 1: Workload descriptions and parameters



(a) Tokubench (b) Sysbench (c) Varmail (d) Fileserver

Figure 6: Ordered mode

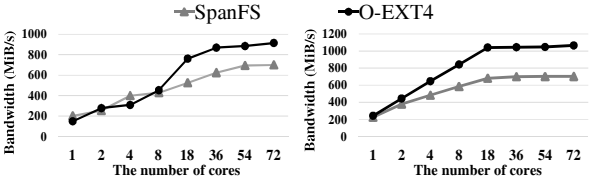


(a) Tokubench (b) Sysbench (c) Varmail (d) Fileserver

Figure 7: Data journaling mode

shown in Figure 6(a), the performance growth of EXT4 is not noticeable as the number of cores increases. P-EXT4 improves the performance by 1.9x compared to EXT4. However, compared to full optimization, this result shows the limitation of our parallel I/O scheme, which does not handle the lock contention. Through full optimization, O-EXT4 improves the performance by 2.2x at 72 cores compared to EXT4. Meanwhile, the performance of O-EXT4 is almost the same beyond 18 cores since the bandwidth is saturated due to the limited write bandwidth and the channels of the SSD. In the case of sysbench as shown in Figure 6(b), P-EXT4 and O-EXT4 improve the performance by 13.8% and 16.3%, respectively, compared to EXT4 at 72 cores. The performance improvement is lower than that of tokubench since sysbench as a data-intensive workload generates far fewer journal I/Os for metadata.

Under the varmail workload as shown in Figure 6(c), P-EXT4 and O-EXT4 scale well compared to the case of tokubench and outperform EXT4 by 1.92x and 2.03x at 72 cores, respectively. O-EXT4 achieves up to 914.3 MiB/s. Since the workload generates a mixture of read-/write operations unlike tokubench, the available bandwidth increases, and therefore, the performance gradually scales at all cores. Meanwhile, the performance of EXT4 decreases beyond 54 cores due to the lock contention. Under the fileserver workload as shown in Figure 6(d), P-EXT4 and O-EXT4 outperform EXT4 by 4.3% and 9.6% at 72 cores, respectively. All the file systems scale in a similar trend at each core, and the



(a) Varmail (b) Fileserver

Figure 8: Comparison with SpanFS

performance gap is not noticeable. The reason is that, similar to the case of sysbench, the fileserver workload is data-intensive, which generates a low number of metadata I/Os. Consequently, our optimized file system improves the performance in the ordered mode by reducing the lock contention and parallelizing the I/O operations, especially for metadata-intensive workloads.

4.2.2 Data journaling mode

We present the performance results in the data journaling mode as shown in Figure 7. In the case of tokubench as shown in Figure 7(a), P-EXT4 and O-EXT4 outperform EXT4 by 73% and 88.2% at 72 cores, respectively. The results show that the overall aspect of the performance is similar to that in the ordered mode. In the case of sysbench as shown in Figure 7(b), P-EXT4 and O-EXT4 show 1.17x and 2.1x faster performance than EXT4 at 72 cores, respectively. The performance improvement is higher than that in the ordered mode since the workload generates many journal I/Os for data. Also, the results show that the improvement by our parallel I/O scheme is low due to the list lock contention.

File systems	Device-level bandwidth	Write time	j_checkpoint_mutex	j_list_lock	j_state_lock	Others
EXT4	692 MiB/s	52220 s (100%)	17946 s (34.4%)	6132 s (11.7%)	102 s (0.2%)	28040 s (53.7%)
P-EXT4	805 MiB/s	45124 s (100%)	0	4890 s (10.8%)	87 s (0.2%)	40147 s (89%)
O-EXT4	1426 MiB/s	25078 s (100%)	0	0	182 s (0.7%)	24896 s (99.3%)

Table 2: Device-level bandwidth and total execution time of main locks and write operations

Modes	Ordered			Data journaling		
Operations	scan	replay	other	scan	replay	other
EXT4	331 ms	62 ms	7 ms	311 ms	81 ms	5 ms
O-EXT4	125 ms	34 ms	9 ms	117 ms	37 ms	4 ms

Table 3: Recovery performance

Under the varmail workload as shown in Figure 7(c), P-EXT4 and O-EXT4 outperform EXT4 by 31.3% and 39.3% at 72 cores, respectively. Unlike the case of the ordered mode, the performance is saturated and sustained beyond 18 cores since writing both the metadata and the data makes the performance reach the full bandwidth faster. Meanwhile, the performance of EXT4 decreases due to the lock contention. In the case of fileserver as shown in Figure 7(d), P-EXT4 and O-EXT4 outperform EXT4 by 1.17x and 2.01x at 72 cores, respectively. O-EXT4 achieves up to 1064.6 MiB/s. The performance of P-EXT4 and EXT4 decreases beyond 36 cores, which demonstrates the need for both concurrent updates on data structures and parallel I/O. Meanwhile, O-EXT4 scales well to 18 cores and increases the performance until 72 cores. Beyond 36 cores, the rate of bandwidth growth is reduced due to the bandwidth limit of the SSD. Consequently, our optimized file system achieves higher performance in the data journaling mode, and the benefit becomes larger in data-intensive workloads.

4.2.3 Comparison with a scalable file system

We compare our optimized file system with SpanFS [16], a scalable file system. We use the varmail and fileserver workloads in the ordered and data journaling modes, respectively. We set the number of domains in SpanFS as same as that of the cores. As shown in Figure 8, both file systems scale well until the performance is saturated in both workloads. Meanwhile, O-EXT4 generally shows better performance and improves the performance by up to 1.45x and 1.51x in the varmail and fileserver workloads, respectively, compared to SpanFS. Especially, in the case of the varmail workload, the performance of O-EXT4 is similar or slower than that of SpanFS at a small number of cores while O-EXT4 shows better performance than SpanFS as the number of cores increases. The results show that our scheme can deliver better performance than the scheme that distributes file services.

4.3 Experimental analysis

Table 2 shows the total execution time for the main locks and the device-level bandwidth at 72 cores in the case of the sysbench workload in the data journaling mode. For this experiment, we measured the execution time by using a time function (`getrawmonotonic()`) for lower overhead and more correctness. As shown in the table, in EXT4, the execution time of the checkpoint mutex and list locks take a large portion of the total write time. In P-EXT4, the bandwidth increases by 16.3%, and the write time decreases by 15.7% compared to EXT4, respectively. As the total write time decreases, the time of the list and state locks decreases while the list lock still takes up 10.8% of the total write time. This demonstrates that the list lock contention can be a performance bottleneck in our parallel I/O scheme. In O-EXT4, the bandwidth increases by 2.06x, and the write time decreases by 2.08x compared to EXT4. This is achieved by removing the list lock contention via our concurrent update scheme. Meanwhile, the contention on the state lock increases due to the removal of the list lock but the portion is still small. Consequently, this result demonstrates that O-EXT4 achieves high-performance transaction processing by enabling both concurrent updates and parallel I/O.

4.4 Recovery performance

To evaluate the recovery performance, we used tokubench and fileserver workloads in the ordered and data journaling modes, respectively. While running the benchmarks, we randomly cut the power of the machine, and both existing and optimized file systems are recovered to a consistent state after more than 30 crashes. Table 3 shows the recovery performance of the ordered and data journaling modes in the file systems. The scan and replay operations occupy the main part of the total recovery time in all cases. Through parallelizing scan and replay I/O operations, O-EXT4 improves the recovery performance by 2.38x and 2.51x compared to EXT4 in the ordered and data journaling modes, respectively. This result demonstrates that our schemes can also be applied to the recovery procedure to provide faster recovery time.

5 Related Work

Lock-free data structures. Valois [33] provides lock-free data structures and algorithms for implementing a shared singly-linked list, allowing concurrent traversal, insertion, and deletion. Zhang et al. [35] introduce new lock-free and wait-free unordered linked list algorithms. They provide the first practical implementation of the unordered linked list that supports wait-free insert, remove, and lookup operations. Our study is inspired by these works [33, 35], and we use a variant of these implementations and apply it to transaction processing in a journaling file system.

Scalable database systems. Silo [30] is an in-memory database system designed for multi-core machines. Silo implements a variant of optimistic concurrency control in which transactions write their updates to shared memory only at commit time and uses a decentralized timestamp based technique to validate transactions at commit time. SiloR [37] adds additional features, such as logging, checkpointing, and recovery to Silo. It uses concurrency in all parts of the system. For example, the log is written concurrently to several disks, and a checkpoint is taken by several concurrent threads that also write to multiple disks. Our study is in line with these works [30, 37] in terms of investigating the multi-core scalability but we focus on the transaction processing in the file system.

Scalable kernels. Cerberus [27] mitigates contention on many shared data structures within OS kernels by clustering multiple commodity operating systems atop a virtual machine monitor. Boyd-Wickizer et al. [4] analyze the scalability of seven system applications running on Linux. They find that all applications trigger scalability bottlenecks inside a Linux kernel. RadixVM [7] presents a scalable virtual memory address space for non-overlapping operations. It avoids cache line contention using three techniques, which are radix trees, Refcache, and targeted TLB shootdowns. Our study is inspired by these works [27, 4, 7] and in line with them in terms of investigating the scalability of OS kernels on multi-cores. In contrast, we focus on transaction processing in file systems on high-performance storage.

Scalable storage stacks. Zheng et al. [36] present a storage system for arrays of commodity SSDs. They create dedicated I/O threads for each SSD and deploy a set-associative parallel page cache, which divides the global page cache into small and independent sets to reduce lock contention. MultiLanes [15] is a virtualized storage system for OS-level virtualization on many cores. It builds an isolated I/O stack on top of a virtualized storage device to eliminate contention on shared kernel data structures and locks. Our study is in line with these works [36, 15] in terms of mitigating the contention on shared resources. In contrast, we focus on updating the data structures concurrently in a lock-free manner in journaling file systems.

Scalable file systems. IceFS [19] partitions the on-disk resources among a new container abstraction called cubes to provide isolated I/O stacks for localized reaction to faults, fast recovery, and concurrent file system updates. Thus, the data and I/O within each cube are disentangled from the data and I/O outside of it. SpanFS [16] is a scalable file system that consists of a collection of micro file system services called domains. It distributes the files and directories among the domains and provides a global file system view on top of the domains to maintain consistency. Each domain performs its file system service, such as data allocation and journaling, independently. Curtis-Maury et al. [8] present a data partitioning mode to parallelize the majority of file system operations. They also provide a fine-grained lock-based multiprocessor model for incremental advances in parallelism.

Min et al. [21] analyze the many-core scalability of five file systems by using their open source benchmark suite (i.e., FxMark). They observe that file systems are hidden scalability bottlenecks in many I/O-intensive applications. iJournaling [23] improves the performance of an `fsync()` call. It journals only the corresponding file-level transaction to the ijournal area for an fsync call while exploiting the advantage of the compound transaction scheme. iJournaling also handles multiple fsync calls simultaneously by allowing each core to have its own ijournal area to improve the scalability. ScaleFS [3] decouples the in-memory file system from the on-disk file system using per-core operation logs to improve many-core scalability. ScaleFS delays propagating updates to the disk until an `fsync` call, which merges the per-core logs and applies the operations to disk. In contrast with our scheme, ScaleFS uses the per-core log to avoid the lock contention and timestamp to sort the operations in the log. Our study is in line with these approaches [19, 16, 8, 21, 23, 3] in terms of investigating the scalability and parallelism of the file systems. In contrast, we enable concurrent updates on data structures in a lock-free manner and parallelize I/O operations cooperatively in transaction processing by focusing its internal operations.

6 Conclusion and Future Work

In this paper, we investigate the locking and I/O operations in transaction processing of the journaling file system. We find that the lock contention on shared data structures and I/O operations by a single thread can be performance bottlenecks on a multi-core platform incorporating high-performance storage. To handle this issue, we present a transaction processing with concurrent updates on data structures and parallel I/O operations. Experiments show that our optimized file system achieves higher performance and scales better than the existing file system. In future work, we will design and implement lock-free mechanisms to handle the locks for other shared resources, such as file, page cache, etc., and evaluate them in different storage environments.

7 Acknowledgments

We thank our shepherd, Dushyanth Narayanan, and anonymous reviewers for valuable comments that greatly improved our paper. This research was supported by National Research Foundation of Korea (NRF) (2015M3C4A7065645, 2015M3C4A7065646, 2016R1D1A1B03934393).

References

[1] APT, K., DE BOER, F. S., AND OLDEROG, E.-R. *Verification of sequential and concurrent programs*. Springer Science & Business Media, 2010.

[2] ARPACI-DUSSEAU, A. C. Model-based failure analysis of journaling file systems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2005), DSN '05, IEEE Computer Society, pp. 802–811.

[3] BHAT, S. S., EQBAL, R., CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 69–86.

[4] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., ZELDOVICH, N., ET AL. An analysis of linux scalability to many cores. In *OSDI* (2010), vol. 10, pp. 86–93.

[5] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 228–243.

[6] CHUTANI, S., ANDERSON, O. T., KAZAR, M. L., LEVERETT, B. W., MASON, W. A., SIDEBOTHAM, R. N., ET AL. The episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference* (1992), San Fransisco, CA, USA, pp. 43–60.

[7] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 211–224.

[8] CURTIS-MAURY, M., DEVADAS, V., FANG, V., AND KULKARNI, A. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association, pp. 419–434.

[9] ESMET, J., BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. The tokufs streaming file system. In *HotStorage* (2012).

[10] FASHEH, M. Ocfs2: The oracle clustered file system, version 2. In *Proceedings of the 2006 Linux Symposium* (2006), Citeseer, pp. 289–302.

[11] GRAY, J., AND REUTER, A. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[12] HAGMANN, R. *Reimplementing the Cedar file system using logging and group commit*, vol. 21. ACM, 1987.

[13] HATZIELEFTHERIOU, A., AND ANASTASIADIS, S. V. Improving bandwidth efficiency for consistent multistream storage. *Trans. Storage 9*, 1 (Mar. 2013), 2:1–2:27.

[14] INTEL SOLID STATE DRIVE DC P3700 SERIES. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3700-spec.pdf`, 2015.

[15] KANG, J., HU, C., WO, T., ZHAI, Y., ZHANG, B., AND HUAI, J. Multilanes: Providing virtualized storage for os-level virtualization on manycores. *Trans. Storage 12*, 3 (June 2016), 12:1–12:31.

[16] KANG, J., ZHANG, B., WO, T., YU, W., DU, L., MA, S., AND HUAI, J. Spanfs: A scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 249–261.

[17] KIM, D., PARK, J., LEE, K.-G., AND LEE, S. *Forensic Analysis of Android Phone Using Ext4 File System Journal Log*. Springer Netherlands, Dordrecht, 2012, pp. 435–446.

[18] KOPYTOV, A. Sysbench: a system performance benchmark. *URL: http://sysbench. sourceforge. net* (2004).

[19] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical disentanglement in a container-based file system. In *OSDI* (2014), pp. 81–96.

[20] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., VIVIER, L., AND S, B. S. A. A and viver, l. the new ext4 filesystem: current status and future plans. In *In Ottawa Linux Symposium. http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf* (2007).

[21] MIN, C., KASHYAP, S., MAASS, S., AND KIM, T. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 71–85.

[22] ÖSTLUND, J., AND WRIGSTAD, T. Multiple aggregate entry points for ownership types. *ECOOP 2012–Object-Oriented Programming* (2012), 156–180.

[23] PARK, D., AND SHIN, D. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 787–798.

[24] PILLAI, T. S., ALAGAPPAN, R., LU, L., CHIDAMBARAM, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Application crash consistency and performance with ccfs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association.

[25] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track* (2005), vol. 194, pp. 196–215.

[26] REISER, H. Reiserfs, 2004.

[27] SONG, X., CHEN, H., CHEN, R., WANG, Y., AND ZANG, B. A case for scaling applications to many-core with os clustering. In *Proceedings of the Sixth Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys'11, ACM, pp. 61–76.

[28] STALLMAN, R. M., AND DEVELOPERCOMMUNITY, G. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA, 2009.

[29] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the xfs file system. In *USENIX Annual Technical Conference* (1996), vol. 15.

[30] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 18–32.

[31] TWEEDIE, S. Ext3, journaling filesystem. In *Ottawa Linux Symposium* (2000), pp. 24–29.

[32] TWEEDIE, S. C. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo* (1998).

[33] VALOIS, J. D. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1995), PODC '95, ACM, pp. 214–222.

[34] WILSON, A. The new and improved filebench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies* (2008).

[35] ZHANG, K., ZHAO, Y., YANG, Y., LIU, Y., AND SPEAR, M. Practical non-blocking unordered lists. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205* (New York, NY, USA, 2013), DISC 2013, Springer-Verlag New York, Inc., pp. 239–253.

[36] ZHENG, D., BURNS, R., AND SZALAY, A. S. Toward millions of file system iops on low-cost, commodity hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2013), SC '13, ACM, pp. 69:1–69:12.

[37] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 465–477.

Designing a True Direct-Access File System with DevFS

Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison

Yuangang Wang, Jun Xu, Gopinath Palani
Huawei Technologies

Abstract

We present DevFS, a direct-access file system embedded completely within a storage device. DevFS provides direct, concurrent access without compromising file system integrity, crash consistency, and security. A novel reverse-caching mechanism enables the usage of host memory for inactive objects, thus reducing memory load upon the device. Evaluation of an emulated DevFS prototype shows more than 2x higher I/O throughput with direct access and up to a 5x reduction in device RAM utilization.

1 Introduction

The world of storage, after decades of focus on hard-drive technologies, is finally opening up towards a new era of fast solid-state storage devices. Flash-based SSDs have become standard technology, forming a new performance tier in the modern datacenter [7, 32]. New, faster flash memory technologies such as NVMe [20] and storage class memory (SCM) such as Intel's 3D X-point [1] promise to revolutionize how we access and store persistent data [10, 13, 50, 53]. State-of-the-art flash memory technologies have reduced storage-access latency to tens of microseconds compared to milliseconds in the hard-drive era [34, 52, 58].

To fully realize the potential of these storage devices, a careful reconsideration of the software storage stack is required. The traditional storage stack requires applications to trap into the OS and interact with multiple software layers such as the in-memory buffer cache, file system, and device drivers. While spending millions of cycles is not a significant problem for slow storage devices such as hard drives [3, 13, 58], for modern ultra-fast storage, software interactions substantially amplify access latencies, thus preventing applications from exploiting hardware benefits [3, 9, 34, 50]. Even the simple act of trapping into and returning from the OS is too costly for modern storage hardware [14, 49, 58].

To reduce OS-level overheads and provide direct storage access for applications, prior work such as Arrakis [34], Moneta-D [8], Strata [26], and others [20, 49, 50] split the file system into user-level and kernel-level components. The user-level component handles all data-plane operations (thus bypassing the OS), and the trusted kernel is used only for control-plane operations such as permission checking. However, prior approaches fail to deliver several important file-system properties. First, using untrusted user-level libraries to maintain file system metadata shared across multiple applications can seriously compromise file-system integrity and crash consistency. Second, unlike user-level networking [51], in file systems, data-plane operations (e.g., read or write to a file) are closely intertwined with control-plane operations (e.g., block allocation); bypassing the OS during data-plane operations can compromise the security guarantees of a file system. Third, most of these approaches require OS support when sharing data across applications even for data-plane operations.

To address these limitations, and realize a true user-level direct-access file system, we propose **DevFS**, a device-level file system inside the storage hardware. The DevFS design uses the compute capability and device-level RAM to provide applications with a high-performance direct-access file system that does not compromise integrity, concurrency, crash consistency, or security. With DevFS, applications use a traditional POSIX interface without trapping into the OS for control-plane and data-plane operations. In addition to providing direct storage access, a file system inside the storage hardware provides direct visibility to hardware features such as device-level capacitance and support for processing data from multiple I/O queues. With capacitance, DevFS can safely commit data even after a system crash and also reduce file system overhead for supporting crash consistency. With knowledge of multiple I/O queues, DevFS can increase file system concurrency by providing each file with its own I/O queue and journal.

A file system inside device hardware also introduces new challenges. First, even modern SSDs have limited RAM capacity due to cost ($/GB) and power constraints. In DevFS, we address this dilemma by introducing *reverse caching,* an approach that aggressively moves inactive file system data structures off the device to the host memory. Second, a file system inside a device is a separate runtime and lacks visibility to OS state (such as process credentials) required for secured file access. To overcome this limitation, we extend the OS to coordinate with DevFS: the OS performs down-calls and shares process-level credentials without impacting direct storage access for applications.

To the best of our knowledge, DevFS is the first design to explore the benefits and implications of a file system inside the device to provide direct user-level access to applications. Due to a lack of real hardware, we implement and emulate DevFS at the device-driver level. Evaluation of benchmarks on the emulated DevFS prototype with direct storage access shows more than 2x higher write and 1.6x higher read throughput as compared to a kernel-level file system. DevFS memory-reduction techniques reduce file system memory usage by up to 5x. Evaluation of a real-world application, Snappy compression [11], shows 22% higher throughput.

In Section 2, we first categorize file systems, and then discuss the limitations of state-of-the-art user-level file systems. In Section 3, we make a case for a device-level file system. In Section 4, we detail the DevFS design and implementation, followed by experimental evaluations in Section 5. In Section 6, we describe the related literature, and finally present our conclusions in Section 7.

2 Motivation

Advancements in storage hardware performance have motivated the need to bypass the OS stack and provide applications with direct access to storage. We first discuss hardware and software trends, followed by a brief history of user-level file systems and their limitations.

2.1 H/W and S/W for User-Level Access

Prior work has explored user-level access for PCI-based solid state drives (SSD) and nonvolatile memory technologies.

Solid-state drives. Solid-state drives (SSD) have become the de facto storage device for consumer electronics as well as enterprise computing. As SSDs have evolved, their bandwidth has significantly increased along with a reduction in access latencies [6, 57]. To address system-to-device interface bottlenecks, modern SSDs have switched to a PCIe-based interface that can support up to 8-16 GB/s maximum throughput and 20-50 μs access latencies. Further, these modern devices use a large pool of I/O queues to which software can concurrently submit requests for higher parallelism.

With advancements in SSD hardware performance, bottlenecks have shifted to software. To reduce software overheads on the data path and exploit device-level parallelism, new standards such as NVMe [54] have been adopted. NVMe allows software to bypass device driver software and directly program device registers with simple commands for reading and writing the device [18].

Storage class memory technologies. Storage class memory (SCM), such as Intel's 3D Xpoint [1] and HP's memristors, are an emerging class of nonvolatile memory (NVM) that provide byte-addressable persistence and provide access via the memory controller. SCMs have properties that resemble DRAM more than a block device. SCMs can provide 2-4x higher capacity than DRAMs, with variable read (100-200ns) and write latency (400-800ns) latency. SCM bandwidth ranges from 8 GB/s to 20 GB/s, which is significantly faster than state-of-the-art SSDs. Importantly, read (loads) and writes (stores) to SCMs happen via the processor cache which plays a vital role in application performance.

Several software solutions that include kernel-level file systems [10, 13, 55, 56], user-level file systems [49], and object storage [17, 50] libraries have been proposed for SCM. Kernel-level file systems retain the POSIX-based block interface and focus on thinning the OS stack by replacing page cache and block layers with simple byte-level load and store operations. Alternative approaches completely bypass the kernel by using an object-based or POSIX-compatible interface over memory-mapped files [49].

2.2 File System Architectures

We broadly categorize file systems into three types: (1) kernel-level file systems, (2) hybrid user-level file systems, and (3) true user-level direct-access file systems. Figure 1 shows these categories, and how control-plane and data-plane operations are managed by each. The figure additionally shows a hybrid user-level file system with a trusted server and a Fuse-based file system.

Kernel-level traditional file systems. Kernel-level file systems act as a central entity managing data and metadata operations as well as control-plane operations [13, 28, 56]. As shown in Figure 1, kernel-level file systems preserve the integrity of metadata and provide crash consistency. Applications using kernel-level file systems trap into the OS for both data-plane and control-plane plane operations.

Hybrid user-level file systems. To allow applications to access storage hardware directly without trapping into the kernel, a class of research file systems [8, 26, 34] split the file system across user and kernel space. In this paper, we refer to these as hybrid user-level file systems. As

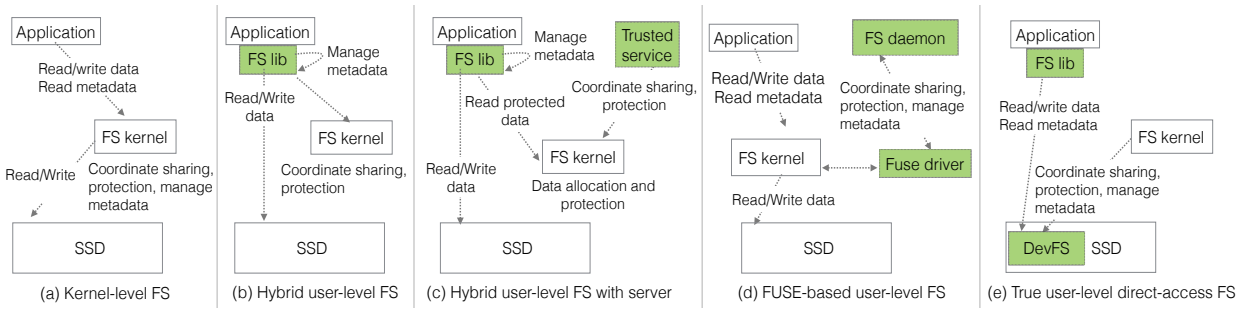


Figure 1: **File system categories.** *(a) shows a kernel-level file system, which manages control-plane and data-plane operations. (b) shows a hybrid user-level file system in which a user library manages data-plane operations. (c) shows a hybrid user-level file system with a trusted server. The server partially manages control-plane operations. (d) shows a Fuse-based file system. (e) shows a true user-level direct-access file system inside the device. The device file system fully manages the control-plane and data-plane.*

shown in Figure 1.b, the user-level file system manages all data and metadata updates without trapping into the kernel for common-case read and writes. The kernel file system is used only for control-plane operations such as permission checks, security, and data sharing across applications.

More specifically, Arrakis [34] is a hybrid user-level file system that proposes a generic user-level I/O framework for both network and storage. Arrakis aims to realize the ideas of U-Net [51] for modern hardware by virtualizing storage for each application and managing all data-plane operations at user-level, but does trap into the OS for control-plane operations. Strata [26], a hybrid user-level file system designed to combine ultra-fast NVM with high-capacity SSD and hard disk, uses NVM as a memory-mapped user-space log and writes application's data-plane operations to a log. A background thread uses a kernel-level file system to digest the logs to SSD or hard disk. For sharing files across processes, Strata traps into the kernel-level file system, which coordinates concurrent data and metadata updates.

Moneta-D [8] is a hybrid user-level file system that customizes SSDs to provide direct-access for data-plane operations. Moneta-D virtualizes an I/O interface (I/O channel) instead of storage to provide isolation and concurrency. Metadata operations are split between user-level and kernel-level. Operations such as file creation and size extension happen inside the kernel. Moneta-D enforces permission checks for data-plane operations with a user-level driver that reads a file's permission and stores them in hardware registers; during an I/O operation, the driver compares the hardware register values with process credentials.

Finally, TxDev [37] proposes a transactional flash system in which each process encapsulates its updates into a transaction request, and the flash device serializes and atomically commits the transactions. While TxDev can reduce the overheads of transactions in either user-level or kernel-level file systems, the resulting system has the same structural advantages and disadvantages of other hybrid user-level file systems.

Hybrid file systems with trusted server. Another class of hybrid file systems, such as Aerie [49], aims to reduce kernel overheads for control-plane operations by using a trusted user-level third-party server similar to a microkernel design [30] (see Figure 1.c). The trusted server runs in a separate address space and facilitates control-plane operations such as permission checking and data sharing; the server also interacts with the OS for other privileged operations.

Fuse-based user-level file systems. Another class of user-level file systems widely known as Fuse [38, 47], are mainly used for customizing and extending the in-kernel file system. As shown in Figure 1.d, in Fuse, the file system is split across a kernel driver and a user-level daemon. All I/O operations trap into the kernel, and the kernel driver simply queues I/O requests for the custom user-level file system daemon to process requests and return control to the application via the driver; as a result, Fuse file systems add an extra kernel trap for all I/O operations. Because we focus on direct-access storage solutions, we do not study Fuse in rest of this paper.

True direct-access user-level file system. In this paper, we propose DevFS, a true user-level direct-access file system as shown in Figure 1.e. DevFS pushes the file system into the device, thus allowing user-level libraries and applications to access storage without trapping into the OS for both control-plane and data-plane operations.

2.3 Challenges

Current state-of-the-art hybrid user-level file systems fail to satisfy three important properties – integrity, crash consistency, and permission enforcement – without trading away direct storage access. We discuss the challenges in satisfying these properties while providing direct access next.

2.3.1 File System Integrity

Maintaining file system integrity is critical for correct behavior of a file system. In traditional file systems, only the trusted kernel manages and updates both in-memory and persistent metadata. However, satisfying file system integrity is hard with a hybrid user-level file system for the following reasons.

Single process. In hybrid user-level file systems such as Arrakis and Moneta-D, each application uses an instance of a file system library that manages both data and metadata. Consider an example of appending a block to a file: the library must allocate a free block, update the bitmap, and update the inode inside a transaction. A buggy or malicious application sharing the address space with the file system library can easily bypass or violate the transaction and incorrectly update the metadata; as a result, the integrity of the file system is compromised. An alternative approach is to use a trusted user-level server as in Aerie [49]. However, because applications and the user-level server run in different address spaces, applications must context-switch even for data-plane operations, thus reducing the benefits of direct storage access [58]. The metadata integrity problem cannot be solved by using TxDev (a transactional flash) in a hybrid user-level file system because TxDev cannot verify the contents of transactions composed by an untrusted user-level library.

Concurrent access and sharing. Maintaining integrity with hybrid user-level file systems is more challenging when applications concurrently access the file system or share data. Updates to in-memory and on-disk metadata must be serialized and ordered across all library instances with some form of shared user-level locking and transactions across libraries. However, a malicious or buggy application can easily bypass the lock or the transaction to update metadata or data, which can lead to an incorrect file system state [25]. Prior systems such as Arrakis [34] and Strata [26] sidestep this problem by trapping into the OS for concurrent file-system access (common-case) and concurrent file access (rare). In contrast, approaches such as Aerie suffer from the context-switch problem.

2.3.2 Crash Consistency

A system can crash or lose power before all in-memory metadata is persisted to storage, resulting in arbitrary file system state such as a persisted inode without its pointed-to data [4, 35, 36]. To provide crash consistency, kernel file systems carefully orchestrate the order of metadata and data updates. For example, in an update transaction, data blocks are first flushed to a journal, followed by the metadata blocks, and finally, a transaction commit record is written to the journal; at some point, the log updates are checkpointed to the original data and metadata locations to free space in the log.

For user-level file systems, every application's untrusted library instance must provide crash consistency, which is challenging for the following reasons. First, if even a single library or application violates the ordering protocol, the file system cannot recover to a consistent state after a crash. Second, with concurrent file system access, transactions across libraries must be ordered; as discussed earlier, serializing updates with user-level locking is ineffective and can easily violate crash consistency guarantees. While a trusted third-party server can enforce ordering, applications suffer from context switches and thus do not achieve the goal of direct access.

2.3.3 Permission Enforcement

Enforcing permission checks for both the control-plane and data-plane is critical for file system security. In a kernel-level file system, when a process requests an I/O operation, the file system uses OS-level process credentials and compares it with the corresponding file (inode) permission. Hybrid user-level file systems [34] use the trusted OS for permission checks only for control-plane operation, and bypass the checks for common-case data-plane operations. Avoiding permission checks for data-plane operations violates security guarantees, specifically when multiple applications share a file system.

3 The Case For DevFS

In the pursuit of providing direct storage access to user-level applications, prior hybrid approaches fail to satisfy one or more fundamental properties of a file system. To address the limitations, and design a true direct-access file system, we propose DevFS, a design that moves the file system inside the device. Applications can directly access DevFS using a standard POSIX interface. DevFS satisfies file system integrity, concurrency, crash consistency, and security guarantees of a kernel-level file system. DevFS also supports a traditional file-system hierarchy such as files and directories, and their related functionality instead of primitive read and write operations. DevFS maintains file system integrity and crash consistency because it is trusted code that acts as a central entity. With minimal support and coordination with the OS, DevFS also enforces permission checks for common-case data-plane operations without requiring applications to trap into the kernel.

3.1 DevFS Advantages And Limitations

Moving a file system inside the device provides numerous benefits but also introduces new limitations.

Benefits. An OS-level file system generally views storage as a black box and lacks direct control over many hardware components, such as device memory, I/O queues, power-loss-protection capacitors, and the file

translation layer (FTL). This lack of control results in a number of limitations.

First, even though storage controllers often contain multiple CPUs that can concurrently process requests from multiple I/O queues [20], a host-based user-level or kernel-level file system cannot control how the device CPUs are utilized, the order in which they process request from queues, or the mapping of queues to elements such as files. However, a device-level file system can redesign file system data structures to exploit hardware-level concurrency for higher performance.

Second, some current devices contain capacitors that can hold power until the device CPUs safely flush all device-memory state to persistent storage in case of an untimely crash [22, 44]. Since software file systems cannot directly use these capacitors, they always use high-overhead journaling or copy-on-write crash consistency techniques. In contrast, a device-level file system can ensure key data structures are flushed if a failure occurs.

Finally, in SSDs and storage class memory technologies, the FTL [21] performs block allocation, logical-to-physical block translation, garbage collection, and wear-leveling, but a software file system must duplicate many of these tasks since it lacks visibility of the FTL. We believe that DevFS provides an opportunity to integrate file system and FTL functionality, but we do not yet explore this idea, leaving it to future work.

Limitations. Moving the file system into the storage device introduces both hardware and software limitations. First, device-level RAM is limited by cost and power consumption; currently device RAM is used mainly by the FTL [16] and thus the amount is proportional to the size of the logical-to-physical block mapping table (e.g., a 512 GB SSD requires a 2 GB RAM). A device-level file system will substantially increase memory footprint and therefore must strive to reduce its memory usage. Second, the number of CPUs inside a storage device can be limited and slower compared to host CPUs. While the lower CPU count impacts I/O parallelism and throughput, the slower CPUs reduce instructions per cycle (IPC) and thus increase I/O latency. Finally, implementing OS utilities and features, such as deduplication, incremental backup, and virus scans, can be challenging. We discuss these limitations and possible solutions in more detail in § 4.7.

Regarding software limitations, a device-level file system runs in a separate environment from the OS and hence cannot rely on the OS for certain functionality or information. In particular, a device-level file system must manage its own memory and must provide a mechanism to access process credentials from the OS.

3.2 Design Principles

To exploit the benefits and address the limitations of a device-level file system, we formulate the following DevFS design principles.

Principle 1: Disentangle file system data structures to embrace hardware-level parallelism. To utilize the hardware-level concurrency of multiple CPU controllers and thousands of I/O queues from which a device can process I/O requests, DevFS maps each fundamental data unit (i.e., a file) to an independent hardware resource. Each file has its own I/O queue and in-memory journal which enables concurrent I/O across different files.

Principle 2: Guarantee file system integrity without compromising direct user-level access. To maintain integrity, the DevFS inside the device acts as a trusted central entity and updates file system metadata. To further maintain integrity when multiple process share data, DevFS shares per-file structures across applications and serializes updates to these structures.

Principle 3: Simplify crash consistency with storage hardware capacitance. Traditional OS-level file systems rely on expensive journaling or log-structured (i.e., copy-on-write) mechanisms to provide crash consistency. While journaling suffers from "double write" [35] costs, log-structured file systems suffer from high garbage-collection overheads [5]. In contrast, DevFS exploits the power-loss-protection capacitors in the hardware to safely update data and metadata in-place without compromising crash consistency. DevFS thus avoids these update-related overheads.

Principle 4: Reduce the device memory footprint of the file system. Unlike a kernel-level file system, DevFS cannot use copious amounts of RAM for its data and metadata structures. To reduce memory usage, in DevFS, only in-memory data structures (inodes, dentries, per-file structures) of active files are kept in device memory, spilling inactive data structures to host memory.

Principle 5: Enable minimal OS-level state sharing with DevFS. DevFS is a separate runtime and implements its own memory management. Concerning state sharing, because DevFS does not have information about processes, we extend the OS to share process credentials with DevFS. The credentials are used by DevFS for permission checks across control-plane and data-plane operations without forcing applications to trap into the kernel.

4 Design

DevFS provides direct user-level access to storage without trapping into the OS for most of its control-plane and data-plane operations. DevFS does not compromise basic file system abstractions (such as files, directories) and properties such as integrity, concurrency, consistency, and security guarantees. We discuss how DevFS realizes

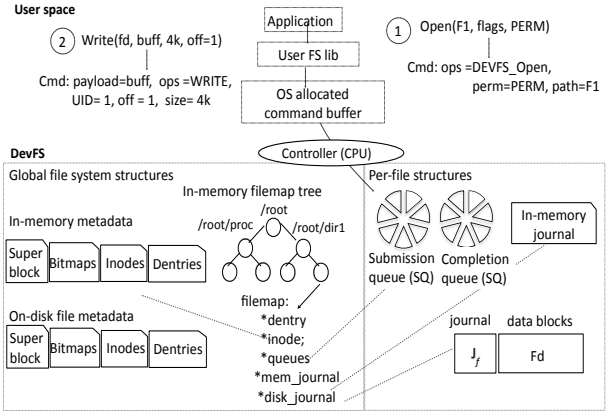


Figure 2: **DevFS high-level design.** *The file system data structure is partitioned into global and per-file structures. The per-file structures are created during file setup. DevFS metadata structures are similar to other kernel-level file system.*

the design principles discussed earlier.

4.1 Disentangling File System Structures

To exploit hardware-level concurrency, DevFS provides each file with a separate I/O queue and journal. DevFS is compatible with traditional POSIX I/O interface.

Global and per-file structures. In DevFS, the file system data structures are divided into global and per-file structures as shown in Figure 2. The global data structures manage state of an entire file system, including metadata such as the superblock, inodes, and bitmaps. The per-file structures enable concurrency: given that modern controllers contain up to four CPUs [41], and this amount is expected to increase [19], DevFS attempts to utilize multiple CPUs. In contrast to prior approaches such as Moneta-D that provide each application with its own I/O channel, DevFS provides a per-file I/O queue and journal. DevFS also maintains an in-memory filemap structure for each file. The filemap structure is created during file creation (or during file open if it is not available in device memory) and is maintained in a red-black tree as shown in the figure. Processes sharing a file also share a filemap structure which serializes access across the per-file I/O queue and the journal.

Most data structures of DevFS are similar to a kernel-level file system. Hence, we reuse and extend in-memory and on-disk data structures from the state-of-the-art persistent memory file system (PMFS) [13]. We use PMFS because it provides direct-access to storage bypassing the file system page cache. Specifically, the DevFS superblock contains global information of a file system, each inode contains per-file metadata and a reference to per-file memory and disk journal, and finally, directory entries (dentries) are maintained in a radix-tree indexed by hash values of file path names.

File system interface. Unlike prior approaches that expose the storage device as a block device for direct access [34], DevFS supports the POSIX I/O interface and abstractions such as files, directories, etc. Similar to modern NVMe-based devices with direct-access capability, DevFS uses command-based programming. To support POSIX compatibility for applications, a user-level library intercepts I/O calls from applications and converts the I/O calls to DevFS commands. On receipt, the DevFS controller (device CPU) uses the request's file descriptor to move the request to a per-file I/O queue for processing and writing to storage.

4.2 Providing File System Integrity

To maintain integrity, file system metadata is always updated by the trusted DevFS. In contrast to hybrid user-level file systems that allow untrusted user-level libraries to update metadata [34], in DevFS, there is no concern about the legitimacy of metadata content (beyond that caused by bugs in the file system).

When a command is added to a per-file I/O queue, DevFS creates a corresponding metadata log record (e.g., for a file append command, the bitmap and inode block), and adds the log record to a per-file in-memory journal using a transaction. When DevFS commits updates from an in-memory I/O queue to storage, it first writes the data followed by the metadata. Updates to global data structures (such as bitmaps) are serialized using locks.

DevFS supports file sharing across processes without trapping into the kernel. Because each file has separate in-memory structures (i.e., an I/O queue and journal), one approach would be to use separate per-file structures for each instance of an open file and synchronize updates across structures; however, synchronization costs and device-memory usage would increase linearly with the number of processes sharing a file. Hence, DevFS shares in-memory structures across processes and serializes updates using a per-file filemap lock; to order updates, DevFS tags each command with a time-stamp counter (TSC). Applications requiring strict data ordering for shared files could implement custom user-level synchronization at application-level.

4.3 Simplifying Crash Consistency

DevFS avoids logging to persistent storage by using device capacitors that can hold power until the device controller can safely flush data and metadata to storage. Traditional kernel-level file systems use either journaling or a copy-on-write techniques, such as log-structured updates, to provide crash consistency; the benefits and implications of these designs are well documented [5, 40]. Journaling commits data and metadata updates to a persistent log before committing to the original data and metadata location; as a result, journaling suffers from the "double write" problem [40, 56]. The log-structured design avoids double writes by treating an entire file system

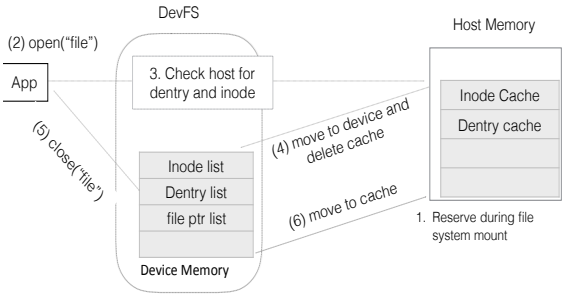


Figure 3: **DevFS reverse caching design.** *DevFS keeps only active and essential file system structures in device memory, and reverse caches others to host memory.*

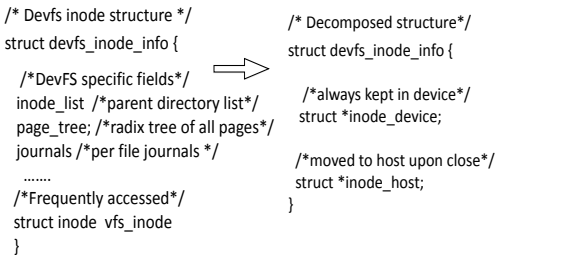


Figure 4: **Decomposing large structures.** *Large static in-memory inode is decomposed to a dynamically allocatable device and host structure. The host structure is reverse cached.*

as a log, appending data and metadata blocks; however, a log-structured design suffers from high garbage collection costs [40]. DevFS uses device-level capacitance to avoid both the double-write and garbage-collection problems.

Modern enterprise SSDs provide power-loss-protection capacitors inside device hardware that can hold power until controllers can safely flush contents of device-level DRAM [22, 44]. In existing systems, the device DRAM primarily contains the FTL's logical-to-physical block translation table, block error correction (ECC) flags, and in-flight data yet to be flushed to storage. Since DevFS runs inside the device, it uses device-level DRAM for all file system data structures.

Although the goal of hardware capacitance is to safely flush device in-memory contents to storage, flushing larger amounts of memory would require a more expensive capacitor; in addition, not all DevFS state needs to be made persistent. To minimize the memory state that must be flushed, DevFS leverages its per-file in-memory journals, as shown in Figure 2. As described previously, after an I/O command is added to a device queue, DevFS writes the command's metadata to a per-file in-memory journal. If a power failure or crash occurs, the device capacitors can hold power for controllers to safely commit in-memory I/O queues and journals to storage, thus avoiding journal writes to storage.

4.4 Minimizing Memory Footprint

We next discuss how DevFS manages device memory followed by three memory reduction techniques. The techniques include on-demand allocation, reverse caching, and a method to decompose inode structures.

DevFS uses its own memory allocator. Unlike the complex-but-generic Linux slab allocator [15], the DevFS allocator is simple and customized to manage only DevFS data structures. In addition to device memory, DevFS reserves and manages a DMA-able region in the host for reverse caching.

In DevFS, there are four types of data structures that dominate memory usage: in-memory inodes, dentries, file pointers, and the DevFS-specific per-file filemap structure. Examining the data structures in detail, we see that each inode, dentry, file pointer, and filemap consume 840 bytes, 192 bytes, 256 bytes, and 156 bytes respectively. Since inodes are responsible for the most memory usage, we examine them further. We find that 593 bytes (70.5%) of the inode structure are used by generic fields that are frequently updated during file operations; referred to as the VFS inode in other file systems, this includes the inode number, a pointer to its data blocks, permissions, access times, locks, and a reference to the corresponding dentry. The remaining 247 bytes (29.5%) of the inode are used by DevFS-specific fields, which include a reference to in-memory and on-disk journals, the dentry, the per-file structure, other list pointers, and per-file I/O queues. To reduce the device memory usage, we propose the following techniques.

On-demand memory allocation. In a naive DevFS design, the in-memory structures associated with a file, such as the I/O queue, in-memory journal, and filemap, are each allocated when a file is opened or created and not released until a file is deleted; however, these structures are not used until an I/O is performed. For workloads that access a large number of files, device memory consumption can be significant. To reduce memory consumption, DevFS uses on-demand allocation that delays allocation of in-memory structures until a read or write request is initiated; these structures are also aggressively released from device memory when a file is closed. Additionally, DevFS dynamically allocates the per-file I/O queue and memory journal and adjusts their sizes based on the availability of free memory.

Reverse caching metadata structures. In traditional OS-level file systems, the memory used by in-memory metadata structures such as inodes and dentries is a small fraction of the overall system memory; therefore, these structures are cached in memory even after the corresponding file is closed in order to avoid reloading the metadata from disk when the file is re-accessed. However, caching metadata in DevFS can significantly increase memory consumption. To reduce device memory usage, DevFS moves certain metadata structures such as in-memory inodes and dentries to host memory after a

file is closed. We call this **reverse caching** because metadata is moved off the device to the host memory.

Figure 3 shows the reverse caching mechanism. A DMA-able host-memory cache is created when DevFS is initialized. The size of the host cache can be configured when mounting DevFS depending on the availability of free host memory; the cache is further partitioned into inode and dentry regions. After moving an inode or dentry to host memory, all its corresponding references (e.g., the inode list of a directory) are updated to point to the host-memory cache. When a file is re-opened, the cached metadata is moved back to device memory. Directories are reverse-cached only after all files in a directory are also reverse-cached. Note that the host cache contains only inodes and dentries of inactive (closed) files, since deleted files are also released from the host cache. Furthermore, in case of an update to in-memory structures that are reverse-cached, the structures are moved to device memory and cached structures in the host memory are deleted. As a result, reverse caching does not introduce any consistency issues. Although using host memory instead of persistent storage as a cache avoids serializing and deserializing data structures, the overhead of data movement between device and host memory depends on interface bandwidth. The data movement overhead could be further reduced by using incremental (delta-based) copying techniques.

Decomposing file system structures. One problem with reverse caching for a complicated and large structure such as an inode is that some fields are accessed even after a file is closed. For example, a file's inode in the directory list is traversed for search operations or other updates to a directory. Moving these structures back and forth from host memory can incur high overheads. To avoid this movement, we decompose the inode structure into a device-inode and host-inode structure as shown in the Figure 4. The device-inode contains fields that are accessed even after a file is closed, and therefore only the host-inode structure is moved to host memory. Each host inode is approximately 593 bytes of the overall 840 bytes. Therefore, this decomposition along with reverse caching significantly reduces inode memory use.

4.5 State Sharing for Permission Check

DevFS provides security for control-plane and data-plane operations without trapping into the kernel by extending the OS to share application credentials.

In a kernel-level file system, before an I/O operation, the file system uses the credentials of a process from the OS-level process structure and compares them with permission information stored in an inode of a file or directory. However, DevFS is a separate runtime and cannot access OS-level data structures directly. To overcome this limitation, as shown in Figure 5, DevFS maintains

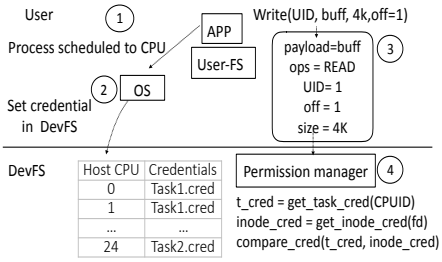


Figure 5: **DevFS permission check design.** *The OS is responsible for updating DevFS credential table with process credentials after a context-switch.*

a credential table in device memory that can be accessed and updated only by the OS, which updates the table with credential information of a new process scheduled on a host CPU. When an I/O request is sent from the host, the request is tagged with an ID number of the initiating CPU. We assume that CPU ID tagged with a request is unforgeable by an untrusted process; DevFS can be easily extended to support other types of unforgeable IDs. Before processing a request, DevFS performs a simple table lookup to compare credentials of a process running on the initiating CPU with the corresponding inode's permissions. Invalid requests are returned with a permission error in the request's completion flag.

We note that one intricate scenario can occur when a process is context-switched from its host CPU before DevFS can process the request. We address this scenario using the following steps: first, whenever a new process is scheduled to use a host CPU, the OS scheduler updates the credential table in DevFS with credentials of currently running process; second, a request is admitted to the device I/O queue only after a permission check. These steps allow DevFS to safely execute requests in the I/O queue even after a process is context-switched. Our future work will examine the overheads of OS downcalls to update the device-level credential table when processes are frequently context-switched across host CPUs.

4.6 Implementation and Emulation

We implement the DevFS prototype to understand the benefits and implications of a file system inside a storage device. Due to the current lack of programmable storage hardware, we implement DevFS as a driver in the Linux 4 kernel and reserve DRAM at boot time to emulate DevFS storage. We now describe our implementation of the DevFS user-level library and device-level file system.

User-level library and interface. DevFS utilizes command-based I/O, similar to modern storage hardware such as NVMe [54, 57]. The user library has three primary responsibilities: to create a command buffer in host memory, to convert the applications POSIX interface into DevFS commands and add them to the com-

mand buffer, and to ring a doorbell for DevFS to process the request. When an application is initialized, the user-level library creates a command buffer by making an ioctl call to the OS, which allocates a DMA-able memory buffer, registers the allocated buffer, and returns the virtual address of the buffer to the user-level library. Currently, DevFS does not support sharing command buffers across processes, and the buffer size is restricted by the Linux kernel allocation (*kmalloc()*) upper limit of 4 MB; these restrictions can be addressed by memory-mapping a larger region of shared memory in the kernel. The user-library adds I/O commands to the buffer and rings a doorbell (emulated with an ioctl) with the address of the command buffer from which DevFS can read I/O requests, perform permission checks, and add them to a device-level I/O queue for processing. For simplicity, our current library implementation only supports synchronous I/O operations: each command has an I/O completion flag that will be set by DevFS, and the user-library must wait until an I/O request completes. The user-library is implemented in about 2K lines of code.

DevFS file system. Because DevFS is a hardware-centric solution, DevFS uses straightforward data structures and techniques that do not substantially increase memory or CPU usage. We extend PMFS with DevFS components and structures described earlier. Regarding DevFS block management, each block in DevFS is a memory page; pages for both metadata and data are allocated from memory reserved for DevFS storage. The per-file memory journal and I/O queue size are set to a default of 4 KB but are each configurable during file system mount. The maximum number of concurrently opened files or directories is limited by the number of I/O queues and journals that can be created in DevFS memory. Finally, DevFS does not yet support memory-mapped I/O. DevFS is implemented in about 9K lines of code.

4.7 Discussion

Moving the file system inside a hardware device avoids OS interaction and allows applications to attain higher performance. However, a device-level file system also introduces CPU limitations and adds complexity in deploying new file system features.

CPU limitations. The scalability and performance of DevFS is dependant on the device-level CPU core count and their frequency. These device CPU limitations can impact (a) applications (or a system with many applications) that use several threads for frequent and non-dependant I/O operations, (b) multi-threaded applications that are I/O read-intensive or metadata lookup-intensive, and finally, (c) CPU-intensive file system features such as deduplication or compression. One possible approach to address the CPU limitation is to identify file-system operations and components that are CPU-intensive and move them to the user-level library in a manner that does not impact integrity, crash consistency, and security. However, realizing this approach would require extending DevFS to support a broader set of commands from the library in addition to application-level POSIX commands. Furthermore, we believe that DevFS's direct-access benefits could motivate hardware designers to increase CPU core count inside the storage device [19], thus alleviating the problem.

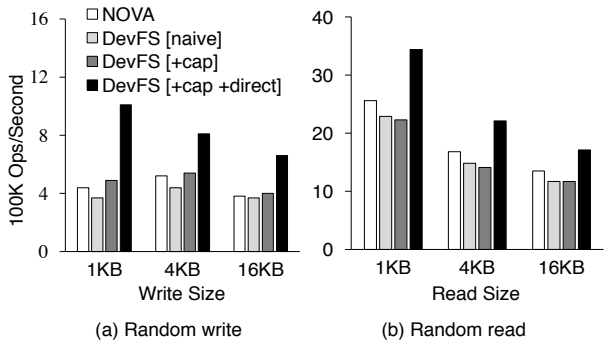
Feature support. Moving the file system into storage complicates the addition of new file system features, such as snapshots, incremental backup, deduplication, or fixing bugs; additionally, limited CPU and memory resources also add to the complexity. One approach to solving this problem is by implementing features that can be run in the background in software (OS or library), exposing the storage device as a raw block device, and using host CPU and memory. Another alternative is to support "reverse computation" by offloading file system state and computation to the host. Our future work will explore the feasibility of these approaches by extending DevFS to support snapshots, deduplication, and software RAID. Regarding bug fixes, changes to DevFS would require a firmware upgrade, which is supported by most hardware vendors today [45]. Additionally, with increasing focus on programmability of I/O hardware (e.g., NICs [8, 29]) as dictated by new standards (e.g., NVMe), support for embedding software into storage should become less challenging.

5 Evaluation

Our evaluation of DevFS aims to answer the following important questions.

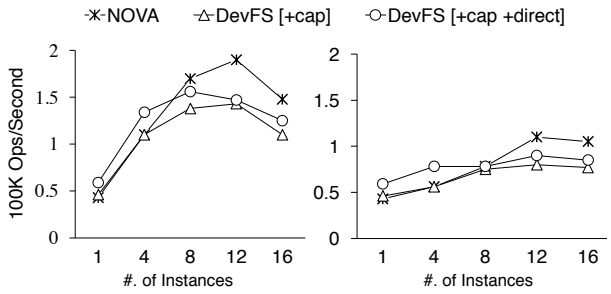
- What is the performance benefit of providing applications with direct-access to a hardware-level file system?
- Does DevFS enable different processes to simultaneously access both the same file system and the same files?
- What is the performance benefit of leveraging device capacitance to reduce the double write overhead of a traditional journal?
- How effective are DevFS's memory reduction mechanisms and how much do they impact performance?
- What is the impact of running DevFS on a slower CPU inside the device compared to the host?

We begin by describing our evaluation methodology and then we evaluate DevFS on micro-benchmarks and real-world applications.



(a) Random write (b) Random read

Figure 6: **Write and Read throughput.** *The graph shows results for Filebench random write and read micro-benchmark. X-axis varies the write size, and the file size is kept constant to 32 GB. Results show single thread performance. For DevFS, the per-file I/O queue and in-memory journal is set to 4 KB.*



(a) Without data sharing (b) With data sharing

Figure 7: **Concurrent access throughput.** *(a) shows throughput without data sharing. (b) shows throughput with data sharing. The x-axis shows the number of concurrent instances. Each instance opens ten files, appends 256 MB to each file using 4 KB writes, and then closes the files. DevFS uses up to 4 device CPUs.*

5.1 Methodology

For our experiments, we use a 40-core Intel Xeon 2.67 GHz dual socket system with 128 GB memory. DevFS reserves 60 GB of memory to emulate storage with maximum bandwidth and minimum latency. DevFS is run on 4 of the cores to emulate a storage device with 4 CPU controllers and with 2 GB of device memory, matching state-of-the-art NVMe SSDs [41, 42].

5.2 Performance

Single process performance. We begin by evaluating the benefits of direct storage access for a very simple workload of a single process accessing a single file with the Filebench workload generator [48]. We study three versions of DevFS: a naive version of DevFS with traditional journaling, DevFS with hardware capacitance support (+cap), and DevFS with capacitance support and without kernel traps (+cap +direct). We emulate DevFS without kernel traps by replicating the benchmark inside a kernel module. For comparison, we use NOVA [56], a state-of-the-art kernel-level file system for storage class memory technologies. Although NOVA does not provide direct access, it does use memory directly for storage and uses a log-structured design.

Figure 6.a shows the throughput of random writes as a function of I/O size. As expected, NOVA performs better than naive DevFS with traditional journaling. Because NOVA uses a log-structured design and writes data and metadata to storage only once, it outperforms DevFS-naive with traditional journaling since DevFS-naive writes to an in-memory journal, a per-file storage log, and the final checkpointed region. For larger I/O sizes (16 KB), the data write starts dominating the cost, thus reducing the impact of journaling on the performance.

However, DevFS with capacitance support, DevFS+cap, exploits the power-loss-protection capability and only writes metadata to the in-memory journal; both the metadata and the data can be directly committed to storage in-place without a storage log. For 1-KB writes, DevFS+cap achieves up to 27% higher throughput than the naive DevFS approach and 12% higher than NOVA. DevFS+cap outperforms NOVA because NOVA must issue additional instructions to flush its buffers, ordering writes to memory with a barrier after each write operation. Finally, by avoiding kernel traps, DevFS+cap+direct provides true direct-access to storage and improves performance by more than 2x and 1.8x for 1-KB and 4-KB writes respectively.

Figure 6.b shows random read throughput. NOVA provides higher throughput than both the DevFS-naive and DevFS+cap approaches because our prototype manages all 4 KB blocks of a file in a B-tree and traverses the tree for every read operation; in contrast, NOVA simply maps a file's contents and converts block offsets to physical addresses with bit-shift operations, which is much faster. Even with our current implementation, DevFS+direct outperforms all other approaches since it avoids expensive kernel traps. We believe that incorporating NOVA's block mapping technique into DevFS would further improve read performance.

Concurrent access performance. One of the advantages of DevFS over existing hybrid user-level file systems is that DevFS enables multiple competing processes to share the same file system and the same open files. To demonstrate this functionality, we begin with a workload in which processes share the same file system, but not the same files: each process opens ten files, appends 256 MB to each file using 4-KB writes, and then closes the files. In Figure 7.a, the number of processes is varied along the x-axis, where each process writes to a separate directory.

For a single process, DevFS+direct provides up to a 39% improvement over both NOVA and DevFS+cap by avoiding kernel traps. Since each file is allocated its own I/O queues and in-memory journal, the performance of DevFS scales well up to 4 instances; since we are emulating 4 storage CPUs, beyond four instances, the device

CPUs are shared across multiple instances and performance does not scale well. In contrast, NOVA is able to use all 40 host CPUs and scales better.

To demonstrate that multiple processes can simultaneously access the same files, we modify the above workload so that each instance accesses the same ten files; the results are shown in Figure 7.b. As desired for file system integrity, when multiple instances share and concurrently update the same file, DevFS serializes metadata updates and updates to the per-file I/O queue and in-memory journal. Again, scaling of DevFS is severely limited beyond 4 instances given the contention for the 4 device CPUs. In other experiments, not shown due to space limitations, we observe that increasing the number of device CPUs directly benefits DevFS scalability.

Summary. By providing direct-access to storage without trapping into the kernel, DevFS can improve write throughput by 1.5x to 2.3x and read throughput by 1.2x to 1.3x. DevFS also benefits from exploiting device capacitance to reduce journaling cost. Finally, unlike hybrid user-level file systems, DevFS supports concurrent file-system access and data sharing across processes; lower I/O throughput beyond four concurrent instances is mainly due to a limited number of device-level CPUs.

5.3 Impact of Reverse Caching

A key goal of DevFS is to reduce memory usage of the file system. We first evaluate the effectiveness of DevFS memory optimizations to reduce memory usage and then investigate the impact on performance.

5.3.1 Memory Reduction

To understand the effectiveness of DevFS memory-reduction techniques, we begin with DevFS+cap and analyze three memory reduction techniques: DevFS+cap+demand allocates each in-memory filemap on-demand and releases them after a file is closed; DevFS+cap+demand+dentry reverse caches the corresponding dentry after a file is closed; DevFS+cap+demand+dentry+inode also decomposes a file's inode into inode-device and inode-host structures and reverse caches the inode-host structure. Because we focus on memory reduction, we do not consider DevFS+direct in this experiment.

Figure 8 shows the amount of memory consumed for the four versions of DevFS on Filebench's file-create workload that opens a file, writes 16 KB, and then closes the file for 1 million files. In the baseline (DevFS+cap), three data structures dominate memory usage: the DevFS inode (840 bytes), the dentry (192 bytes), and the filemap (156 bytes). While file pointers, per-file I/O queues, and in-memory journals are released after a file is closed, the three other structures are not freed until the file is deleted.

The first memory optimization, DevFS+cap+demand, dynamically allocates the filemap when a read or write

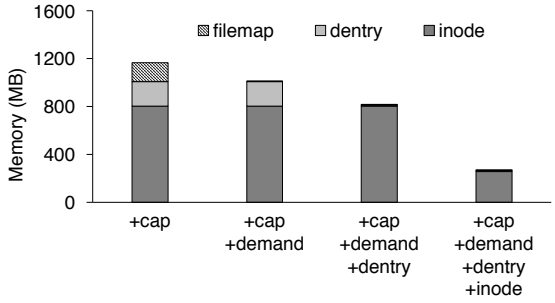


Figure 8: **DevFS memory reduction.** *+cap represents a baseline without memory reduction. Other bars show incremental memory reduction technique impact.*

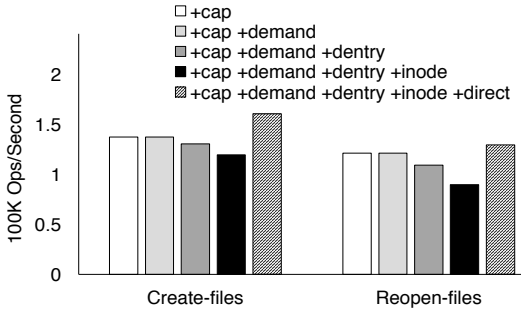


Figure 9: **Throughput impact of memory reduction.** *Reopen-files benchmark reopens closed files; as a result, structures cached in host memory are moved back to device.*

is performed and releases the filemap after closing the file; this reduces memory consumption by 156 MB (13.4%). Reverse caching of dentries, shown by DevFS+cap+demand+dentry, reduces device memory usage by 193 MB (16.6%) by moving them to the host memory; the small dentry memory usage visible in the graph represents directory dentries which are not moved to the host memory in order to provide fast directory lookup. Finally, decomposing the large inode structure into two smaller structures, inode-device (262 bytes) and inode-host (578 bytes), and reverse caching the inode-host structure reduces memory usage significantly. The three mechanisms cumulatively reduce device memory usage by up to 78% (5x) compared to the baseline. In our current implementation, we consider only these three data structures, but reverse caching could easily be extended to other file system data structures.

5.3.2 Performance Impact

The memory reduction techniques used by DevFS do have an impact on performance. To evaluate their impact on throughput, in addition to the file-create benchmark used above, we also evaluate a file-reopen workload that re-opens each of the files in the file-create benchmark immediately after it is closed. We also show the throughput for direct-access (DevFS+cap+demand+dentry+inode+direct) that avoids expensive kernel traps.

For both benchmarks, DevFS with no memory opti-

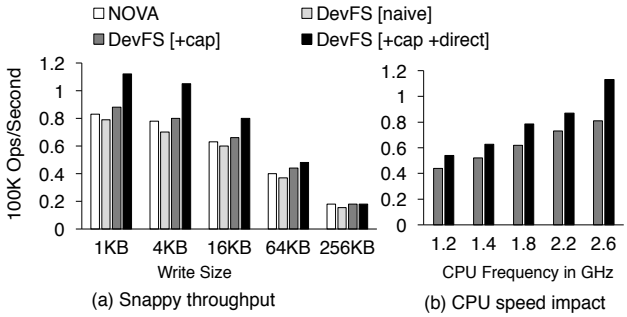


Figure 10: **Snappy compression throughput and CPU speed impact.** *Application uses 4 CPUs. Memory reduction techniques are enabled for DevFS (+cap) and DevFS (+cap +direct). For DevFS (+cap +direct), Snappy is run as a kernel module. The CPU speed is varied by scaling the frequency.*

mizations and DevFS with on-demand allocation have similar throughput because the only difference is exactly when the filemap is allocated. However, the reverse caching techniques do impact throughput. For the file-create benchmark, reverse caching only the dentry (DevFS+cap+demand+dentry) reduces throughput by 5%, while also reverse caching the inode (DevFS+cap+demand+dentry+inode) by 13%. Performance degradation occurs because reverse caching involves significant work: allocating memory in the host DRAM, copying structures to host memory, updating the parent list with the new memory address, and later releasing the device memory. The performance of reverse caching inodes is worse than that of dentries, due to their relative sizes (578 bytes vs 196 bytes). While the direct-access approach has similar overheads, by avoiding kernel traps for file open, write, and close, and it provides higher performance compared to all other approaches.

With the file-reopen benchmark, reverse caching moves the corresponding inodes and dentries back to device memory, causing a throughput drop of 26%. Our results for the file-reopen benchmark can be considered worst-case behavior since most real-world applications spend more time performing I/O before closing a file. Our current mechanism performs aggressive reverse caching, but could easily be extended to slightly delay reverse caching based on the availability of free memory in the device.

Summary. DevFS memory-reduction techniques can reduce device DRAM usage by up to 5x. Although worst-case benchmarks do suffer some performance impact with these techniques, we believe memory reduction is essential for device-level file systems and that DevFS will obtain both memory reduction and high performance for realistic workloads.

5.4 Snappy File Compression

To understand the performance impact on a real-world application, we use Snappy [11] compression. Snappy is widely used as a data compression engine for several applications including MapReduce, RocksDB, MongoDB, and Google Chrome. Snappy reads a file, performs compression, and writes the output to a file; for durability, we add an `fsync()` after writing the output. Snappy optimizes throughput and is both CPU- and I/O-intensive; for small files, the I/O time dominates the computation time. Snappy can be used at both user-level and kernel-level [23] which helps us to understand the impact of direct access. For the workload, we use four application threads, 16 GB of image files from OpenImage repository [24], and vary the size of files from 1 KB to 256 KB.

Comparing the performance of NOVA, DevFS-naive, DevFS+cap, and DevFS+direct, we see the same trends for the Snappy workload as we did for the previous micro-benchmarks. As shown in Figure 10.a, NOVA performs better than DevFS-naive due to DevFS-naive's journaling cost, while DevFS+cap removes this overhead. Because DevFS+direct avoids trapping into the kernel when reading and writing across all application threads, it provides up to 22% higher throughput than DevFS-cap for 4-KB files; as the file size increases, the benefit of DevFS+direct is reduced since compression costs dominate runtime.

Device CPU Impact. One of the challenges of DevFS is that it is restricted to the CPUs on the storage device, and these device CPUs may be slower than those on the host. To quantify this performance impact, we run the Snappy workload as we vary the speed of the "device" CPUs, keeping the "host" CPUs at their original speed of 2.6 GHz [27]; the threads performing compression always run on the fast "host" CPUs. Figure 10.b shows the performance impact for 4-KB file compression for two versions of DevFS; we choose 4-KB files since it stresses DevFS performance more than with larger files (which instead stress CPU performance). As expected, DevFS-direct consistently performs better than DevFS-cap. More importantly, we do see that reducing device CPU frequency does have a significant impact on performance (e.g., reducing device CPU frequency from 2.6 GHz to 1.4 GHz reduces throughput by 66%). However, comparing across graphs, we see that even with a 1.8 GHz device CPU, the performance of DevFS-direct is similar to that of NOVA running on all high-speed host CPUs. For workloads that are more CPU intensive, the impact of slower device CPUs on DevFS performance is smaller (not shown due to space constraints).

Summary. DevFS-direct provides considerable performance improvement even for applications that are both CPU and I/O-intensive. We observe that although slower device CPUs do impact performance of DevFS, DevFS can still outperform other approaches.

6 Related Work

Significant prior work has focused on providing direct-access to storage, moving computation to storage, or programmability of SSDs.

Direct-access storage. Several hybrid user-level file system implementations, such as Intel's SPDK [18], Light NVM [6], and Micron's User Space NVME [33] provide direct-access to storage by exposing them as a raw block device and exporting a userspace device driver for block access. Light NVM goes one step further to enable I/O-intensive applications to implement their own FTL. However, these approaches do not support traditional file-system abstractions and instead expose storage as a raw block device; they do not support fundamental properties of a file system such as integrity, concurrency, crash consistency, or security.

Computation inside storage. Providing compute capability inside storage for performing batch tasks have been explored for past four decades. Systems such as CASSM [46] and RARES [31] have proposed adding several processors to a disk for performing computation inside storage. ActiveStorage [2, 39] uses one CPU inside a hard disk for performing database scans and search operations, whereas Smart-SSD [12] is designed for query processing inside SSDs. Architectures such as BlueDBM [19] have shown the benefits of scaling compute and DRAM inside flash memory for running "big data" applications. DevFS also uses device-level RAM and compute capability; however, DevFS uses these resources for running a high-performance file system that applications can use.

Programmability. Willow [43] develops a system to improve SSD programmability. Willow's I/O component is offloaded to an SSD to bypass the OS and perform direct read and write operations. However, without a centralized file system, Willow also has the same general structural advantages and disadvantages of hybrid user-level file systems.

7 Conclusion

In this paper, we address the limitations of prior hybrid user-level file systems by presenting DevFS, an approach that pushes file system functionality down into device hardware. DevFS is a trusted file system inside the device that preserves metadata integrity and concurrency by exploiting hardware-level parallelism, leverages hardware power-loss control to provide low-overhead crash consistency, and coordinates with the OS to satisfy security guarantees. We address the hardware limitations of low device RAM capacity by proposing three memory reduction techniques (on-demand allocation, reverse caching, and decomposing data structures) to reduce file system memory usage by 5x(at the cost of a small performance reduction). Performance evaluation of our DevFS prototype shows more than 2x improvement in I/O throughput with direct-access to storage. We believe our DevFS prototype is a first step towards building a true direct-access file system. Several engineering challenges, such as realizing DevFS in real hardware, supporting RAID, and integrating DevFS with the FTL, remain as future work.

Acknowledgements

We thank the anonymous reviewers and Ed Nightingale (our shepherd) for their insightful comments. We thank the members of the ADSL for their valuable input. This material was supported by funding from NSF grants CNS-1421033 and CNS-1218405, and DOE grant DE-SC0014935. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active Disks: Programming Model, Algorithms and Evaluation. *SIGPLAN Not.*, 33(11):81–91, October 1998.

[3] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: A Protoype Phase Change Memory Storage Array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, Portland, OR, 2011.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[5] Valerie Aurora. Log Structured File System Issues. https://lwn.net/Articles/353411/.

[6] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-channel SSD Subsystem. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, Santa clara, CA, USA, 2017.

[7] Eric Brewer. FAST Keynote: Disks and their Cloudy Future, 2015. https://www.usenix.org/sites/default/files/conference/protected-files/fast16_slides_brewer.pdf.

[8] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. *SIGARCH Comput. Archit. News*, 40(1), March 2012.

[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, Newport Beach, California, USA, 2011.

[10] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, Big Sky, Montana, USA, 2009.

[11] Jeff Dean, Sanjay Ghemawat, and Steinar H. Gunderson. Snappy Compression. https://github.com/google/snappy.

[12] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[13] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, Copper Mountain, Colorado, USA, 1995.

[15] Mel Gorman. Understanding the Linux Virtual Memory Manager. http://bit.ly/1n1xIhg.

[16] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 127–144, New York, NY, USA, 2017. ACM.

[17] Intel. NVM Library. https://github.com/pmem/nvml.

[18] Intel. Storage Performance Development Kit. http://www.spdk.io/.

[19] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. BlueDBM: Distributed Flash Storage for Big Data Analytics. *ACM Trans. Comput. Syst.*, 34(3):7:1–7:31, June 2016.

[20] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, 2016. USENIX Association.

[21] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. A Space-Efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[22] Kingston. Kingston power loss control. https://www.kingston.com/us/ssd/enterprise/technical_brief/tantalum_capacitors.

[23] Andi Kleen. Snappy Kernel Port. https://github.com/andikleen/snappy-c.

[24] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://github.com/openimages*, 2017.

[25] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, 2017. USENIX Association.

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

[27] Etienne Le Sueur and Gernot Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, Vancouver, BC, Canada, 2010.

[28] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, Santa Clara, CA, 2015.

[29] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, Shanghai, China, 2017.

[30] J. Liedtke. On Micro-kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, Copper Mountain, Colorado, USA, 1995.

[31] Chyuan Shiun Lin, Diane C. P. Smith, and John Miles Smith. The Design of a Rotating Associative Memory for Relational Database Applications. *ACM Trans. Database Syst.*, 1(1):53–65, March 1976.

[32] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. *SIGMETRICS Perform. Eval. Rev.*, 43(1):177–190, June 2015.

[33] Micron. Micron User Space NVMe. `https://github.com/MicronSSD/unvme/`.

[34] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Broomfield, CO, 2014.

[35] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, Santa clara, CA, USA, 2017.

[36] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Broomfield, CO, 2014.

[37] 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, San Diego, California, 2008.

[38] Aditya Rajgarhia and Ashish Gehani. Performance and Extension of User Space File Systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM.

[39] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[40] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1), February 1992.

[41] Samsung. NVMe SSD 960 Polaris Controller. `http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/NVMe_SSD_960_PRO_EVO_Brochure.pdf`.

[42] Samsung. Samsung nvme ssd 960 data sheet. `http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_960_PRO_Data_Sheet_Rev_1_1.pdf`.

[43] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Broomfield, CO, 2014.

[44] Y. Son, J. Choi, J. Jeon, C. Min, S. Kim, H. Y. Yeom, and H. Han. SSD-Assisted Backup and Recovery for Database Systems. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 285–296, April 2017.

[45] StorageReview.com. Firmware Upgrade. `http://www.storagereview.com/how_upgrade_ssd_firmware`.

[46] Stanley Y. W. Su and G. Jack Lipovski. CASSM: A Cellular System for Very Large Data Bases. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, Framingham, Massachusetts, 1975.

[47] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-space File Systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, Santa clara, CA, USA, 2017.

[48] Tarasov Vasily. Filebench. `https://github.com/filebench/filebench`.

[49] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014.

[50] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, Newport Beach, California, USA, 2011.

[51] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, Copper Mountain, Colorado, USA, 1995.

[52] Michael Wei, Matias Bjørling, Philippe Bonnet, and Steven Swanson. I/O Speculation for the Microsecond Era. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, Philadelphia, PA, 2014.

[53] Matthew Wilcox and Ross Zwisler. Linux DAX. `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`.

[54] NVM Express Workgroup. NVMExpress Specification. `://www.nvmexpress.org/resources/specifications/`.

[55] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, Seattle, Washington, 2011.

[56] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, 2016.

[57] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, Haifa, Israel, 2015.

[58] Jisoo Yang, Dave B. Minturn, and Frank Hady. When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, San Jose, CA, 2012.

FStream: Managing Flash Streams in the File System

Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty
Joo-Young Hwang, Sangyeun Cho, Daniel DG Lee, Jaeheon Jeong
Samsung Electronics Co., Ltd.

Abstract

The performance and lifespan of a solid-state drive (SSD) depend not only on the current input workload but also on its internal media fragmentation formed over time, as stale data are spread over a wide range of physical space in an SSD. The recently proposed *streams* gives a means for the host system to control how data are placed on the physical media (abstracted by a stream) and effectively reduce the media fragmentation. This work proposes *FStream*, a file system approach to taking advantage of this facility. FStream extracts streams at the file system level and avoids complex application level data mapping to streams. Experimental results show that FStream enhances the filebench performance by 5%∼35% and reduces WAF (Write Amplification Factor) by 7%∼46%. For a NoSQL database benchmark, performance is improved by up to 38% and WAF is reduced by up to 81%.

1 Introduction

Solid-state drives (SSDs) are rapidly replacing hard disk drives (HDDs) in enterprise data centers. SSDs maintain the traditional logical block device abstraction with the help from internal software, commonly known as flash translation layer (FTL). The FTL allows SSDs to substitute HDDs without complex modification in the block device interface of an OS.

Prior work has revealed, however, that this compatibility comes at a cost; when the underlying media is fragmented as a device is aged, the operational efficiency of the SSD deteriorates dramatically due to garbage collection overheads [11, 13]. More specifically, a user write I/O translates into an amplified amount of actual media writes [9], which shortens device lifetime and hampers performance. The ratio of the actual media writes to the user I/O is called *write amplification factor* (WAF).

A large body of prior work has been undertaken to address the write amplification problem and the SSD wear-out issue [3, 7]. To the same end, we focus on *how to take advantage of the multi-streamed SSD mechanism* [8]. This mechanism opens up a way to dictate data placement on an SSD's underlying physical media, abstracted by *streams*. In principle, if the host system perfectly maps data having the same lifetime to the same streams, an SSD's write amplification becomes one, completely eliminating the media fragmentation problem.

Prior works have revealed two strategies to leverage streams. The first strategy would map application data to disparate streams based on an understanding of the expected lifetime of those data. For example, files in different levels of a log-structured merge tree could be assigned to a separate stream. Case studies show that this strategy works well for NoSQL databases like Cassandra and RocksDB [8, 14]. Unfortunately, this application-level customization strategy requires that a system designer understand her target application's internal working fairly well, remaining a challenge to the designer. The other strategy aimed to "automate" the process of mapping write I/O operations to an SSD stream with no application changes. For example, the recently proposed AutoStream scheme assigns a stream to each write request based on estimated lifetime from past LBA access patterns [15]. However, this scheme has not been proven to work well under complex workload scenarios, particularly when the workload changes dynamically. Moreover, LBA based pattern detection is not practical when file data are updated in an out-of-place manner, as in copy-on-write and log-structured file systems. The above sketched strategies capture the two extremes in the design space—*application level customization* vs. *block level full automation*.

In this work, we take another strategy, where *we separate streams at the file system layer*. Our approach is motivated by the observation that file system metadata and journal data are short-lived and are good targets for separation from user data. Naturally, the primary component of our scheme, when applied to a journaling file sys-

tem like ext4 and xfs, is to allocate a separate stream for metadata and journal data, respectively. As a corollary component of our scheme, we also propose to separate databases redo/undo log file as a distinct stream at the file system layer. We implement our scheme, *FStream*, in Linux ext4 and xfs file systems and perform experiments using a variety of workloads and a stream-capable NVMe (NVM Express) SSD. Our experiments show that FStream robustly achieves a near-optimal WAF (close to 1) across the workloads we examined. We make the following contributions in this work.

- We provide an automated multi-streaming of different types of file system generated data with respect to their lifetime;
- We enhance the existing journaling file systems, ext4 and xfs, to use the multi-streamed SSDs with minimally invasive changes; and
- We achieve stream classification for application data using file system layer information.

The remainder of this paper is organized as follows. First, we describe the background of our study in Section 2, with respect to the problems of previous multi-stream schemes. Section 3 describes FStream and its implementation details. We show experimental results in Section 4 and conclude in Section 5.

2 Background

2.1 Write-amplification in SSDs

Flash memory has an inherent characteristic of *erase-before-program*. Write, also called "program" operation, happens at the granularity of a NAND page. A page cannot be rewritten unless it is "erased" first. In-place update is not possible due to this erase-before-write characteristic. Hence, overwrite is handled by placing the data in a new page, and invalidating the previous one. Erase operation is done in the unit of a NAND block, which is a collection of multiple NAND pages. Before erasing a NAND block, all its valid pages need to be copied out elsewhere; this is done in a process called *garbage-collection* (GC). These valid page movements cause additional writes that consume bandwidth, thereby leading to performance degradation and fluctuation. These additional writes also reduce endurance as program-erase cycles are limited for NAND blocks. One way to measure GC overheads is *write amplification factor* (WAF), which is described as the ratio of writes performed on flash memory to writes requested from the host system.

$$WAF = \frac{Amount\ of\ writes\ committed\ to\ flash}{Amount\ of\ writes\ that\ arrived\ from\ the\ host}$$

WAF may soar more often than not as an SSD experiences aging [8].

2.2 Multi-streamed SSD

The multi-streamed SSD [8] endeavors to keep WAF in check by focusing on the placement of data. It allows the host to pass a hint (stream ID) along with write requests. The stream ID provides a means to convey hints on lifetime of data that are getting written [5]. The multi-streamed SSD groups data having the same stream ID to be stored to the same NAND block. By avoiding mixing data of different lifetime, fragmentation is reduced inside NAND blocks. Figure 1 shows an example of how the data placement using multi-stream could reduce media fragmentation.

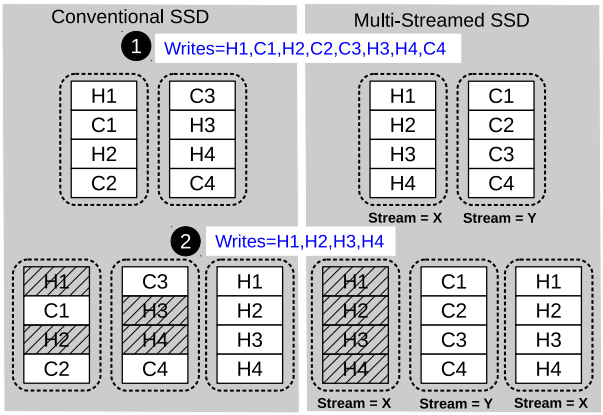


Figure 1: *Data placement comparison for two write sequences.* Here H=Hot, C=Cold, and we assume there are four pages per flash block.

Multi-stream is now a part of T10 (SCSI) standard, and under discussion in NVMe (NVM Express) working group. NVMe 1.3 specification introduces support for multi-stream in the form of "directives" [1]. An NVMe write command has the provision to carry a stream ID.

2.3 Leveraging Streams

While the multi-streamed SSD provides the facility of segregating data into streams, its benefit largely depends upon how well the streams are leveraged by the host. Identifying what should and should not go into the same stream is of cardinal importance for maximum benefit. Previous work [8, 14] shows benefits of application-assigned streams. This approach has the benefit of determining data lifetime accurately, but it involves modifying the source code of the target application, leading to increased deployment effort. Also when multiple applications try to assign streams, a centralized stream assignment is required to avoid conflicts. Instead of direct assignment of stream IDs, recently Linux (from 4.13 kernel) supports `fcntl()` interface to send data lifetime hints to file systems to exploit the multi-streamed SSDs [2, 6]. AutoStream [15] takes stream management to the NVMe device driver layer. It monitors requests

from file systems and estimates data lifetime. However, only limited information (e.g., request size, block addresses) is available in the driver layer, and even worse, the address-based algorithm may be ineffective under a copy-on-write file system.

Our approach, *FStream*, implements stream management intelligence at the file system layer. File systems have readily the information about file system generated data such as metadata and journal. To detect lifetime of user data, we take a simple yet efficient method which uses file's name or extension. For the sake of brevity, we do not cover the estimation of user data lifetime in detail at the file system layer.

3 FStream

We start by highlighting the motivation behind employing multi-stream in file systems. This is followed by overview of ext4 and xfs on-disk layout and journaling methods. Then we delve into the details of Ext4Stream and XFStream, which are stream-aware variants of ext4 and xfs, respectively.

3.1 Motivation

Applications can have better knowledge about the lifetime and update frequency of data that they write than file systems do. However, applications do not know about the lifetime and update frequency of file system metadata. The file system metadata usually have different update frequencies than applications' data, and are often influenced by on-disk layout and write policy of a particular file system. Typically file systems keep data and metadata logically separated, but they may not remain "physically" separated on SSDs. While carrying out file operations, metadata writes may get mixed with data writes in the same NAND block or one type of metadata may get mixed with another type of metadata. File systems equipped with stream separation capability may reduce the mixing of applications' data and file system metadata, and improve WAF and performance.

3.2 Ext4 metadata and journaling

The ext4 file system divides the disk-region in multiple equal-size regions called "block groups," as shown in Figure 2. Each block group contains data and their related metadata together which helps in reducing seeks for HDDs. Ext4 introduces *flex-bg* feature, which "clubs" a series of block groups whose metadata are consolidated in the first block group. Each file/directory requires an inode, which is of size 256 bytes by default. These inodes are stored in the *inode table*. *inode bitmap* and *block bitmap* are used for allocation of inodes and data blocks, respectively. *Group descriptor* contains the location of other metadata regions (inode table, block bitmap, inode bitmap) within the block group. Another type of metadata is a directory block.

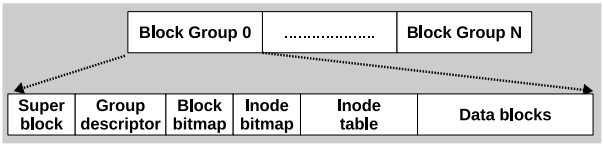


Figure 2: Ext4 on-disk layout. For simplicity, we have not shown flex-bg.

File-system consistency is achieved through write-ahead logging in journal. Journal is a special file whose blocks reside in user data area, pre-allocated at the time of file system format. Ext4 has three journaling modes; *data-writeback* (metadata journaling), *data-ordered* (metadata journaling + write data before metadata), and *data-journal* (data journaling). The default mode is data-ordered.

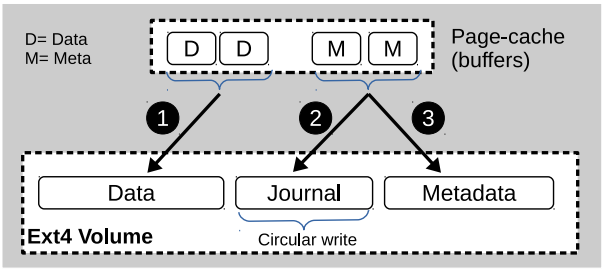


Figure 3: Ext4 *journal in ordered mode.* Ext4 writes data and journal in sequence. Metadata blocks are written to their actual home location after they are persisted to the journal.

Ext4 journals at a block granularity, i.e., even if few bytes of an inode are changed, the entire block (typically 4KiB) containing many inodes is journaled. For journaling it takes assistance from another component called journaling block device (JBD), which has its own kernel thread called jbd2. The journal area is written in a circular fashion. Figure 3 shows journaling operation in the ordered mode. During a transaction, ext4 updates metadata in in-memory buffers, and informs the jbd2 thread to commit a transaction. jbd2 maintains a timer (default 5 seconds), on expiry of which it writes modified metadata into the journal area, apart from transaction related bookkeeping data. Once changes have been made durable,the transaction is considered committed. Then, the metadata changes in memory are flushed to their original locations by write-back threads, which is called *checkpointing*.

3.3 Xfs metadata and journaling

Similar to ext4, xfs also divides the disk region into multiple equal-size regions called *allocation groups*, as shown in Figure 4. The primary objective of allocation

groups is to increase parallelism rather than disk locality, unlike EXT4 block groups. Each allocation group maintains its own superblock and other structures for free space management and inode allocation, thereby allowing parallel metadata operations. Free space management within an allocation group is done by using B+ trees. Inodes are allocated in chunks of 64. These chunks are managed in another B+ tree meant exclusively for inode allocation.

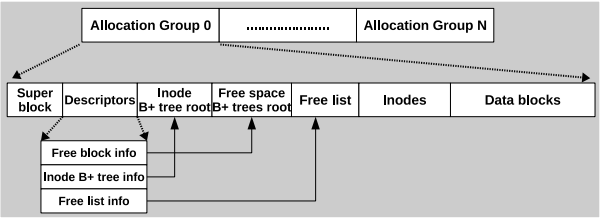


Figure 4: Xfs on-disk layout.

For transaction safety, xfs implements metadata journaling. A separate region called "log" is created during file system creation (`mkfs.xfs`). Log is written in a circular fashion as transactions are performed. Xfs maintains many log buffers (default 8) in memory, which can record the changes for multiple transactions. Default commit interval is 30 seconds. During commit, modified log buffers are written to on-disk log area. Post commit, modified metadata buffers are scheduled for flushing to their actual disk locations.

3.4 Ext4Stream: Multi-stream in ext4

Table 1 lists the streams we introduced in ext4. These streams can be enabled with the corresponding mount option listed in the table.

Mount-option	Stream
journal-stream	Separate journal writes
inode-stream	Separate inode writes
dir-stream	Separate directory blocks
misc-stream	Separate inode/block bitmap and group descriptor
fname-stream	Assign distinct stream to file(s) with specific name
extn-stream	File-extension based stream

Table 1: Streams introduced in Ext4Stream.

The **journal_stream** mount option is to separate journal writes. We added a `j_streamid` field in the `journal_s` structure. When ext4 is mounted with the `journal_stream` option, a stream ID is allocated and stored in the `j_streamid` field. jbd2 passes this stream ID when it writes dirty buffers and descriptor blocks in a journal area using `submit_bh` and related functions.

Ext4 makes use of buffer-head (`bh`) structures for various metadata buffers including inode, bitmaps, group descriptors and directory data blocks. We added a new field `streamid` in buffer-head to store a stream ID, and modified `submit_bh`. While forming an I/O from buffer-head, this field is also set in `bio`, taking stream ID information to a lower layer. Ext4Stream maintains stream IDs for different metadata regions in its superblock, and, depending on the type of metadata buffer-head, it sets `bh->streamid` accordingly.

The **inode_stream** mount option is to separate inode writes. Default inode size is 256 bytes, so single 4KiB FS block can store 16 inodes. Modification in one inode leads to writing of an entire 4KiB block. When Ext4Stream modifies inode buffer, it also sets `bh->streamid` with the stream ID meant for the inode stream.

The **dir_stream** mount option is to keep directory blocks into its own stream. When a new file or subdirectory is created inside a directory, a new directory entry needs to be added. This triggers either update of an existing data block belonging to the directory or addition of a new data block. Directory blocks are organized in a `htree`; leaf nodes contain directory entries and non-leaf nodes contain indexing information. We assign a same stream ID for both types of directory blocks.

The **misc_stream** is to keep inode/block bitmap blocks and group descriptor blocks into a stream. These regions receive updates during the creation/deletion of file/directory and when data blocks are allocated to file/directory. We group these regions into a single stream because they are of small size.

The **fname_stream** helps to put data of certain special files into distinct stream. Motivation is to use this for separating undo/redo log for SQL and NoSQL databases.

The **extn_stream** is to enable file extension based stream recognition. Data blocks of certain files, such as multimedia files, can be considered cold. Ext4Stream can parse extension of files during file creation. If it matches with some well-known extensions, file is assigned a different stream ID. This helps prevent hot or cold data blocks getting mixed with other types of data/metadata blocks.

3.5 XFStream: Multi-stream in xfs

Table 2 lists the streams we introduced to xfs. These streams can be enabled with the corresponding mount options listed in the table.

Xfs implements its own metadata buffering rather than using page cache. The `xfs_buf_t` structure is used to represent a buffer. Apart from metadata, in-memory buffers of log are implemented via `xfs_buf_t`. When the buffer is flushed, a `bio` is prepared out of `xfs_buf_t`. We

Mount-option	Stream
log_stream	Separate writes occurring in log
inode_stream	Separate inode writes
fname_stream	Assign distinct stream to file(s) with specific name

Table 2: Streams introduced in XFStream.

added a new field called `streamid` in `xfs_buf_t`, and used that to set the stream information in `bio`.

The **log_stream** enables XFStream to perform writes in the log area with its own stream ID. Each mounted xfs volume is represented by a `xfs_mount_t` structure. We added the field `log_streamid` in it, which is set when xfs is mounted with `log_stream`. This field is used to convey stream information in `xfs_buf_t` representing the log buffer.

The **inode_stream** mount option enables XFStream to separate inode writes into a stream. A new field `ino_streamid` kept in `xfs_mount_t` is set to stream ID meant for inodes. This field is used to convey stream information in `xfs_buf_t` representing the inode buffer.

Finally, the **fname_stream** enables XFStream to assign a distinct stream to file(s) with specific name(s).

4 Evaluation

Our experimental system configurations are as follows.

- System: Dell Poweredge R720 server with 32 cores and 32GB memory,
- OS: Linux kernel 4.5 with io-streamid support,
- SSD: Samsung PM963 480GB, with the allocation granularity [1] of 1.1GB,
- Benchmarks: filebench [12], YCSB (Yahoo! Cloud Serving Benchmark) 0.1.4 [4] on Cassandra 1.2.10 [10].

The SSD we used supports up to 9 streams; eight NVMe standard compliant streams and one default stream. If a write command does not specify its stream ID, it is written to the default stream.

We conduct experiments in two parts; the first part is to measure the benefit of separating file system metadata and journal. Each test starts with a fresh state involving device format and file system format. To reduce variance between the runs, we disable lazy journal and inode table initialization at the time of ext4 format. As a warming workload for filebench, we write a single file sequentially to fill 80% of logical device capacity, to ensure that 80% of the logical space stays valid throughout the test. Remaining logical space involves the actual experiment. The varmail and fileserver workloads included in the filebench are used to simulate mail server and file server workloads, respectively. The number of files in both workloads is set to 900,000; default values are used for other parameters. Each filebench workload is run for two hours with 14GB of files, performing deletion, creation, append, sync (only for varmail), and random read. Since the size of the workloads is smaller than that of RAM, vast majority of the operations that actually reach the device are likely to be write operations. In order to acquire WAF, we retrieve the number of NAND writes and host writes from FTL, and divide the former by the latter.

In the second part, we measure the benefits in data-intensive workloads by applying automatic stream assignment on certain application specific files. Previous work [8] has reported improvement by modifying Cassandra source to categorize writes into multiple streams. FStream assigns distinct stream to Cassandra *Commitlog* through fname_stream facility. Load phase involves loading 120,000,000 keys, followed by insertion of 80,000,000 keys during run phase.

[1] A multi-streamed SSD allocate and expand stream in the unit of allocation granularity

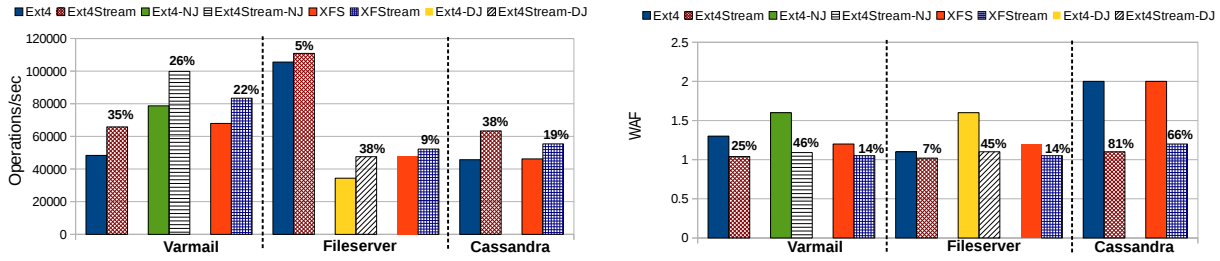
4.1 Filebench results

Category	varmail			fileserver	
	ext4	ext4 -nj	xfs	ext4	xfs
Journal	61%	-	60%	26%	16%
Inode	8%	21%	9%	16%	32%
Directory	4%	15.8%	-	3%	-
Other meta	0.2%	0.2%	-	0.2%	-
Data	26.8%	63%	31%	54.8%	52%

Table 3: Distribution of I/O types during a filebench run.

The benefit of separating metadata or journal into different streams depends on the amount of metadata write traffic and degree of mixing among various I/O types. As shown in Figure 5, Ext4Stream shows 35% performance increase and 25% WAF reduction than baseline ext4 for varmail. XFStream shows 22% performance increase and 14% WAF reduction for varmail compared to xfs. Both Ext4Stream and XFStream show more enhancements for varmail than for fileserver, because varmail is more metadata-intensive than fileserver, as shown in Table 3.

To investigate the effect of the stream separation for file systems without journaling, we disable the journal in ext4, denoted as ext4-nj. Under varmail workload, ext4-nj performs better than ext4 by 62%, which is mainly due to the removal of journal writes. Stream separation



(a) Throughput comparison

(b) WAF comparison

Figure 5: Above graphs show performance and WAF improvement percentage obtained with multi-stream variants of file-systems. Here Ext4-NJ denotes Ext4 without journal, and Ext4-DJ denotes Ext4 with data journal.

improves the performance and WAF of ext-nj by 26% and 46%, respectively.

A key observation in fileserver is that reducing metadata writes is more important for performance than reducing journal writes. xfs generates 16% more inode writes and 10% less journal writes for fileserver than ext4. Though the sum of metadata and journal writes is similar, xfs's performance is less than half of ext4's performance. The reason is that the metadata writes are random access while journal writes are sequential. Sequential writes are better for FTL's GC efficiency, and hence are good for performance and lifetime.

Another important observation comes from ext4 fileserver write distribution in Table 3 which shows 16% inode write, but only 0.2% other-meta which includes inode-bitmap as well. This is because of a jbd2 optimization. If a transaction modifies an already-dirty meta buffer, jbd2 delays its writeback by resetting its dirty timer, which is why inode/block bitmap buffer writes remain low despite large number of block/inode allocations.

As shown in Fig 5(b), ext4 WAF remains close to one during the fileserver test. However, when ext4 is operated with *data=journal* mode (shown by ext4-DJ), WAF soars above 1.5 due to continuous mixing between journal and application-data. Ext4Stream-DJ eliminates this mixing and brings WAF down back to near one.

4.2 Cassandra results

Cassandra workloads are data-intensive. Database changes are first made to an in-memory structure, called "memtable", and are written to an on-disk commitlog file. The commitlog implements the consistency for Cassandra databases as file system journal does for ext4 and xfs. It is written far more often than file system journal. By separating the commitlog from databases, done by file systems through detecting the file name of commitlog files, we observe 38% throughput improvement and 81% WAF reduction. Cassandra commitlog files are named as commitlog-* with date and time information. With fname_stream option, files with their names starting with commitlog flow into a single stream. Even if multiple instances of Cassandra run on a single SSD, commitlog files are written only to the stream assigned by fname_stream.

5 Conclusions

In this paper, we advocate an approach of applying multi-stream in the file system layer in order to address the SSD aging problem and GC overheads. Previous proposals of using multi-stream are either application level customization or block level full automation. We aimed to make a new step forward from those proposals. We separate streams at the file system level. We focused on the attributes of file system generated data, such as journal and metadata, because they are short-lived thus suitable for stream separation. We implemented an automatic separation of those file system generated data, with no need for user intervention. Not only have we provided fully automated separation of metadata and journal, but also we advise to separate the redo/undo logs used by applications to different streams. Physical data separation achieved by our stream separation scheme, FStream, helps FTL reduce GC overheads, and thereby enhance both performance and lifetime of SSDs. We applied FStream to ext4 and xfs and obtained encouraging results for various workloads that mimic real-life servers in data centers. Experimental results showed that our scheme enhances the filebench performance by 5%∼35% and reduces WAF by 7%∼46%. For a Cassandra workload, performance is improved by up to 38% and WAF is reduced by up to 81%.

Our proposal can bring sizable benefits in terms of SSD performance and lifetime. As future work, we consider applying FStream to log-structured file systems, like f2fs, and copy-on-write file systems, e.g., btrfs. We also plan to evaluate how allocation granularity and optimal write size affects the performance and endurance.

References

[1] The NVM Express 1.3 Specification. http://www.nvmexpress.org/.

[2] AXBOE, J. Add support for write life time hints, June 2017.

[3] CHIANG, M.-L., LEE, P. C., AND CHANG, R.-C. Managing flash memory in personal communication devices. In *Consumer Electronics, 1997. ISCE'97., Proceedings of 1997 IEEE International Symposium on* (1997), IEEE, pp. 177–182.

[4] 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* (2010), ACM, pp. 143–154.

[5] EDGE, J. Stream IDs and I/O hints, May 2016.

[6] EDGE, J. Stream ID status update, Mar 2017.

[7] HSIEH, J.-W., KUO, T.-W., AND CHANG, L.-P. Efficient identification of hot data for flash memory storage systems. *ACM Transactions on Storage (TOS) 2*, 1 (2006), 22–40.

[8] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, 2014), USENIX Association.

[9] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *Usenix Winter* (1995), pp. 155–164.

[10] LAKSHMAN, A., AND MALIK, P. Cassandra. http://cassandra.apache.org/, July 2008.

[11] SHIMPI, A. L. The ssd anthology: Understanding ssds and new drivers from ocz. http://db-engines.com/en/ranking/wide+column+store, February 2014.

[12] TARASOV, V., ZADOK, E., AND SHEPLER, S. Filebench: A flexible framework for file system benchmarking. *USENIX; login 41* (2016).

[13] YAN, S., LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., CHIEN, A. A., AND GUNAWI, H. S. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. In *FAST* (2017), pp. 15–28.

[14] YANG, F., DOU, K., CHEN, S., HOU, M., KANG, J., AND CHO, S. Optimizing nosql DB on flash: A case study of rocksdb. In *2015 IEEE 15th Intl Conf on Scalable Computing and Communications, Beijing, China, August 10-14, 2015* (2015), pp. 1062–1069.

[15] YANG, J., PANDURANGAN, R., CHOI, C., AND BALAKRISHNAN, V. Autostream: automatic stream management for multi-streamed ssds. In *Proceedings of the 10th ACM International Systems and Storage Conference, SYSTOR 2017, Haifa, Israel, May 22-24, 2017* (2017), pp. 3:1–3:11.

Improving Docker Registry Design based on Production Workload Analysis

Ali Anwar[1], Mohamed Mohamed[2], Vasily Tarasov[2], Michael Littley[1], Lukas Rupprecht[2], Yue Cheng[3]*, Nannan Zhao[1], Dimitrios Skourtis[2], Amit S. Warke[2], Heiko Ludwig[2], Dean Hildebrand[2]†, and Ali R. Butt[1]

[1]*Virginia Tech,* [2]*IBM Research–Almaden,* [3]*George Mason University*

Abstract

Containers offer an efficient way to run workloads as independent microservices that can be developed, tested and deployed in an agile manner. To facilitate this process, container frameworks offer a registry service that enables users to publish and version container images and share them with others. The registry service plays a critical role in the startup time of containers since many container starts entail the retrieval of container images from a registry. To support research efforts on optimizing the registry service, large-scale and realistic traces are required. In this paper, we perform a comprehensive characterization of a large-scale registry workload based on traces that we collected over the course of 75 days from five IBM data centers hosting production-level registries. We present a trace replayer to perform our analysis and infer a number of crucial insights about container workloads, such as request type distribution, access patterns, and response times. Based on these insights, we derive design implications for the registry and demonstrate their ability to improve performance. Both the traces and the replayer are open-sourced to facilitate further research.

1 Introduction

Container management frameworks such as Docker [22] and CoreOS Container Linux [3] have established containers [41, 44] as a lightweight alternative to virtual machines. These frameworks use Linux *cgroups* and *namespaces* to limit the resource consumption and visibility of a container, respectively, and provide isolation in shared, multi-tenant environments at scale. In contrast to virtual machines, containers share the underlying operating system kernel, which enables fast deployment with low performance overhead [35]. This, in turn, is driving the rapid adoption of the container technology in the enterprise setting [23].

The utility of containers goes beyond performance, as they also enable a *microservice* architecture as a new model for developing and distributing software [16, 17, 24]. Here, individual software components focusing on small functionalities are packaged into container *images* that include the software and all dependencies required to run it. These *microservices* can then be deployed and combined to construct larger, more complex architectures using lightweight communication mechanisms such as REST or gRPC [9].

To facilitate the deployment of microservices, Docker provides a *registry service*. The registry acts as a central image repository that allows users to publish their images and make them accessible to others. To run a specific software component, users then only need to "pull" the required image from the registry into local storage. A variety of Docker registry deployments exist such as Docker Hub [5], IBM Cloud container registry [12], or Artifactory [1].

The registry is a data-intensive application. As the number of stored images and concurrent client requests increases, the registry becomes a performance bottleneck in the lifecycle of a container [37, 39, 42]. Our estimates show that the widely-used public container registry, Docker Hub [5], stores at least hundreds of terabytes of data, and grows by about 1,500 new public repositories daily, which excludes numerous private repositories and image updates. Pulling images from a registry of such scale can account for as much as 76% of the container start time [37]. Several recent studies have proposed novel approaches to improve Docker client and registry communication [37, 39, 42]. However, these studies only use small datasets and synthetic workloads.

In this paper, for the first time in the known literature, we perform a large-scale and comprehensive analysis of a real-world Docker registry workload. To achieve this, we started with collecting long-span production-level traces from five datacenters in IBM Cloud container registry service. IBM Cloud serves a diverse set of customers, ranging from individuals, to small and medium businesses, to large enterprises and government institutions. Our traces cover all availability zones and many components of the registry service over the course of 75 days, which totals to over 38 million requests and accounts for more than 181.3 TB of data transferred.

We sanitized and anonymized the collected traces and then created a high-speed, distributed, and versatile Docker trace replayer. To the best of our knowledge, this is the first trace replayer for Docker. To facilitate future research and engineering efforts, we release

*Most of this work was done while at Virginia Tech.
†Now at Google.

both the anonymized traces and the replayer for public use at https://dssl.cs.vt.edu/drtp/. We believe our traces can provide valuable insights into container registry workloads across different users, applications, and datacenters. For example, the traces can be used to identify Docker registry's distinctive access patterns and subsequently design workload-aware registry optimizations. The trace replayer can be used to benchmark registry setups as well as for testing and debugging registry enhancements and new features.

We further performed comprehensive characterization of the traces across several dimensions. We analyzed the request ratios and sizes, the parallelism level, the idle time distribution, and the burstiness of the workload, among other aspects. During the course of our investigation, we made several insightful discoveries about the nature of Docker workloads. We found, for example, that the workload is highly read-intensive comprising of 90-95% pull compared to push operations. Given the fact that our traces come from several datacenters, we were able to find both common and divergent traits of different registries. For example, our analysis reveals that the workload not only depends on the purpose of the registry but also on the age of the registry service. The older registry services show more predictable trends in terms of access patterns and image popularity. Our analysis, in part, is tailored to exploring the feasibility of caching and prefetching techniques in Docker. In this respect, we observe that 25% of the total requests are for top 10 repositories and 12% of the requests are for top 10 layers. Moreover, 95% of the time is spent by the registry in fetching the image content from the backend object store. Finally, based on our findings, we derive several design implications for container registry services.

2 Background

Docker [22] is a container management framework that facilitates the creation and deployment of containers. Each Docker container is spawned from an *image*—a collection of files sufficient to run a specific containerized application. For example, an image which packages the Apache web server contains all dependencies required to run the server. Docker provides convenient tools to combine files in images and run containers from images on end hosts. Each end host runs a daemon process which accepts and processes user commands.

Images are further divided into *layers*, each consisting of a subset of the files in the image. The layered model allows images to be structured in sub-components which can be shared by other containers on the same host. For example, a layer may contain a certain version of the Java runtime environment and all containers requiring this version can share it from a single layer, reducing storage and network utilization.

2.1 Docker Registry

To simplify their distribution, images are kept in an online *registry*. The registry acts as a storage and content delivery system, holding named Docker images. Some popular Docker registries are Docker Hub [5], Quay.io [20], Artifactory [1], Google Container Registry [8], and IBM Cloud container registry [12].

Users can create *repositories* in the registry, which hold images for a particular application or system such as Redis, WordPress, or Ubuntu. Images in such repositories are often used for building other application images. Images can have different versions, known as *tags*. The combination of user name, repository name, and tag uniquely identifies an image.

Users add new images or update existing ones by pushing to the registry and retrieve images by pulling from the registry. The information about which layers constitute a particular image is kept in a metadata file called *manifest*. The manifest also describes other image settings such as target hardware architecture, executable to start in a container, and environment variables. When an image is pulled, only the layers that are not already available locally are transferred over the network.

In this study we use Docker Registry's version 2 API which relies on the concept of content addressability. Each layer has a content addressable identifier called *digest*, which uniquely identifies a layer by taking a collision-resistant hash of its data (SHA256 by default). This allows Docker to efficiently check whether two layers are identical and deduplicate them for sharing between different images.

Pulling an Image. Clients communicate with the registry using a RESTful HTTP API. To retrieve an image, a user sends a `pull` command to the local Docker daemon. The daemon then fetches the image manifest by issuing a `GET <name>/manifests/<tag>` request, where `<name>` defines user and repository name while `<tag>` defines the image tag.

Among other fields, manifest contains name, tag, and fsLayers fields. The daemon uses the digests from the fsLayers field to download individual layers that are not already available in local storage. The client checks if a layer is available in the registry by using `HEAD <name>/blobs/<digest>` requests.

Layers are stored in the registry as compressed tarballs ("blobs" in Docker terminology) and are pulled by issuing a `GET <name>/blobs/<digest>` request. The registry can redirect layer requests to a different URL, e.g., to an object store, which stores the actual layers. In this case, the Docker client downloads the layers directly from the new location. By default, the daemon downloads and extracts up to three layers in parallel.

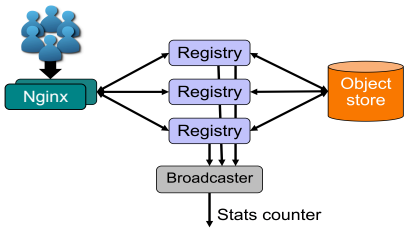


Figure 1: IBM Cloud Registry architecture. Nginx receives users requests and forwards them to registry servers. Registry servers fetch data from the backend object store and reply back.

Pushing an Image. To upload a new image to the registry or update an existing one, clients send a `push` command to the daemon. Pushing works in reverse order compared to pulling. After creating the manifest locally the daemon first pushes all the layers and then the manifest to the registry.

Docker checks if a layer is already present in the registry by issuing a `HEAD <name>/blobs/<digest>` request. If the layer is absent, its upload starts with a `POST <name>/blobs/uploads/` request to the registry which returns a URL containing a unique upload identifier (`<uuid>`) that the client can use to transfer the actual layer data. Docker then uploads layers using *monolithic* or *chunked* transfers. Monolithic transfer uploads the entire data of a layer in a single `PUT` request. To carry out chunked transfer, Docker specifies a byte range in the header along with the corresponding part of the blob using `PATCH <name>/blobs/uploads/<uuid>` requests. Then Docker submits a final `PUT` request with a layer digest parameter. After all layers are uploaded, the client uploads the manifest using `PUT <name>/manifests/<digest>` request.

2.2 IBM Cloud Container Registry

In this work we collect traces from IBM's container registry which is a part of the IBM Cloud platform [11]. The registry is a key component for supporting Docker in IBM Cloud and serves as a sink for container images produced by build pipelines and as the source for container deployments. The registry is used by a diverse set of customers, ranging from individuals, to small and medium businesses, to large enterprises and government institutions. These customers use the IBM container registry to distribute a vast variety of images that include operating systems, databases, cluster deployment setups, analytics frameworks, weather data solutions, testing infrastructures, continuous integration setups, etc.

The IBM Cloud container registry is a fully managed, highly available, high-performance, v2 registry based on the open-source Docker registry [4]. It tracks the Docker project codebase in order to support the majority of the latest registry features. The open-source functionality is extended by several microservices, offering features such as multi-tenancy with registry namespaces, a vulnerability advisor, and redundant deployment across availability zones in different geographical regions.

IBM's container registry stack consists of over eighteen components. Figure 1 depicts three components that we trace in our study: 1) Nginx, 2) registry servers, and 3) broadcaster. Nginx acts as a load balancer and forwards customers' HTTPS connections to a selected registry server based on the requested URL. Registry servers are configured to use OpenStack Swift [18, 25, 26] as a backend object store. The broadcaster provides registry event filtering and distribution, e.g., it notifies the vulnerability advisor component on new image pushes.

Though all user requests to the registry pass through Nginx, Nginx logs contain only limited information. To obtain complete information required for our analysis we also collected traces at registry servers and broadcaster. Traces from registry servers provide information about request distribution, traces from Nginx provide response time information, and broadcaster traces allow us to study layer sizes.

The IBM container registry setup spans five geographical locations: Dallas (`dal`), London (`lon`), Frankfurt (`fra`), Sydney (`syd`), and Montreal. Every geographical location forms a single Availability Zone (AZ), except Dallas and Montreal. Dallas hosts Staging (`stg`) and Production (`dal`) AZs, while Montreal is home for Prestaging (`prs`) and Development (`dev`) AZs. The `dal`, `lon`, `fra`, and `syd` AZs are client-facing and serving production workloads, while `stg` is a staging location used internally by IBM employees. `prs` and `dev` are used exclusively for internal development and testing of the registry service. Out of the four production registries `dal` is the oldest, followed by `lon`, and `fra`. `Syd` is the youngest registry and we started collecting traces for it since its first day of operation.

Each AZ has an individual control plane and ingress paths, but backend components, e.g. , object storage, are shared. This means that AZ's are completely network isolated but images are shared across AZ's. The registry setup is identical in hardware, software, and system configuration across all AZs, except for `prs` and `dev`. `prs` and `dev` are only half the size of the other AZs, because they are used for development and testing and do not directly serve clients. Every AZ hosts six registry instances, except for `prs` and `dev`, which host three.

3 Tracing Methodology

To collect traces from the IBM Cloud registry, we obtained access to the system's logging service (§3.1). The logging service collects request logs from the different system components and the log data contains a variety of information, such as the requested image, the type of request and a timestamp (§3.2). This information is sufficient to carry out our analysis. Besides collecting the

Aavailablity Zone	Duration (days)	Trace data (GB)	Filtered and anonym. (GB)	Requests (millions)	Data ingress (TB)	Data egress (TB)	Images pushed (1,000)	Images pulled (1,000)	Up since (mm/yy)
Dallas (dal)	75	115	12	20.85	5.50	107.5	356	5,000	06/15
London (lon)	75	40	4	7.55	1.70	25.0	331	2,200	10/15
Frankfurt (fra)	75	17	2	1.80	0.40	3.30	90	950	04/16
Sydney (syd)	65	5	0.5	1.03	0.29	1.87	105	360	04/16
Staging (stg)	65	25	3.2	5.90	2.41	29.2	327	1,560	-
Prestaging (prs)	65	4	0.5	0.75	0.23	2.45	65	140	-
Development (dev)	55	2	0.2	0.34	0.01	1.44	15	70	-
TOTAL	475	208	22.4	38.22	10.54	170.76	1289	10280	-

Table 1: Characteristics of studied data. `dal` and `lon` were migrated to v2 in April 2016.


```
{
    "host": "579633fd",
    "http.request.duration": 0.879271282,
    "http.request.method": "GET",
    "http.request.remoteaddr": "40535jf8",
    "http.request.uri": "v2/ca64kj67/as87d65g/blobs/b
        26s986d",
    "http.request.useragent": "docker/17.04.0-ce go/
        go1.7.5..)",
    "http.response.status": 200,
    "http.response.written": 1518,
    "id": "9f63984h",
    "timestamp": "2017-07-01T01:39:37.098Z"
}
```


Figure 2: Sample of anonymized data.

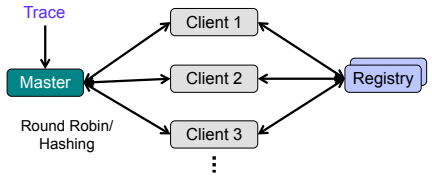


Figure 3: Trace replayer. Master parses the trace and forwards request to one of the clients either in round robin or applying hash to the `http.request.remoteaddr` field in the trace.

traces, we also developed a trace replayer (§3.3) that can be used by others to evaluate, e.g., Docker registry's performance. In this paper we used the trace replayer to evaluate several novel optimizations the were inspired by the results of the trace analysis. We made the traces and the replayer publicly available at:

`https://dssl.cs.vt.edu/drtp/`

3.1 Logging Service

Logs are centrally managed using an "ELK" stack (ElasticSearch [7], Logstash [14] and Kibana [13]). A Logstash agent on each server ships logs to one of the centralized log servers, where they are indexed and added to an ElasticSearch cluster. The logs can then be queried using the Kibana web UI or using the ElasticSearch APIs directly. ElasticSearch is a scalable and reliable text-based search engine which allows to run full text and structured search queries against the log data. Each AZ has its own ElasticSearch setup deployed on five to eight nodes and collects around 2 TB of log data daily. This includes system usage, health information, logs from different components etc. Collected data is indexed by time.

3.2 Collected Data

For trace collection we pull data from the ElasticSearch setup of each AZ for the "Registry", "Nginx", and "Broadcaster" components as shown in Figure 1. We filter all requests that relate to pushing and pulling of images, i.e. `GET`, `PUT`, `HEAD`, `PATCH` and `POST` requests. Table 1 shows the high-level characteristics of the collected traces. The total amount of our traces spans seven availability zones and a duration of 75 days from 06/20/2017 to 09/02/2017. This results in a total of 208 GB of trace data containing over 38 million requests, with more than 180TB of data transferred in them (data ingress/egress). Next, we combine the traces from different components by matching the incoming HTTP request identifier across the components. Then we remove redundant fields to shrink the trace size and in the end we anonymize them. The total size of the anonymized traces is 22.4 GB.

Figure 2 shows a sample trace record. It consists of 10 fields: the `host` field shows the anonymized registry server which served the request; `http.request.duration` is the response time of the request in seconds; `http.request.method` is the HTTP request method (e.g., `PUT` or `GET`); `http.request.remoteaddr` is the anonymized remote client IP address; `http.request.uri` is the anonymized requested url; `http.request.useragent` shows the Docker client version used to make the request; `http.response.status` shows the HTTP response code for this request; `http.response.written` shows the amount of data that was received or sent; `id` shows the unique request identifier; `timestamp` contains the request arrival time in UTC timezone.

3.3 Trace Replayer

To study the collected traces further and use them to evaluate various registry optimizations, we designed and implemented a trace replayer. It consists of a master node and multiple client nodes as shown in Figure 3. The master node parses the anonymized trace file one request at a time and forwards it to one of the clients. Requests are forwarded to clients in either round robin fashion or by hashing the `http.request.remoteaddr` field in the trace. By using hashing, the trace replayer maintains the request locality to ensure all HTTP requests corresponding to one image push or pull are generated by the same client node as they were seen by the original registry service. In some cases this option may generate workload

skewness as some of the clients issue more requests than others. This method is useful for large-scale testing with many clients.

Clients are responsible for issuing the HTTP requests to the registry setup. For all `PUT` layer requests, a client generates a random file of corresponding size and transfers it to the registry. As the content of the newly generated file is not same as the content of the layer seen in the trace, the digest/SHA256 is going to be different for the two. Hence, upon successful completion of the request, the client replies back to the master with the request latency as well as the digest of the newly generated file. The master keeps track of the mapping between the digest in the trace and its corresponding newly generated digest. For all future `GET` requests for this layer, the master issues requests for the new digest instead of the one seen in the trace. For all `GET` requests the client just reports the latency.

The trace replayer runs in two phases: warmup and actual testing. During the warmup phase, the master iterates over the `GET` requests to make sure that all corresponding manifests and layers already exist in the registry setup. In the testing phase all requests are issued in the same order as seen in the trace file.

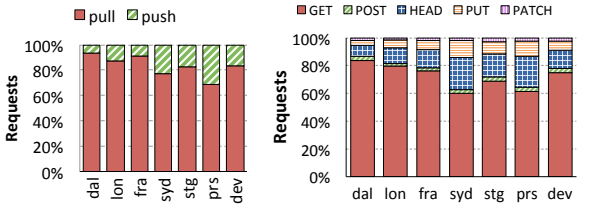
The requests are issued by the trace replayer in two modes: 1) "as fast as possible", and 2) "as is", to account for the timestamp of each request. The master side of the trace replayer is multithreaded and each client's progress is tracked in a separate thread. Once all clients finish their jobs, aggregated throughput and latency is calculated. Per-request latency and per-client latency and throughput are recorded separately.

The trace replayer can operate in two modes to perform two types of analysis: 1) performance analysis of a large scale registry setup and 2) offline analysis of traces.

Performance analysis mode. The Docker registry utilizes multiple resources (CPU, Memory, Storage, Network) and provisioning them is hard without a real workload. The performance analysis mode allows to benchmark what throughput and latency can a Docker registry installation achieve when deployed on specific provisioned resources. For example, in a typical deployment, Docker is I/O intensive and the replayer can be used to benchmark network storage solutions that act as a backend for the registry.

Offline analysis mode. In this mode, the master does not forward the requests to the clients but rather hands them off to an analytic plugin to handle any requested operation. This mode is useful to perform offline analysis of the traces. For example, the trace player can simulate different caching policies and determine the effect of using different cache sizes. In Sections §5.3 and §5.4 we use this mode to perform caching and prefetching analysis.

Additional analysis. By making our traces and trace replayer publicly available we enable more detailed analysis in the future. For example, one can create a module for the replayer's performance analysis mode that analyzes request arrival rates with a user-defined time granularity. One may also study the impact of using content delivery networks to cache popular images by running the trace replayer in the performance analysis mode. Furthermore, to understand the effect of deduplication on data reduction in the registry, researchers can conduct studies on real layers in combination with our trace replayer. The relationship between resource provisioning vs. workload demands can be established by benchmarking registry setups using our trace replayer and traces.



(a) Push vs. Pull. (b) Request Ratio.

Figure 4: Image pull vs. push ratio, and distribution of HTTP requests served by registry.

4 Workload Characterization

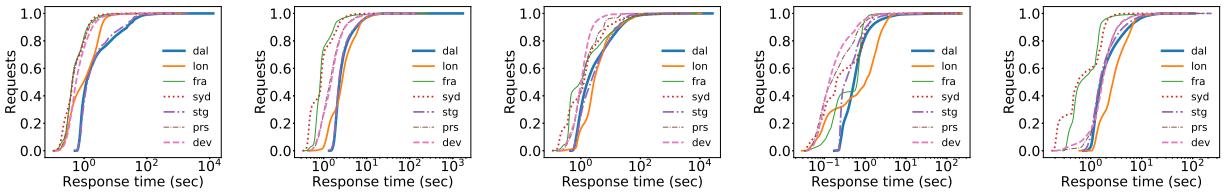
To determine possible registry optimizations, such as caching, prefetching, efficient resource provisioning, and site-specific optimizations, we center our workload analysis around the following five questions:

1. What is the general workload the registry serves? What are request type and size distributions? (§4.1)
2. Do response times vary between production, staging, pre-staging, and development deployments? (§4.2)
3. Is there spatial locality in registry requests? (§4.3)
4. Do any correlations exist among subsequent requests? Can future requests be predicted? (§4.4)
5. What are the workload's temporal properties? Are there bursts and is there any temporal locality? (§4.5)

4.1 Request Analysis

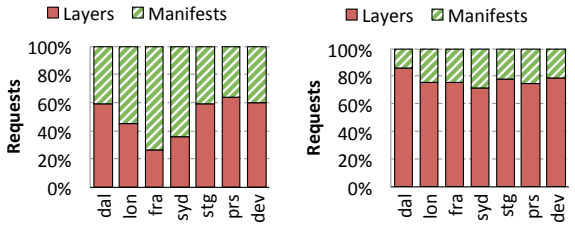
We start with the request type and size analysis to understand the basic properties of the registry's workload.

Request type distribution. Figure 4(a) shows the ratio of images pulled from vs. pushed to the registry. As expected, the registry workload is read-intensive. For `dal`, `lon`, and `fra`, we observe that 90%–95% of requests are pulls (i.e. reads). `Syd` exhibits a lower pull ratio of 78% because it is a newer installation and, therefore, it is being populated more intensively than mature registries. Non-production registries (`stg`, `prs`, `dev`) also demonstrate a lower (68–82%) rate of pulls than production registries, due to higher image churn rates. Each push



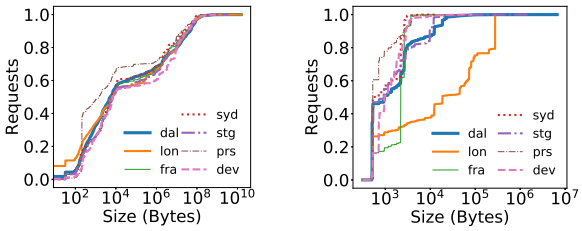
(a) GET layers. (b) PUT layers. (c) PATCH layers. (d) GET manifests. (e) PUT manifests.

Figure 8: CDF of response time for `GET`, `PUT`, `PATCH` requests to layers and `GET` and `PUT` requests to manifests.



(a) Pull. (b) Push.

Figure 5: The ratio of requests that access either an image manifest or a layer.



(a) Layers. (b) Manifests.

Figure 6: CDF of manifest and layer sizes for GET requests.

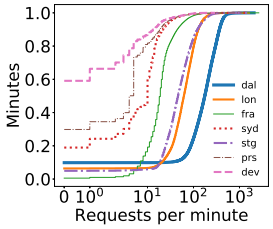


Figure 7: CDF of requests per minute.

or pull consists of a sequence of HTTP requests as discussed in §2. Figure 4(b) shows the distribution of different HTTP requests served by the registry. All registries receive more than 60% of `GET` requests and 10%–22% of `HEAD` requests. `PUT` requests are 1.9–5.8× more common than `PATCH` requests because `PUT`s are used for uploading manifests (in addition to layers) and many layers are small enough to be uploaded in a single request.

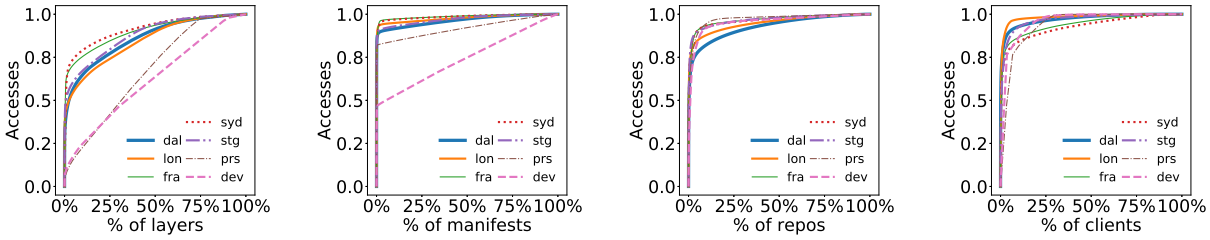
Figures 5(a) and 5(b) show the manifest vs. layer ratio for pull and push image requests, respectively. We include `GET` requests in pull count, while pushes include `PUT` *or* `HEAD` requests to account for attempts to upload the layers that are already present in the registry. For pulls we observe that, except for `syd` and `fra`, 50% or more requests retrieve layers rather than manifests. This is expected as a single manifest refers to multiple layers. Our investigation revealed that the divergent behavior of `syd` and `fra` is caused by their clients trying to pull images that they have already pulled in the past. This results into many `GET` requests to the manifests without subsequent `GET` requests to the layers. For pushes, we see that accesses to layers dominate accesses to manifests.

Request size distribution. Figure 6 shows the CDF of manifest and layer sizes for `GET` and `PUT` requests. In Figure 6(a) we observe that about 65% of the layers are smaller than 1 MB and around 80% are smaller than 10 MB. In Figure 6(b), we find that the typical manifest size is around 1 KB for all AZs except for `lon` where 50% of the GET requests are for manifests larger than 10 KB. For `lon`, a large number of requests are for manifests that are compatible with the older Docker version, hence increasing their size. We observe similar trends for `PUT`s for all the AZs (not shown in the Figures).

4.2 Registry Load and Response Time

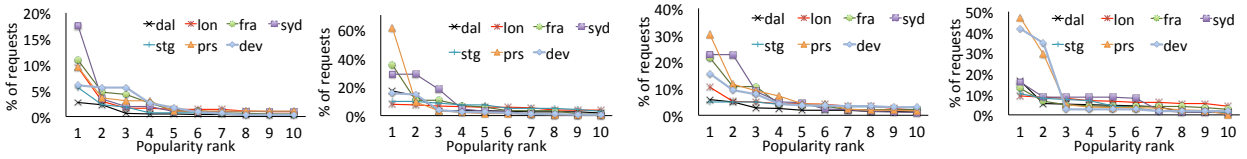
Load distribution. Figure 7 shows the CDF of received requests per minute over time. `dal` has the highest overall load and services at least 100 requests per minute more than 80% of the time. `lon` and `stg` are second and third, followed by `fra`, `syd`, `prs`, and `dev`, in descending order. This trend is consistent across the different request types (not shown). The ordering of AZs by the load yields two main observations. First, development and pre-staging registries experience low utilization. `dev`, for example, does not receive any requests 57% of the time. Second, registry load increases with its age. In our traces `dal` and `lon` have been running the longest while `fra` and `syd` have only been deployed recently.

Response time distribution. Figure 8 shows the CDFs of response time of different requests to layers and to



(a) Layer popularity. (b) Manifest popularity. (c) Repository popularity. (d) Client popularity.

Figure 10: CDF of access for layers, manifests, repositories, and clients.



(a) Layers popularity (b) Manifests popularity (c) Repos popularity (d) Clients popularity

Figure 11: Popularity of top ten layers, manifests, repositories, and clients.

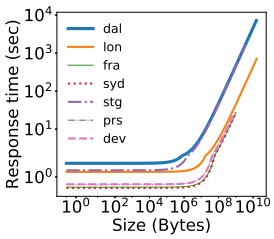


Figure 9: Dependency of response time on the layer size.

manifests. As `dal` is the highest loaded AZ, its request response times are higher compared to other AZs. More than 60% of the `GET` layer requests take more than one second to finish (Figure 8(a)). For the top 25% of requests we see a response time of ten seconds and higher. `fra`, `syd`, `prs`, and `dev` are not highly loaded, so they have the lowest latency in serving the `GET` layer requests. `PUT` and `PATCH` layer requests (Figures 8(b) and 8(c)) follow similar trends. However, `PATCH` requests are visibly slower than `GET`s and `PUT`s as they carry more data. We also analyze the dependency of response time on the layer size (see Figure 9) and find that response times remain nearly constant for layers smaller than 1 MB and then start to grow linearly.

Figure 8(d) and 8(e) show the response time distributions for `PUT` and `GET` requests to manifests, respectively. Since manifests are smaller and cached, we observe significantly smaller and more stable latencies than that of requests serving layers. One interesting observation is that `lon` has the highest response time when serving manifests (300-400 ms more than `dal`). This is because `lon` serves manifests with larger sizes compared to other AZs. This is also consistent with the results shown in Figure 6(b). For the `PUT` manifest requests we observe a more uniform trend across the AZs as the size of the new manifests is similar for all the AZs.

4.3 Popularity Analysis

In this section we study the popularity of layers, manifests, users, repositories, and clients to answer whether image accesses are skewed and produce hot-spots.

Popularity distribution. Figure 10 shows the CDF of the access rate of layers, manifests, repositories, and clients. Figure 10(a) demonstrates that there is a heavy skew in layer accesses. For example, the 1% most frequently accessed layers in `dal` account for 42% of all requests while in `syd` this increases to 59%. However, requests to the `dev` and `prs` sites are almost evenly distributed. The reason is that during testing, developers frequently push or pull images that are not available neither at registry nor at client side. We also observe that the younger AZs experience a higher skew compared to the older AZs. We believe this is due to the fact that accesses become more evenly distributed over a long period of time.

For manifest accesses (Figure 10(b)) skew is more significant than for layers. This confirms that there are indeed hot *images* which can benefit from caching. Repository accesses (Figure 10(c)) reflect this fact but show slightly less skew as manifests are contained in repositories and hence there are less repositories than manifests. The same trend holds for users under which repositories are stored (not shown in Figure 10). Furthermore, we

find that client accesses are also heavily skewed (Figure 10(d)). This means that there are few highly active clients while most of them only submit few requests. This trend is consistent across all AZs. While this does not directly affect the workload, clients can be biased towards a certain subset of images which will contribute to the access skew.

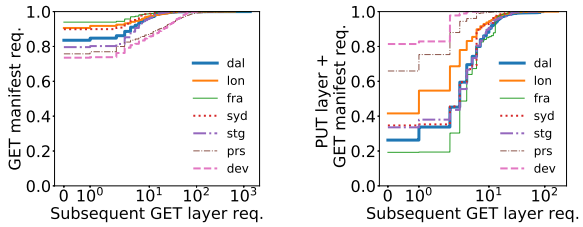
Top-10 analysis. To further understand the popularity distribution of registry items, we analyze the top 10 hottest items in each category. Figure 11(a) shows the access rates for the top 10 layers, which account for 8%–30% of all accesses depending on the registry. The most popular layer (rank 1) in all AZs absorbs 1–10% of all requests while in `syd` it absorbs 19%. The popularity rate drops rapidly as we move from most popular to tenth most popular layer. The relative amount of accesses for the top 10 layers is the lowest for `dal` as it stores the most layers and experiences the highest amount of requests.

For the top 10 manifests (Figure 11(b)), we observe that some container images are highly popular and account for as many as 40% of the requests in `fra` and `syd`, and 60% in `prs`. Note that a manifest is fetched even if the image is already cached at the client side. Hence, a manifest fetch does not necessarily mean that the corresponding layers are fetched (§4.4). Similar to Figure 10, the skew decreases for repository popularity (Figure 11(c)) and user popularity. Part of the reason for the small number of highly accessed images in younger AZ is that registry services in production are tested periodically to monitor their health and performance. For the AZs with a smaller workload (`fra` and `syd`), those test images make up three out of the top five most accessed images. We intentionally did not exclude these images from our analysis as they are typically part of the registry workload in production environments.

Figure 11(d) shows that the most popular client submits around 15% of the total requests. This excludes `prs` and `dev`, which are used by the registry development team for internal development and testing. These two AZs only have a small number of clients, and 2 clients contribute around 80% of all requests.

Overall, the detailed top-10 analysis shows that while there are a few highly popular test images, the popularity of the remaining hot items is decreasing fast and hence, overly small caches will be insufficient to effectively cache data. For example, based on these results, we estimate that a cache size of 2% of the dataset size can provide 40% and higher hit ratios.

We also analyzed the pull count of the top 10 hottest repositories on Docker Hub. We found that the most downloaded repository (*Alpine Linux*) has a pull count of more than 1 billion while the tenth most popular repository (*Ubuntu*) has a pull count of 369 million. This trend



(a) Subsequent GET layers per GET manifest.

(b) Subsequent GET layers per PUT layer + GET manifest.

Figure 12: Relationship between GET manifest and subsequent GET layer requests.

further verifies that caching can be highly effective for increasing the performance of container registries.

4.4 Request Correlation

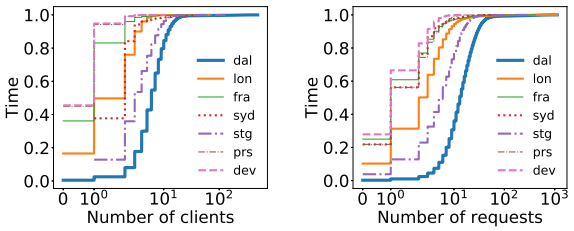
In this section we investigate whether a `GET` request for a certain manifest always results in subsequent `GET` requests to the corresponding layers. Therefore, we define a client *session* as the duration from the time a client connects until a certain threshold. We varied the threshold from 1 to 10 minutes but could not observe significant differences. However, values less than 1 minute dramatically affect the results as that is less than the typical time a client takes to pull an image. We set the session threshold to 1 minute and then count all `GET` layer requests that follow a `GET` manifest request within a session.

Figure 12(a) shows the CDF of the number of times clients issue the corresponding `GET` layer requests after retrieving a manifest. In most cases, ranging from 96% for `dev` to 73% for `fra`, `GET` manifest requests are not followed by any subsequent request. The reason is that whenever a client has already fetched an image and then pulls an image, only the manifest file is requested to check if there has been any change in the image. This shows that there is no strong correlation between `GET` manifest and layer requests.

We then focused only on `GET` manifest request that were received within the session of a `PUT` request to the same repository, from which the manifest is fetched (Figure 12(b)). This leads to a significant increase in subsequent `GET` layer requests within a session for all production and staging AZs. The manifest requests not followed by `GET` layer requests are due to the fact that clients sometimes pull the same image more than once. Overall, our analysis reveals a strong correlation between `GET` manifest and subsequent layer requests if preceding `PUT` requests are considered.

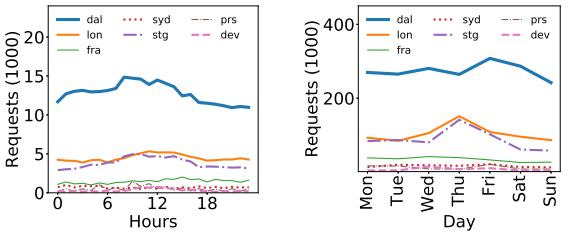
4.5 Temporal Properties

Next, we investigate whether the workload shows any temporal patterns.



(a) Client concurrency. (b) Request concurrency.

Figure 13: CDF of client and request concurrency.



(a) Hours of day (b) Days of week

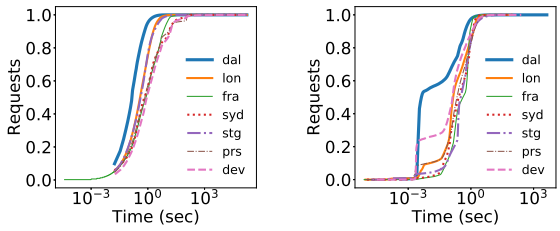
Figure 14: Average number of requests over the tracing period for each hour of the day and day of the week.

Client and request concurrency. We start with measuring how many clients and requests are *active* at a given point in time. Active clients are the clients that maintain a connection to the registry, while active requests are the requests that were received but have not yet been processed by the registry. Figures 13(a) and 13(b) show the results for clients and requests, respectively. Overall, the median number of concurrently active clients is low, ranging from 0.6 clients for `dev` to 7 clients for `dal`. However, there are peak periods during which several hundred clients are connected at the same time. We observe a similar trend for concurrently active requests.

To understand whether these peak periods follow a certain pattern, we plot the average number of requests per hour and day across all traced hours and days in Figures 14(a) and Figure 14(b). For `dal`, we observe that request numbers are decreasing during the night and over the weekend. While other AZs show a similar trend, it is less pronounced at those sites. This suggests that registry resources can be provisioned statically for hours and days. We plan to explore short-term bursts in the future.

Inter-arrival and idle times. Next, we look at request inter-arrival and idle times to study whether the registry experiences longer periods of idleness, during which less resources are required. Inter-arrival time is defined as the time between two subsequent requests. Idle time is the time during which there are no active requests.

Figure 15(a) shows the inter-arrival times. `dal`, `lon` and `stg` experience the highest request frequency with a 99th percentile of inter-arrival time around 3 s while for other AZs it is around 110 s. When looking at idle times (Figure 15(b)), we observe that idle periods are short and in most cases below 1 s. However, the amount of experienced idle periods varies significantly across AZs. Throughout the entire collection time, `dal` saw only approximately 0.1 million idle periods while `lon` experienced more than 1.5 million. While some AZs experience a large amount of idle periods, their duration is short and hence, they are hard to exploit with traditional resource provisioning approaches.



(a) Request inter-arrival time. (b) Request idle time.

Figure 15: CDF of request inter-arrival and idle times

4.6 Analysis Summary

We summarize our analysis in seven observations:

1. `GET` requests are dominant in all registries and more than half of the requests are for layers, opening an opportunity for effective layer caching and prefetching at the registry.
2. 65% and 80% of all layers are smaller than 1 MB and 10 MB, respectively, making individual layers suitable for caching.
3. The registry load is affected by the registry's intended use case and the age of the registry. Younger, non-production registries experience lower loads compared to longer running, production systems. This should be considered when provisioning resources for an AZ to save cost and use existing resources more efficiently.
4. Response times correlate with registry load and hence also depend on the age (younger registries experience less load) and the use case of the registry.
5. Registry accesses to layers, manifests, repositories, and by users are heavily skewed. Few extremely hot images are accessed frequently but the popularity drops rapidly. Therefore, caching techniques are feasible but cache sizes should be selected carefully.
6. There is a strong correlation between PUT requests and subsequent GET manifest and GET layer requests. The registry can leverage this pattern to prefetch the layers from the backend object store to the cache, significantly reducing pull latencies for the client. This correlation exists for both popular as well as non-

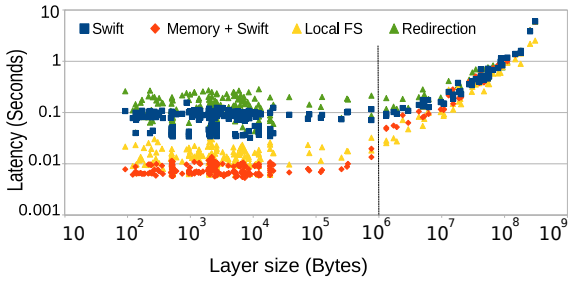


Figure 16: Effect of various backend storage technologies on registry performance.

popular images.

7. While there are weak declines in request rates during weekends, we did not find pronounced repeated spikes that can be used to improve resource provisioning.

5 Registry Design Improvements

In this section, we use the observations from §4 to design two improvements to the container registry: (i) a multi-layer cache for popular layers; and (ii) a tracker for newly pushed layers, which enables prefetching of the newest layers from the backend object store. We evaluate our design using our trace replayer.

5.1 Implementation

We implemented the trace replayer and its performance analysis mode in Python. This mode allows us to study the effect of different storage technologies on response latency. We use Bottle [2] for routing requests between the master and clients and the dxf library [6] for storing and retrieving data in/from the registry. For caching and prefetching, we implemented two separate modules. To implement the in-memory layer cache, we modified the Swift storage driver for the registry (about 200 LoC modified/added). The modified driver stores the small sized layers in memory and uses Swift for larger layers.

5.2 Performance Analysis

The registry is launched on a 32 core machine with 64 GB of main memory and 512 GB of SSD storage, and the Swift object store runs on a separate set of nodes of similar configuration. The trace replayer is started on an additional six nodes (one master and five clients). We made sure that the trace replayer or the object store are never the bottleneck during this analysis. All nodes are connected via 10 Gbps network links. To drive the analysis, the trace replayer is used to replay 10,000 requests from the `dal` trace (August 1st, 2017 starting at 12 am).

We compare four different backends: 1) Swift; 2) memory for layers smaller than 1 MB and Swift for rest of the layers (Memory + Swift); 3) local file system with SSD (Local FS); and 4) Redirection, i.e. the registry replies back with the link to the layer in Swift and the client then fetches the layer directly from Swift. Swift, Local FS, and Redirection are by default supported by the Docker registry.

Figure 16 shows the latency vs. layer size for all backends. We observe that, for small sized layers (i.e. layers less than 1 MB), the response time is the lowest (0.008 s on average) for Memory + Swift. This is followed by Local FS, which yields an average response time of around 0.013 s and Swift with an average response time of 0.07 s. Redirection performs the worst with average response time of 0.11 s.

For large size objects, we observe that Memory + Swift and Local FS are comparable and both beat Swift and Redirection. Moreover, for layers slightly larger than 1 MB, Swift outperforms Redirection. However, for very large layers, Swift and Redirection perform similarly, with average response latencies of 0.63 s and 0.59 s, respectively.

The results highlight the advantage of having a fast backend storage system for the registry, and demonstrate the opportunity for caching to significantly improve registry performance.

5.3 Two-level Cache

In designing our cache, we chose to exploit the high capacity memory as well as SSDs that are present in modern server machines. We also observed that a small fraction of layers are too large to justify the use of memory to cache them. Consequently, we design a two-level cache consisting of main memory (for smaller layers) and SSDs (for larger layers). We do not have to deal with possible cache invalidation as layers are content addressable and any change in a layer also changes its digest. This results in a "new" layer for caching while the older version of the layer is no longer accessed and eventually gets evicted from the cache.

Hit ratio analysis. We perform a simulation-based evaluation of our two-level cache for the registry servers. For these experiments, we mimic the IBM registry server setup. We simulate the same number of servers as there are in each AZ and for each server, we add memory and SSD caches. The registry servers do not share the cache as the Docker registry implementation is non-distributed. However, the setup can be scaled by adding more registry servers behind the Nginx load balancer.

We use the LRU caching policy for both the memory and the SSD level cache. We select cache sizes of 2%, 4%, 6%, 8%, and 10% of the data ingress for each AZ. The data ingress of an AZ is the amount of *new* data stored in that AZ during the 75 days period during which we collected the traces. For the SSD level cache sizes, we select 10×, 15×, and 20× the size of the memory cache. Any object evicted from the memory cache goes first to

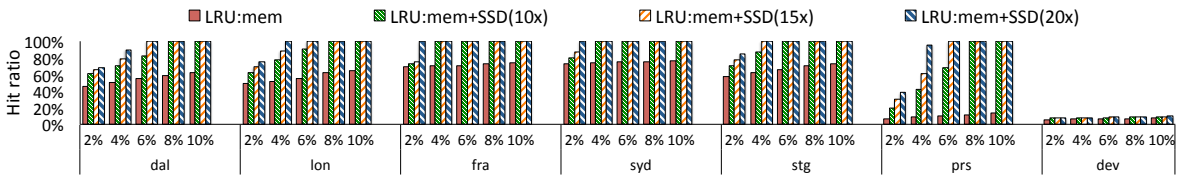


Figure 17: Hit ratio of LRU caching policy for both the memory and the SSD level cache.

the SSD cache before it is completely evicted. We store layers smaller than 100 MB in the memory level cache, while larger layers are stored in the SSD level cache. For our analysis, we iterate over the traces to warm the cache and start calculating the hit and miss ratios upon observing the first eviction from the cache. Given our long trace period, the first eviction happens early relative to the time it takes to replay all traces.

Figure 17 shows the hit ratios. We see that for the production and staging AZs, adding even a single level of LRU-based memory cache yields a hit ratio of 40% for `dal` with a cache size of 2% of ingress data and as high as 78% for `fra` and `syd` with a cache size of 10% of ingress data.

Increasing the cache size increases the hit ratio, until it reaches the max of 78%. This is because we only put layers less than 100 MB in the memory cache. However, when we enable the second level cache, we achieve a combined hit ratio of 100% with 6% cache size for `dal` and 4% cache size for the other four AZs. We observe different results for the `prs` and `dev` AZs. As these two traces represent testing interactions by the registry development team, we do not see any advantage of using the cache in this case.

5.4 Prefetching Layers

Our second design improvement is to enable prefetching of layers from the backend object store by predicting what layers are most likely to be requested. Therefore, we use our observations of the push-pull relationship established in §4.4 to predict what layers to prefetch as shown in Algorithm 1.

In §4.4, we observed that the incoming PUT requests determine which layers will be prefetched when the registry receives a subsequent GET manifest request. When a PUT is received, the repository and the layer specified in the request will be added to a look up table that includes the request arrival time and the client address. When a GET manifest request is received from a client within a certain threshold LM_{thresh}, the host checks if the look up table contains the repository specified in the request. If it is a hit and the client's address is not present in the table, then the address of the client is added to the table and the layer is prefetched from the backend object store. Note that both the amount of time that the entries

Algorithm 1: Layers Prefetching Algorithm.

Input: LM_{thresh}: Threshold for duration between PUT layer and subsequent GET manifest requests, ML_{thresh}: Threshold for duration to keep prefetched layer.

```
1  while true do
2      r ← request received
3      if r = PUTlayer then
           /* Create new entry for layer                          */
4          RepoMap[r.repo] ← NewEntry(r.client, r.layer)
5          RepoMap[r.repo] ← set LM_timer
           /* When LM_timer > LM_thresh, entry is evicted         */
6      else if r = GETmanifest then
7          if r.client not in RepoMap[r.repo] for r.layer then
8              RepoMap[r.repo] ← add r.client
9              PrefetchedLayers ← prefetch r.layer
10             PrefetchedLayers[r.layer] ← set ML_timer
               /* When ML_timer > ML_thresh, layer is evicted     */
11             prefetch++
12     else if r = GETlayer then
13         if r.layer in PrefetchedLayers then
14             serve from PrefetchedLayers[r.layer]
15             prefetch_hit++
16         else
17             serve from object store
```

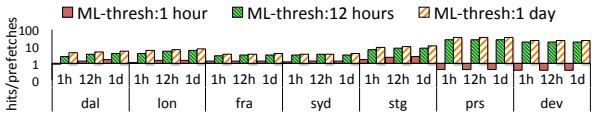


Figure 18: Hits/prefetch ratio.

remain in the look up table and how long the layers are cached at the registry side, defined by ML_{thresh}, are configurable.

Hits/prefetch analysis. We tested our algorithm using different values for retaining look up table entries, LM_{thresh}, and retaining prefetched layers, ML_{thresh}. We use values of 1 hour, 12 hours, and 1 day for each of the threshold parameters. Figure 18 shows the results. Single bars represent ML_{thresh} values while groups of bars are assigned to LM_{thresh} values.

On one hand, we find that increasing ML_{thresh} can significantly increase the hit/prefetch ratio. On the other hand, increasing the retention threshold for the *look up* table entries only marginally increases the hit ratio. This is because the longer an entry persists in the table, the fewer prefetches it serves as the record of clients added to the table increases. We also find that the maximum

amount of memory used by dal, lon, fra, syd, prs, and dev is 10 GB, 1.7 GB, 0.5 GB, 1 GB, 2 MB, and 69 MB respectively. We note that for both prs and dev the maximum amount of memory is low because they experience less activity and therefore contain less PUT requests compared to other cases.

Our analysis shows that it is possible to improve registry performance by adding an appropriate sized cache. For small layers, a cache can improve response latencies by an order of magnitude and achieve hit ratios above 90%. We also show that it is possible to predict the GET layer requests under certain scenario to facilitate prefetching.

6 Related Work

To put our study in context we start with describing related research on Docker containers, Docker registry, workload analysis, and data caching.

Docker containers. Improving performance of container storage has recently attracted attention from both industry and academia. DRR [34] improves common copy-on-write performance targeting a dense container-intensive workload. Tarasov et al. [45] study the impact of the storage driver choice on the performance of Docker containers for different workloads running inside the containers. Contrary to this work, we focus on the registry side of a container workload.

Docker registry. Other works have looked at optimizing image retrieval from a registry side [37, 42]. Slacker [37] speeds up the container startup time by utilizing lazy cloning and lazy propagation. Images are fetched from a shared NFS store and only the minimal amount of data needed to start the container is retrieved initially. Additional data is fetched on demand. However, this design tightens the integration between the registry and the Docker client as clients now need to be connected to the registry at all times (via NFS) in case additional image data is required. Contrariwise, our study focuses on the current state-of-the-art Docker deployment in which the registry is an independent instance and completely decoupled from the clients.

CoMICon [42] proposes a system for cooperative management of Docker images among a set of nodes using peer-to-peer (P2P) protocol. In its essence, CoMICon attempts to fetch a missing layer from a node in close proximity before asking a remote registry for it. Our work is orthogonal to this approach as it analyzes a registry production workload. The results of our analysis and the collected traces can also be used to evaluate new registry designs such as CoMICon.

To the best of our knowledge, similar to IBM Cloud, most public registries [5, 8, 19] use the open-source implementation of the Docker registry [4]. Our findings are applicable to all such registry deployments.

Workload analysis studies. A number of works [27, 38] have studied web service workloads to better understand how complex distributed systems behave at scale. Similar studies exist [31, 30] which focus on storage and file system workloads to understand access patterns and locate performance bottlenecks. No prior work has explored the emerging container workloads in depth.

Slacker [37] also includes the HelloBench [10] benchmark to analyze push/pull performance of images. However, Slacker looks at client-side performance while our analysis is focused at registry side. Our work takes a first step in performing a comprehensive and large-scale study on real-world Docker container registries.

Caching and prefetching. Caching and prefetching have long been effective techniques to improve system performance. For example, modern datacenters use distributed memory cache servers [15, 21, 32, 33] to improve database query performance by caching the query results. A large body of research [28, 29, 36, 40, 43, 46, 47] studied the effects of combining caching and prefetching. In our work we demonstrate that the addition of caches significantly improves container registry's performance, while layer prefetching reduces the pull latency for large and less popular images.

7 Conclusion

Docker registry platform plays a critical role in providing containerized services. However, heretofore, the workload characteristics of production registry deployments have remained unknown. In this paper, we presented the first characterization of such a workload. We collected and analyzed large-scale trace data from five geographically distributed datacenters housing production Docker registries. The traces span 38 million requests over a period of 75 days, resulting in 181.3 TB of traces.

In our workload analysis we answer pertinent questions about the registry workload and provide insights to improve the performance and usage of Docker registries. Based on our findings, we proposed effective caching and prefetching strategies which exploit registry-specific workload characteristics to significantly improve performance. Finally, we have open-sourced our traces and also provide a trace replayer, which can be used to serve as a solid basis for new research and studies on container registries and container-based virtualization.

Acknowledgments. We thank our shepherd, Pramod Bhatotia, and reviewers for their feedback. We would also like to thank Jack Baines, Stuart Hayton, James Hart, IBM Cloud container services team, and James Davis. This work is sponsored in part by IBM, and by the NSF under the grants: CNS-1565314, CNS-1405697, and CNS-1615411.

References

[1] Artifactory. https://www.jfrog.com/confluence/display/RTF/Docker+Registry.

[2] Bottle: Python Web Framework. https://github.com/bottlepy/bottle.

[3] CoreOS. https://coreos.com/.

[4] Docker-Registry. https://github.com/docker/docker-registry.

[5] Dockerhub. https://hub.docker.com.

[6] dxf. https://github.com/davedoesdev/dxf.

[7] ElastiSearch. https://github.com/elastic/elasticsearch.

[8] Google Container Registry. https://cloud.google.com/container-registry/.

[9] gRPC. https://grpc.io/.

[10] HelloBench. https://github.com/Tintri/hello-bench.

[11] IBM Cloud. https://www.ibm.com/cloud-computing/.

[12] IBM Cloud Container Registry. https://console.bluemix.net/docs/services/Registry/index.html.

[13] Kibana. https://github.com/elastic/kibana.

[14] Logstash. https://github.com/elastic/logstash.

[15] Memcached. https://memcached.org/.

[16] Microservices and Docker containers. goo.gl/UrVPdU.

[17] Microservices Architecture, Containers and Docker. goo.gl/jsQ1sL.

[18] OpenStack Swift. https://docs.openstack.org/swift/.

[19] Project Harbor. https://github.com/vmware/harbor.

[20] Quay.io. https://quay.io/.

[21] Redis. https://redis.io/.

[22] What is Docker. https://www.docker.com/what-docker.

[23] 451 RESEARCH. Application Containers Will Be a $2.7Bn Market by 2020. http://bit.ly/2uryjDI.

[24] AMARAL, M., POLO, J., CARRERA, D., MOHOMED, I., UNUVAR, M., AND STEINDER, M. Performance evaluation of microservices architectures using containers. In *IEEE NCA* (2015).

[25] ANWAR, A., CHENG, Y., GUPTA, A., AND BUTT, A. R. Taming the cloud object storage with mos. In *ACM PDSW* (2015).

[26] ANWAR, A., CHENG, Y., GUPTA, A., AND BUTT, A. R. Mos: Workload-aware elasticity for cloud object stores. In *ACM HPDC* (2016).

[27] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS* (2012).

[28] BUTT, A. R., GNIADY, C., AND HU, Y. C. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *ACM SIGMETRICS* (2005).

[29] CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst. 14*, 4 (Nov. 1996), 311–343.

[30] CHEN, M., HILDEBRAND, D., KUENNING, G., SHANKARANARAYANA, S., SINGH, B., AND ZADOK, E. Newer is sometimes better: An evaluation of nfsv4.1. In *ACM SIGMETRICS* (2015).

[31] CHEN, Y., SRINIVASAN, K., GOODSON, G., AND KATZ, R. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *ACM SOSP* (2011).

[32] CHENG, Y., GUPTA, A., AND BUTT, A. R. An in-memory object caching framework with adaptive load balancing. In *ACM EuroSys* (2015).

[33] CHENG, Y., GUPTA, A., POVZNER, A., AND BUTT, A. R. High performance in-memory caching through flexible fine-grained services. In *ACM SOCC* (2013).

[34] DELL EMC. Improving Copy-on-Write Performance in Container Storage Drivers. https://www.snia.org/sites/default/files/SDC/2016/presentations/capacity_optimization/FrankZaho_Improving_COW_Performance_ContainerStorage_Drivers-Final-2.pdf.

[35] FELTER, W., FERREIRA, A., RAJAMONY, R., AND RUBIO, J. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *IEEE ISPASS* (2015).

[36] GNIADY, C., BUTT, A. R., AND HU, Y. C. Program-counter-based pattern classification in buffer caching. In *USENIX OSDI* (2004).

[37] HARTER, T., SALMON, B., LIU, R., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Slacker: Fast Distribution with Lazy Docker Containers. In *USENIX FAST* (2016).

[38] HUANG, Q., BIRMAN, K., VAN RENESSE, R., LLOYD, W., KUMAR, S., AND LI, H. C. An analysis of facebook photo caching. In *ACM SOSP* (2013).

[39] KANGJIN, W., YONG, Y., YING, L., HANMEI, L., AND LIN, M. Fid: A faster image distribution system for docker platform. In *IEEE AMLCS* (2017).

[40] LI, M., VARKI, E., BHATIA, S., AND MERCHANT, A. Tap: Table-based prefetching for storage caches. In *USENIX FAST* (2008).

[41] MENAGE, P. B. Adding Generic Process Containers to the Linux Kernel. In *Linux Symposium* (2007).

[42] NATHAN, S., GHOSH, R., MUKHERJEE, T., AND NARAYANAN, K. CoMICon: A Co-Operative Management System for Docker Container Images. In *IEEE IC2E* (2017).

[43] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *ACM SOSP* (1995).

[44] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *ACM EuroSys* (2007).

[45] TARASOV, V., RUPPRECHT, L., SKOURTIS, D., WARKE, A., HILDEBRAND, D., MOHAMED, M., MANDAGERE, N., LI, W., RANGASWAMI, R., AND ZHAO, M. In search of the ideal storage configuration for Docker containers. In *IEEE AMLCS* (2017).

[46] WIEL, S. P. V., AND LILJA, D. J. When caches aren't enough: data prefetching techniques. *Computer 30*, 7 (Jul 1997), 23–30.

[47] ZHANG, Z., KULKARNI, A., MA, X., AND ZHOU, Y. Memory resource allocation for file system prefetching: From a supply chain management perspective. In *ACM EuroSys* (2009).

RAID+: Deterministic and Balanced Data Distribution for Large Disk Enclosures

Guangyan Zhang*, Zican Huang*, Xiaosong Ma†, Songlin Yang*, Zhufan Wang*, Weimin Zheng*
*Tsinghua University, †Qatar Computing Research Institute, HBKU

Abstract

Existing RAID solutions partition large disk enclosures so that each RAID group uses its own disks exclusively. This achieves good performance isolation across underlying disk groups, at the cost of disk under-utilization and slow RAID reconstruction from disk failures.

We propose RAID+, a new RAID construction mechanism that spreads *both normal I/O and reconstruction workloads* to a larger disk pool in a balanced manner. Unlike systems conducting randomized placement, RAID+ employs deterministic addressing enabled by the mathematical properties of mutually orthogonal Latin squares, based on which it constructs 3-D data templates mapping a logical data volume to uniformly distributed disk blocks across all disks. While the total read/write volume remains unchanged, with or without disk failures, many more disk drives participate in data service and disk reconstruction. Our evaluation with a 60-drive disk enclosure using both synthetic and real-world workloads shows that RAID+ significantly speeds up data recovery while delivering better normal I/O performance and higher multi-tenant system throughput.

1 Introduction

For the past 30 years, Redundant Array of Inexpensive Disks (RAID) [40] has been used pervasively in servers and shared computing platforms. With parity-based RAID levels (*e.g.*, RAID-5 and RAID-6), users obtain high performance via parallel accesses and reliability via data redundancy.

With continued advance in disk capacity and slow improvement in speed, however, RAID rebuild time keeps increasing [13, 54]. For example, a recent NetApp document specifies that a 2TB SATA 7200-RPM disk takes 12.8 hours to rebuild on an idle system [12]. When performed online on a heavily loaded system, rebuild can take dramatically longer. Such slow rebuild brings two consequences. First, it raises the risk of a second failure and consequently data loss. Second, prolonged recovery subjects foreground applications to long periods of I/O performance degradation. Note that high-performance solid-state drives (SSDs) actually exacerbate this problem, as their growing deployment promotes storage system construction using more high-density, low-performance hard disks [43].

One inherent reason for such slow recovery is that, with conventional RAID, each disk drive involved in RAID reconstruction is read or written entirely. Despite the growing width of RAID arrays (with each array typically containing several to around a dozen disks), the recovery time is determined by reading/writing an entire disk. No matter how many disk arrays coexist in a shared/virtual storage system, resources are isolated between underlying RAID arrays. Idle or lightly-loaded disks cannot offer help to peers in other RAID arrays, who might be overwhelmed by high access traffic, RAID recovery, or, in the worst case, both.

Many approaches to enhancing the reconstruction performance have been proposed [3, 18, 21, 28, 42, 52, 55, 56], which fall into three categories: 1) *designing better data layout* in a disk group [18, 48, 52, 56], 2) *optimizing the reconstruction workflow* [19, 31, 45, 54], and 3) *improving the rate control of RAID reconstruction* [32, 44, 47]. Most methods focused on a single RAID group, and to our best knowledge, no solution yet has eliminated load imbalance both in normal operations and during RAID reconstruction. While random data placement can utilize larger groups of disks [12, 16, 41, 51], it requires extra book keeping and lookup, and does not deliver load balance within shorter ranges of blocks (crucial for sequential access and RAID rebuild performance, as shown in our experiments).

This paper presents RAID+, a new RAID construction mechanism that spreads *both normal and reconstruction I/O* to effectively utilize emerging commodity enclosures (such as the NetAPP DE6600 and EMC VNX-series) with dozens of or even 100+ disks. Unlike systems conducting random placement, RAID+ employs a deterministic addressing algorithm that leverages the mathe-

matical properties of *mutually orthogonal Latin squares*. Such properties allow RAID+ to construct 3-D data templates, each employing a user-specified RAID level and stripe width, that map logical data extents to uniformly distributed disk blocks within a larger disk pool.

While the total read/write volume remains unchanged, with or without disk failures, RAID+ enlists many more disk drives in data service and disk reconstruction. This allows it to provide more consistent performance, much faster recovery, and better protection from permanent data loss. In addition, in multi-tenant settings it automatically lends elastic resources to individual workloads' varying intensity, via a flexible and scalable integration of multiple disk groups. We find that this often leads to higher overall resource utilization, though like most schemes for workload consolidation, in the worst case it may incur I/O interference. Such elasticity, combined with the capability of constructing multiple logical volumes adopting different RAID levels and stripe widths within the same physical pool, makes RAID+ especially attractive to cloud and shared datacenter environments employing large disk enclosures/trays.

We implemented a RAID+ prototype by modifying the Linux `MD` (Multiple Devices) driver, and evaluated it using a 60-drive disk enclosure. Results show that RAID+ in most cases outperforms both RAID-50 and randomized RAID-5 placement schemes, while offering faster reconstruction (2.1-7.5× over RAID-50, 1.0-2.5× over hash-based random placement). Like randomized placement, it significantly improves overall throughput in multi-tenant environments (average 2.1× over RAID-5). But unlike randomized placement, RAID+'s deterministic addressing allows simple implementation and delivers better sequential performance (for application and rebuild I/O) by guaranteeing uniform data distribution within smaller extents and retaining spatial locality.

2 RAID+ Overview

2.1 Latin Square Based Data Organization

With conventional k-disk RAID arrays, each data stripe is exactly k-block wide (including both data and parity), squarely striking through all disks. The RAID type (level) and stripe width both remain fixed throughout a given disk array. RAID+, instead, uses *Latin-square-based templates* to allocate space from a larger n-disk array. A template constructs $n\times(n-1)$ k-block stripes, each mapped to a k-subset of the n disks. Different k values and RAID types can be adopted by different templates sharing the same n disks. Like conventional RAID, RAID+ arrays can be hardware- or software-based, offered as RAID+ enclosures with special RAID adapters or formed by software on top of connected disks.

Figure 1(a) portrays conventional RAIDs, where disks are physically partitioned into two RAID groups, with potentially different RAID settings. Each disk belongs to one fixed RAID array, except the shared hot spares. One can integrate multiple underlying RAID groups (likely homogeneous in this case) into a logical volume, by *concatenating* them, or *striping* data across them. The widely adopted RAID-50, for example, belongs to the latter case. Alternatively, one can build a logical volume on each underlying RAID group, separately serving different workloads sharing the disk pool. These two options are used in our experiments for single- and multi-workload evaluation, respectively.

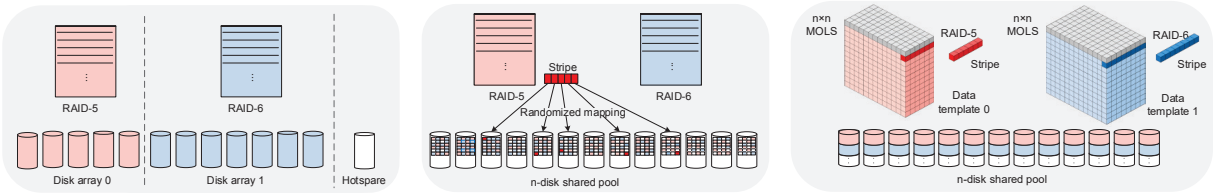
With conventional RAID organization, when a failure happens, the recovery process only involves disks within the same RAID group, reading from the $k-1$ surviving drives and reconstructing lost data on a spare drive in its entirety. As a result, the rebuild speed is capped by the slower between read and write speeds of a single disk.

Figure 1(b) shows an alternative approach, where RAID volumes are built by distributing blocks in each k-block RAID stripe to randomly selected k disks. This retains the fault tolerance of RAID yet spreads each volume to all n disks within the pool.

Figure 1(c) shows our proposed RAID+, also a flat organization of the same n-disk pool, where two data templates are used to carve space uniformly from all disks. Each template is designed by "stacking" a sequence of k $n\times n$ mutually orthogonal Latin squares, whose definition is given in the next section. Each Latin square cell stores a disk ID within $[0,n-1]$. Cells at the same location through the k layers then form a k-width data stripe (highlighted). Given such a set of k Latin-squares, one can easily compute the locations of any stripe' blocks, on a k-subset of the n drives. The mathematical properties of mutually orthogonal Latin squares guarantee the uniform data distribution on all working drives, either for normal or single-failure recovery accesses. Since data distribution by each template is always uniform, users can host different RAID organizations within the same n-drive disk pool, such as RAID-5 using the red and RAID-6 using the blue template.

When a disk failure occurs, both random placement and RAID+ allow all surviving disks to equally participate in reconstruction, cutting theoretical RAID rebuild time to $k/(n-1)$ of that of conventional RAID. Also, hot spares are optional with these organizations. However, as we shall see later in the paper, RAID+' deterministic and uniform data placement enables it to achieve perfect load balance within smaller address extents and retain spatial locality, both significant advantages (crucial to sequential access and RAID rebuild) over random schemes.

In addition, a RAID+ pool can perform in an *interim mode* with multiple disk failures, by continuously main-



(a) Conventional RAID (b) Random data placement (c) RAID+

Figure 1: Different ways of utilizing a disk pool much larger than typical RAID array sizes

	Application I/O		Rebuild I/O		
	Isol.	*Thrp*	$T_{rebuild}$	*Interf.*	*MTTDL*
RAID-5C	High	$T \cdot k$	C/B	Part-High	t
RAID-50	Low	$T \cdot (n-s)$	C/B	Part-High	t
RAID+	Low	$T \cdot n$	$C \cdot k/(B \cdot (n-1))$	Univ-Low	$> t \cdot (k-1)/k$

Table 1: Comparison of RAID-5 organizations

taining its uniform or near-uniform data distribution. In fact, RAID+ reserves space for data recovery and always performs a fast all-to-all reconstruction. When hot spares are available or failed disks are repaired, the recovered data will be replicated to replacement disk(s) in background, hiding the slow single-disk writing latency.

2.2 Comparison of RAID Usage Modes

Table 1 gives several major metrics, comparing RAID+ with common existing solutions utilizing larger disk pools. s denotes the number of hot spare disks, while C and B denote single-disk capacity and bandwidth, respectively. Without loss of generality, we use RAID-5 as the elementary RAID level. Here *RAID-5C* and *RAID-50* refer to the aforementioned "concatenated" and "striped" volume construction modes. We omit random placement schemes as they are similar to RAID+ in these aspects (but suffer from inferior load balance and locality).

For application I/O, RAID-5C has good inter-application isolation, as different workloads are more likely to involve separate underlying RAID-5 arrays, while both RAID-50 and RAID+ would be subject to performance interference from concurrent workloads. The tradeoff is aggregate performance per volume: files on RAID-5C can only utilize 1-2 physical k-disk RAID array at a time, while RAID-50 and RAID+ could enlist most or all disks. This applies to both sequential accesses (in bandwidth) and random ones (in IOPS).

For RAID rebuild I/O, both RAID-5C and RAID-50 limit reconstruction to the physical array with the disk failure. Their recovery time ($T_{rebuild}$) is equivalent to a single-disk full scan, assuming perfect read-write overlap. In contrast, RAID+ enlists all $n-1$ surviving disks in the read-write of the k-disk capacity ($C \cdot k$), making recovery itself much faster. Regarding application-perceived interference, with RAID-5C the RAID reconstruction process is only visible to accesses to the same physical array. RAID-50 gets similar partial exposure with random accesses, but could be universally affected with larger, sequential reads/writes, as shown in our evaluation (Table 3). With RAID+'s all-to-all data recovery, reconstruction traffic can be perceived by most user requests, but the interference is lighter and lasts shorter.

Finally, the *MTTDL* column describes *mean time to data loss*, considering the probability of non-recoverable failures (such as second disk failure before reconstruction completes with RAID-5). We find RAID-5C and RAID-50 with the same *MTTDL* and give a conservative lower bound for RAID+ relative to it. The bound is fairly close to 1 and configurable by k. With RAID-6, however, we found that RAID+ actually enjoys a significant improvement in *MTTDL* over conventional systems [49], by prioritizing the reconstruction of significantly fewer yet more vulnerable stripes.

3 Latin Squares for Data Distribution

We first introduce basic concepts and theorems of mutually orthogonal Latin squares (MOLS), followed by an example illustrating its use in constructing RAID+.

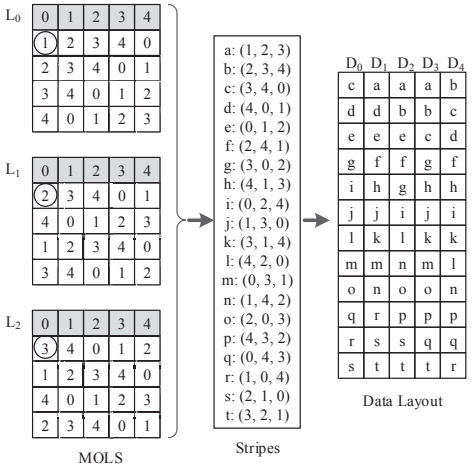


Figure 2: RAID+ layout ($n = 5, k = 3$)

Definition 1. *A Latin square of order n is an $n \times n$ array filled with n different items, each occurring exactly once in each row and column.*

Definition 2. *Let L_1 and L_2 be two n-order Latin squares. L_1 and L_2 are mutually orthogonal if, when su-*

perimposed, each of the n^2 ordered pairs occur exactly once across the overlapping cells of the two squares.

Definition 3. *Given a set of Latin squares, if all its member pairs are mutually orthogonal, we call it a set of mutually orthogonal Latin squares (MOLS).*

For example, the left side in Figure 2 gives three sample 5-order MOLS. Here the item set $I = \{0,1,2,3,4\}$ has each member appearing strictly once in any row/column of the three 5×5 squares. When any two of these MOLS are stacked together and one reads through the 25 aligned cell-pairs, each unique pair $\langle i, j \rangle (i \in I, j \in I)$ also appears exactly once.

Theorem 1. *With any given order n, there can be at most $(n-1)$ MOLS, with this upper bound achieved when n is a power of a prime number.*

Theorem 2. *When n is a power of a prime number, a complete set of $(n-1)$ MOLS can be constructed by filling the i^{th} square L_i $(0 < i < n)$ using $L_i[x,y] = i \cdot x + y$.*[1]

With such construction, the first row of all $n-1$ MOLS are identical, as shown in Figure 2. However, below the first row, the corresponding values at coordinates $[x,y]$ across all or any subset of the $n-1$ MOLS are *guaranteed* to be distinct.

Next we reuse the Latin squares shown in Figure 2 to illustrate how RAID+ works. With RAID+ templates, the order of Latin squares (n, 5 in this case) corresponds to the disk pool size. The number of Latin squares "stacked together" (k, 3 in this case) corresponds to the RAID stripe width. Suppose we now construct a logical RAID-5 volume, distributing data blocks with a stripe width of 3 across 5 disks.

We ignore the first row of all squares and construct $n(n-1)$ *stripes* by copying contents from the remaining $n(n-1)$ cells across all three squares. The middle column in Figure 2 gives a full list of these 20 stripes. The derived $n(n-1)$ stripe sequence guides block assignment onto the n disks: each item maps to the corresponding disk ID. *E.g.*, the first stripe ("stripe a") is made by looking up the $[1,0]$ cell of L_0, L_1, and L_2, resulting in tuple $\langle 1,2,3 \rangle$. Its 3 blocks (2 data and 1 parity) will thus reside on disks (1, 2, 3), respectively, while those from "stripe b" will reside on disks (2, 3, 4), and so on.

The right column in Figure 2 gives the resulted data layout from the disks' point of view. As these 20 3-block stripes guarantee a uniform distribution of disk ID numbers, the 60 blocks form a 5×12 *RAID+ template*, to be repeatedly used in distributing data to the 5 disks.

The intuition is that aside from the first row, k MOLS give us uniform and deterministic data distribution across n disks, with k-block stripes. Unlike traditional RAID systems, where the disk array size equals the stripe width, our MOLS-based design allows k (and the RAID type) to be decoupled from n, enabling the construction of different virtual RAID volumes with small or moderate stripe widths on top of much larger disk pools.

As to be discussed in more details later, another desirable feature of MOLS is that, when one of the disks fails, blocks needed to recover the lost data are also *uniformly distributed among the $n-1$ surviving disks*. This allows for quick read, reproduction, and write of the (temporarily) lost data in parallel by these $n-1$ disks.

4 Normal Data Layout

4.1 Valid Disk Pool Sizes of RAID+

The precondition of building a RAID+ system is that we can construct $(k+m)$ n-order MOLS, k of which used to construct the normal data layout and m reserved as *spare MOLS* for data redistribution in the face of disk failures (details in Section 5). m should be large enough to support the highest fault tolerance level among all RAID volumes within this RAID+ pool. For example, if a volume adopts RAID-6, then we need $m \geq 2$.

Although the number of n-order MOLS for general n remains an open problem, $(n-1)$ is known to exist when n is a power of a prime number [7]. Therefore, as long as n, the total number of disks, is one such *valid pool size*, one can perform the deterministic calculation of $n-1$ MOLS using the algorithm given in Theorem 2. Also, with these valid n values, the corresponding MOLS set possesses several attractive properties for balanced data and recovery load distribution.

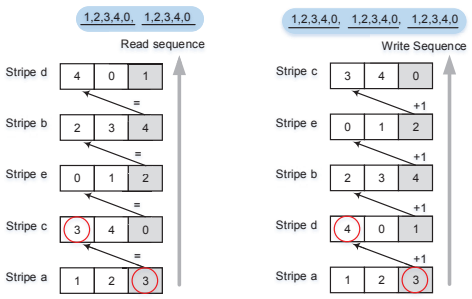
The requirement may sound demanding, but it turns out qualifying numbers are abundant and not far apart. For example, between 4 and 128, we have the following 42 valid n values: 4, 5, 7, 8, 9, 11, 13, 16, 17, 19, 23, 25 27, 29, 31, 32, 37, 41, 43, 47, 49, 53, 59, 61, 64, 67, 71, 73, 79, 81, 83, 89, 97, 101, 103, 107, 109, 113, 121, 125, 127, and 128. Since our envisioned RAID+ disk pools contain dozens of to 100+ disks, there are plenty of valid n values to choose from.

The density of valid n values further allows physical performance isolation should it be desired. Multiple RAID+ logical sub-pools can be constructed within a larger physical pool. *E.g.*, a 60-disk pool can support sub-pools with configuration (11+49), (23+37), (8+11+41), etc., all with valid sub-pool n values.

4.2 Stripes-to-Disks Mapping

RAID+ supports two modes, a *normal layout*, with guaranteed uniform distribution across all disks, and an *interim layout*, with uniform or slightly skewed data distribution among surviving disks, under one or more disk failures. Below we give more formal discussion of data

[1] "$\cdot$" and "$+$" here denote *finite field* multiplication and addition [7].



(a) Read friendly (b) Write friendly

Figure 3: Stripe order for a 5-disk array ($k = 3$). Gray indicates parity blocks. The head of each stripe equals the tail of the previous one added by 0 (read-friendly) or 1 (write-friendly).

organization with RAID+ under normal operation with a valid initial disk pool size n, with failure recovery and interim layout discussed in the next section.

Given k MOLS of order n, $\{L_0, L_1, ..., L_{k-1}\}$, a *RAID+ template* is constructed by traversing these k Latin squares simultaneously in a row-major order from the second row on. For each position $[x, y]$ ($0 < x < n$, $0 \leq y < n$), the k-block stripe $S_{x,y}$ is obtained by listing the corresponding values of L_i at this position: $S_{x,y} = \{L_0[x,y], L_1[x,y], ..., L_{k-1}[x,y]\}$, giving the disk IDs to place the k blocks of $S_{x,y}$. Since $n-1$ rows with n columns in the Latin squares are traversed, there are $n(n-1)$ stripes in a full cycle of this RAID+ template.

Below are the major properties of MOLS-based data layout. The proofs are omitted due to the space limit.

Property 1. *With normal data layout, any two blocks within a data stripe are placed on separate disk drives.*

This property guarantees that (1) the I/O workload in accessing each k-block stripe is uniformly distributed to k disks, and (2) a single disk failure results in the loss of at most one data block within any stripe.

Property 2. *With normal data layout, the n disks are assigned equal shares of both data and parity blocks.*

This property guarantees the same read-write load balancing as with RAID-5, allowing equal distribution of both data and parity blocks. This is particularly important to storage devices with asymmetric read-write performance and/or write leveling requirement, such as NAND flash disks. Unlike RAID-5, though, RAID+ decouples the stripe size k from a potentially much larger pool size n, allowing load balancing to be performed at a much wider scope, without sacrificing the fault tolerance allowed by the adopted RAID level.

4.3 Throughput-Friendly Addressing

So far, the RAID+ template gives a deterministic mapping from data blocks in any k-block stripe to n disks. However, since each stripe will be mapped to a k-subset of n disks, the ordering of the $n \times (n-1)$ stripes within the logical address space has impact on disk contention, I/O parallelism utilization, and spatial locality.

To this end, RAID+ allows stripe ordering (block addressing) to be done in different ways considering workload-specific needs. In particular, different RAID+ volumes sharing the same physical n-disk pool can each adopt its own addressing strategy. Below we describe two sample addressing algorithms targeting large sequential reads and writes, respectively (considering that block addressing matters less with random accesses). For the ease of illustration, we adopt simple RAID-4, where the first two blocks in each stripe are data blocks and the last one parity. The key difference between the two patterns here is that with RAID redundancy, sequential reads will skip parity blocks while sequential writes need to update both data and parity.

RAID+ performs stripe ordering by rows in the MOLS, with the process repeated at each row. To form the n-stripe sequence for the x^{th} row ($S_{x,0}$, $S_{x,1}$, ..., $S_{x,n-1}$), RAID+ starts by setting $S_{x,0}$ as the stripe given at the $[x,0]$ position of the MOLS, walking through the remainder of the row as follows:

- *Sequential-read friendly ordering* The head of each subsequent stripe is the tail of its predecessor (Figure 3(a)). *I.e.*, we choose $S_{x,i}$ such that $S_{x,i}(0) = S_{x,i-1}(k-1)$. The rationale here is that the last block within a RAID-4 stripe is a parity block, which will not be involved in user read operations.
- *Sequential-write friendly ordering* The head of each subsequent stripe is the sum of the tail of the previous one and x (Figure 3(b)). *I.e.*, we choose $S_{x,i}$ such that $S_{x,i}(0) = S_{x,i-1}(k-1) + x$. This is considering that for full-stripe writes resulted from sequential write workloads, all the blocks within a stripe will be updated.

Finally, such logical ordering of stripes within a RAID+ volume also corresponds to the relative ordering of blocks on each disk. *E.g.*, the middle column in Figure 2 gives a "plain" row-major stripe ordering (neither read- nor write-optimized). This ordering uniquely defines the block ordering on each of the 5 disks (the column below each disk ID). In this case, D_0 carries blocks assigned to "0": the 3rd block in stripe c, the 2nd in d, the 1st in e, ..., etc. Given n, k, the block addressing scheme, and the RAID level adopted, the logical to physical block mapping within any RAID+ template can be completed by simple calculation. Our implementation uses a tiny lookup table (sized 93KB for $n = 59$ and $k = 7$) to accelerate such in-template data addressing.

4.4 Multi-Template Storage/Addressing

One major advantage of RAID+ is to accommodate multiple virtual RAID arrays (volumes) within the same shared disk pool, each servicing different

users/workloads. Every such virtual array comes with its own MOLS-based template, stripe width k_i and RAID level, block size, as well as block addressing scheme. Within the large n-disk physical address space, capacity is allocated at the granularity of RAID+ templates.

RAID+ uses a per-volume index table to store the physical locations of its template instantiation. Since the templates are rather large, containing $n(n-1) \times k_i$ blocks for the i^{th} volume, maintaining such mapping (one "base address" per template instance) brings little overhead. Logical blocks of a given volume can then be easily mapped to its physical location by coupling the proper template instance offset with in-template addressing discussed earlier. Compared to random data distribution [12, 16, 41, 51], RAID+ offers uniform data distribution and direct addressing while only requiring single-step, template-level offset maintenance and lookup.

5 Data Recovery and Interim Layout

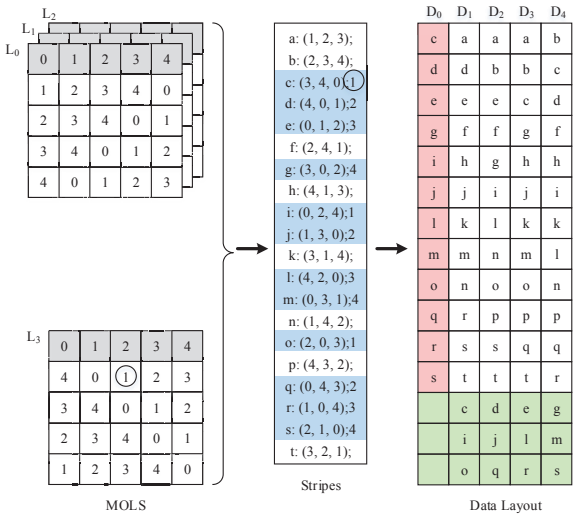


Figure 4: Interim data layout for a 5-disk RAID+ array when one disk fails. RAID+ uses the fourth MOLS, L_3, to generate uniform data distribution.

The MOLS-based RAID+ data distribution offers all-to-all fast data recovery involving all surviving disks in a disk pool, directly into an *interim layout*. When the failed disk gets repaired or replaced, the normal data layout can be restored in background.

5.1 Interim Layout under 1-disk Failure

Upon a disk failure in an n-disk pool, RAID+ performs fast data recovery to recalculate the lost blocks and distribute them to the $(n-1)$ surviving disks. Thanks again to MOLS properties, when n is a valid pool size (power of prime), the resulted *interim data layout* preserves uniform data distribution. Suppose disk D_f ($f \in [0, n-1]$) fails, below we describe the construction of the interim layout with k-block stripes, reusing the former example. Figure 4 illustrates this process, where D_f is D_0.

Let R be the set of stripes affected by D_f's failure, which contain the item f (0 in this example). For any template, there are a total of $n(n-1)k/n = (n-1)k$ blocks on each disk. As each stripe cannot have two blocks assigned to the same disk, these $(n-1)k$ blocks correspond to the same number of stripes that are involved in data recovery, as shown in the middle column of Figure 4.

Recall that a valid pool size n allows for $(n-1)$ n-order MOLS, out of which k are used for the normal layout. Now we select any one from the remaining $(n-k-1)$ MOLS, as L_k (Latin square L_3 in Figure 4). This additional Latin square will be used to guide the placement of blocks assigned to D_f, with the item f in the affected stripes replaced with a new surviving disk ID.

The intuition is that when we append the new Latin square to the back of the existing k MOLS "stack" and read through each position $[x, y]$, we extend the k-block stripes to $(k+1)$-block ones, with again uniformly distributed item-set permutations. Now take each affected stripe, and replace the (now missing) f with the item r at the corresponding position in L_k, we relocate the missing block used to be assigned to D_f to the surviving disk D_r. E.g., in Figure 4, each "0" in these 12 stripes would be replaced with another integer in $\{1,2,3,4\}$, such as stripe c transforming from $(3,4,0)$ to $(3,4,1)$, as the corresponding position in the additional Latin square ($L_3[1,2]$) has item "1". The first two blocks in this affected stripe, on D_3 and D_4 respectively, would not need to move.

Below are the major properties associated with MOLS-based data layout concerning data recovery and the interim layout, assuming a valid pool size n.

Property 3. *With the n-disk normal layout, all blocks correlated with those on any given drive (i.e., blocks sharing stripes with blocks on this disk) are distributed evenly among the other disks.*

The implication here is that when the first disk fails, the read workload to recover unavailable blocks is evenly distributed among all surviving disks.

Property 4. *With the $(n-1)$-disk interim layout, any two blocks within a data stripe are still placed on separate disk drives.*

Property 5. *All the $(n-1)k$ missing blocks on any single failed disk can be redistributed to all the surviving $(n-1)$ disks evenly, each receiving k additional blocks.*

These two properties imply that (1) the write workload involved in RAID+ recovery from a single-disk failure is also uniformly distributed among all surviving disks, (2) data stripes in the $(n-1)$-disk interim layout preserves the same RAID fault tolerance as in the normal layout,

and (3) the $(n-1)$-disk interim layout also retains the uniform data distribution to allow perfectly balanced I/O servicing even after losing one disk.

The particular significance of $(n-1)$-disk interim layout lies in the fact that the probability of single-disk failure is much higher than that of having two or more failed disks, especially when hot spares are available. Considering this, plus that disk capacity is relatively abundant in typical server environments, RAID+ performs an additional performance optimization by reserving recovery data space with normal data layout. More specifically, RAID+ actually allocates $n \times k$ physical blocks per disk for a data template. $(n-1)k$ of them are used to store data/parity blocks in the normal layout, while the remaining k blocks (the green area in Figure 4) are reserved for storing reconstructed data whenever there is a single-disk failure. This way, under such a failure, the reconstructed data are physically adjacent to the normal layout blocks, preserving spatial locality in data accesses. In our implementation, the content of the aforementioned small lookup table is modified to support fast interim data addressing, without additional space overhead.

5.2 Parallel Data Recovery

Under a single disk failure, the MOLS-based design lets both read and write workloads involved in RAID reconstruction and temporary relocation be uniformly distributed to the entire pool. This breaks the performance limit of conventional RAID systems, where the recovery work is only distributed within the RAID array affected.

However, even with uniform data distribution, the parallel read/write operations in data recovery could still generate resource contention, transient load imbalance, or unnecessary disk seeks, if care is not taken. To this end, RAID+ orchestrates its all-to-all data reconstruction by letting the surviving disks work on a subset of the $(n-1)k$ affected stripes at a time, alternating between reader and writer roles. Barriers are used in between such iterations, creating a natural break point for RAID+ to check upon user I/O requests, potentially slowing down or temporarily suspending the recovery depending on the current application request intensity, QoS specifications, and configurable system policies (such as starvation prevention to ensure the completion of data recovery).

5.3 Multiple Disk Failures

MOLS-based design also handles multiple failures gracefully. If another disk failure occurs after data recovery from a disk failure, we repeat the process described in Section 5.1 with another spare Latin square. When m disks are lost but tolerated by the adopted RAID level, by appending m spare MOLS to the stack of k used in the normal layout, we can calculate the eventual $(n-m)$-disk interim layout. Recognizing that the *affected* data stripes have different degrees of data loss, RAID+ prioritizes the reconstruction of the more vulnerable stripes.

Due to space limit, we give a brief summary of related results: when a RAID+ pool keeps losing disks (without disk replacement), Monte Carlo simulation shows very slight imbalance in data distribution (CoV of up to 0.29%), while system experiments show application performance degradation of up to 6% (except with sequential read, where RAID+ loses the benefit of its unique read-friendly addressing when more disks fail).

6 Evaluation

We implemented RAID+ in the `MD` (Multiple Devices) driver in Linux Kernel 3.14.35, a software RAID system that forms a common framework for all RAID systems tested in our evaluation. Despite theoretical properties appearing sophisticated, MOLS-based addressing is simple to implement, taking a mere 12 lines of code.

Test platform Our testbed uses a SuperMicro 4U storage server with two 12-core Intel XEON E5-2650 V4 processors and 128GB DDR4 memory, running Ubuntu 14.04 with Linux kernel v3.14.35. Two AOC-S3008L-L8I SAS JBOD adapters, each connected to a 30-bay SAS3 expander backplane via two channels, host 60 Seagate Constellation 7200RPM 2TB HDDs. The I/O channels afford a total I/O bandwidth of 24GB/s (400MB/s for each disk), significantly exceeding the aggregate sequential bandwidth from the disks. In all experiments, 50GB capacity of each disk is used.

RAID configurations Unless otherwise noted, our tests use 59 out of the aforementioned 60-disk pool ($n = 59$). The stripe width k is set at 7 (6+1 RAID).

For comparison, we evaluate two commonly adopted conventional RAID organizations utilizing such large disk pools, both of which build eight 6+1 RAID-5 arrays with 64KB stripe unit size, consuming 56 disks with the last 3 reserved as hot spares. *RAID-5C* divides each array into multiple 1GB extents and concatenates them in a round-robin manner, while *RAID-50* stripes across these 8 arrays at block size of 12MB (2MB$\times$6).

We also evaluate two randomized data placement schemes, RAID_R and RAID_H. Both place blocks within each stripe to different disks while aiming for balanced data distribution to all n disks. To assign a block to a disk, RAID_R utilizes the system random number generator (with system time as seed) and is therefore nondeterministic. RAID_H, instead, uses the Jenkins hash function [26] adopted by systems such as Ceph [9, 50]. If a block is mapped to a disk already used in the current stripe, mapping will be recalculated until collision free (following CRUSH [51] for RAID_H). Our experiments find the two schemes often perform similarly, in which case we show only results of the better one.

# of disks	RAID-50	RAID_R	RAID_H	RAID+
$56+3$	$307s$	$60s$	$102s$	$41s$
$28+3$	$307s$	$99s$	$143s$	$83s$

Table 2: Offline rebuild time comparison

Both RAID+ and the random schemes build (6+1) RAID-4 arrays, with 64KB stripe unit size. Such striping continues on the same set of disks until a 2MB space allocation unit is filled per disk, before starting a new 7-block stripe. We have also implemented RAID-6 and observed similar performance trends, but omit results here due to space limit.

I/O Workloads We use three types of I/O workloads:

- *Synthetic workloads:* we use `fio` [15], a widely used I/O workload generator to produce four representative elementary workloads: sequential read, sequential write, random read, and random write.
- *I/O traces:* We use 8 public block-level I/O traces, namely `src1_1`, `usr_1`, `prn_0`, `prn_1`, `proj_0`, and `prxy_1` from MSR Cambridge [37], plus `Fin2` and `WebS_2` from SPC [1]. Based on load level observed, we followed existing practice in prior research [17, 27, 48] and accelerated the SPC traces (`Fin2` by $5\times$ and `WebS_2` by $3\times$) while replaying all others a tempo.
- *I/O-intensive applications:* we also use four I/O-heavy real applications: GridGraph [60] (an out-of-core graph engine), TPC-C [46] (in-house implementation of the well-known RDBMS transaction benchmark standard), a Facebook-like photo access workload (synthesized using Facebook's published workload characteristics [4, 24]), and a MongoDB [35] NoSQL workload from the YCSB suite [11].

6.1 Reconstruction Performance

We start by evaluating reconstruction performance, one major advantage of RAID+ over alternative schemes, with disk failures created by unplugging random disk(s).

Offline rebuild Table 2 gives the offline rebuild time with a single-disk failure. RAID+ is tested with two valid n values, 59 and 31 (shared by the random schemes). The same pools would give RAID-50 3 hotspares each, with 8 and 4 RAID-5 arrays respectively. RAID-5C results would be identical to RAID-50 here.

In both cases, RAID+ consistently outperforms RAID-50, delivering a speedup of $7.5\times$ and $3.7\times$. Such results approach the theoretical speedup of 8.29 and 4.29, respectively, given in Table 1. The gap is mainly due to less sequential reconstruction read/write patterns compared with RAID-50, as RAID+'s recovery load per disk is much smaller yet non-contiguous. Unlike RAID-50, with rebuild time independent of the disk pool size, RAID+ spreads the rebuild workload to larger pools uniformly and lowers the rebuild time proportionally.

The two random schemes outperform RAID-50 here also by having more disks participate in recovery. However, their rebuild takes significantly longer than that of RAID+. Further examination reveals that though they achieve overall balanced data distribution, random schemes suffer much higher skewness within each window of dozens/hundreds of blocks. E.g., within a RAID+ template size, the RAID_H has CoV of rebuild read/write load distribution of 31.9%/38.9%, while RAID_R has 12.72%/33.87%.[2] Such "local load balance" is crucial for RAID rebuild, with sequential and synchronized operations, where overloaded stragglers could easily drag down the entire array's recovery progress. RAID+, in contrast, retains its absolute load balance within such smaller windows and delivers much higher rebuild speed.

Finally, this advantage grows with the disk pool size n, as such perfect local load balance gives RAID+ higher profit margin by evenly utilizing n disks. In this sense, RAID-50 has recovery bandwidth independent of n by utilizing a small fixed-size sub-pool. The random schemes perform between these two extremes, achieving good global yet poor local load balance.

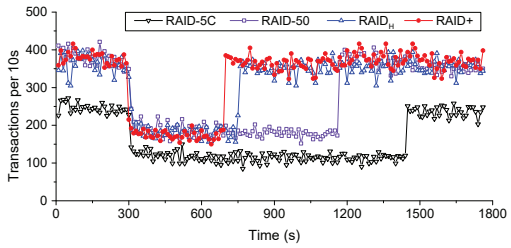


Figure 5: TPC-C online rebuild w. single-disk failure

Single-workload online rebuild Next, we examine online rebuild by creating a single-disk failure and performing reconstruction without stopping the execution of application(s). Figure 5 illustrates one sample test case (TPC-C). It plots the number of transactions committed per 10-second episode along the timeline, with a disk failure incurred at 300 seconds into the execution.

First, results demonstrate that RAID+ matches the `TPC-C` throughput of RAID-50 and RAID_H (all beating RAID-5C, unsurprisingly) in normal operation. Second, RAID+ offers much shorter online rebuild time than conventional RAID (396 seconds vs. RAID-5C's 1137 and RAID-50's 858). RAID_H comes closer, but still takes 11.4% longer than RAID+. Third, RAID+, RAID-50, and RAID_H bring similar degrades to TPC-C performance during rebuild. Although RAID-5C sees smaller relative performance impact, its degraded performance still lags behind due to its lower baseline. Overall,

[2] Here RAID_R has more even read distribution due to larger read volume than write in rebuild. RAID_H however exhibits more skewed distribution of blocks to read involved in recovery, with CoV level appearing to be dependent on the hash function used.

App		RAID-50	RAID-5C	$RAID_H$	RAID+
FaceBook	*app perf*	1	1.02	1.41	1.42
	reb perf	1	1.05	2.29	2.36
TPC-C	*app perf*	1	0.61	1.00	1.03
	reb perf	1	0.75	1.94	2.17
GridGraph	*app perf*	1	0.27	1.22	1.23
	reb perf	1	3.14	2.05	2.06
MongoDB	*app perf*	1	0.89	0.99	1.01
	reb perf	1	1.05	1.60	2.08

Table 3: Online rebuild performance comparison, in terms of speedup against corresponding RAID-50 results

during the 900 seconds following the disk failure's onset, RAID+ manages to complete 44.43%, 139.70%, and 5.11% more transactions than RAID-50, RAID-5C, and $RAID_H$, respectively. `TPC-C` throughput stays consistent as recovery progresses, as its degradation is dominated by the rebuild I/O activities rather than transactions that happen to hit the failed disk.

Table 3 summarizes online reconstruction performance, giving both the application performance and the rebuild speed, all in the form of speedup with respect to corresponding RAID-50 results (the higher the better). We use the same RAID rebuild rate setting (minimal at 80MB/s and maximum at 200 MB/s) within the MD driver for RAID-50 and RAID-5C, and configure RAID+ and $RAID_H$ to avoid application performance degradation from the RAID-50 baseline during rebuild (with only one exception where $RAID_H$ achieves 99% of the baseline performance). The results reveal that RAID+ and $RAID_H$ simultaneously improve both the application and rebuild performance from RAID-50. Between them, RAID+ is consistently better, with significantly faster rebuild and slightly better application performance for TPC-C and NoSQL. RAID-5C, at least with the default rate control setting, loses on both fronts (except for FaceBook, where it slightly outperforms RAID-50).

Multi-workload online rebuild For multi-workload evaluation, we use a smaller pool size of 29,[3] with stripe width remaining at 7, to construct 4 logical RAID volumes. RAID-5 builds 4 disjoint 6+1 arrays, plus one last disk reserved as hot spare. RAID+ constructs 4 volumes with the same deterministic template ($n = 29, k = 7$) across the entire pool. $RAID_H$ randomly distributes blocks from 4 virtual 6+1 array volumes to all 29 disks.

In each experiment, we sample 4 out of 8 MSR/SPC I/O traces as a *workload mix* to run simultaneously on the RAID volumes, for 28 minutes. The requests are replayed using the original timestamps, therefore identical sets of requests are issued across tests. We create a single-disk failure in the whole pool at time 0 and perform reconstruction without stopping user applications.

[3]Considering the moderate request levels in test programs/traces, the smaller pool size allows us to test smaller (and higher number of) workload mixes, with results easier to plot and analyze.

Figure 6 illustrates one such test case, showing the average I/O request latency in 60-second episodes along the execution timeline, for each workload. With RAID-5, the failure is contained within one volume (running `Fin2` in Figure 6(a)), while with other schemes, it affects all volumes. The vertical lines indicate time points when each scheme finishes online rebuild. Similar to single-workload results, RAID+ has slightly faster rebuild than $RAID_H$, both beating RAID-5 by almost 4 times. Intuitively, RAID+ and $RAID_H$ excel by spreading rebuild work to all 4 volumes rather than only one, which also enables them to eliminate dramatic latency increases brought by RAID-5's online rebuild (Figure 6(a)). Thus compared with RAID-5, during the entire reconstruction, RAID+ and $RAID_H$ reduces the `Fin2` workload average latency by over 90%, and the 99% tail latency by 89% (48*ms* vs. 418*ms*).

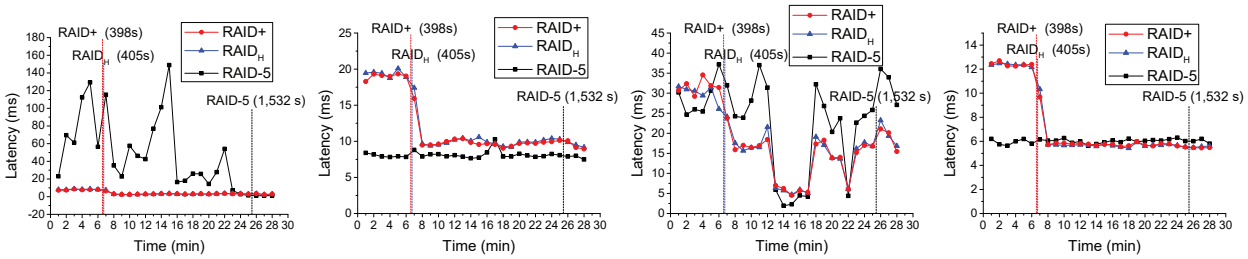
As expected, involving all disks expose the failure to all 4 volumes, roughly doubling the average latency of RAID+/$RAID_H$ before rebuild completes over RAID-5 for the `prxy_1` and `WebS_2` workloads. However, the much shorter rebuild time not only reduces system vulnerability, but also prevents any volume to be under dramatic performance degradation for prolonged periods. Note that while inter-volume isolation is broken here, disk failures are not user application artifacts but anomalies from the underlying platform. Therefore, RAID+ allows a larger disk pool to become more resilient, recover faster from failures, and provide more consistent performance during recovery.

6.2 Normal I/O Performance

Single-workload evaluation Figure 7 gives the normal synthetic workload performance running `fio`, with varying request sizes. The access footprint is large enough to span all RAID-5 arrays with RAID-5C. All RAID systems, including RAID+, perform very similarly in random read/write tests. Therefore, we only show sequential performance here.

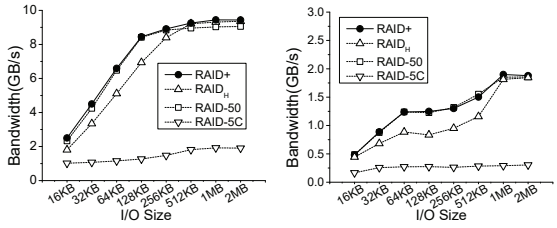
RAID+ slightly outperforms RAID-50 in most cases, by using 59 rather than 56 disks. Compared to them, $RAID_H$ offers moderately lower sequential performance, again due to poor local load balance and inferior spatial locality within each disk. RAID-5C predictably lags behind others with sequential accesses, as in most cases only one RAID-5 array is utilized.

Figure 8 shows results with two sample MSR traces, plotting latency data points (averaged over 60-second episodes) along the execution timeline. Again RAID+ outperforms RAID-50, with an average improvement of 6.23% under `prxy_1` and 87.04% under `usr_1`. This is because RAID+ uses all 59 disks (rather than 56) and `usr_1` is read-dominant [37], with RAID+ adopts read-friendly addressing in this set of tests. For similar



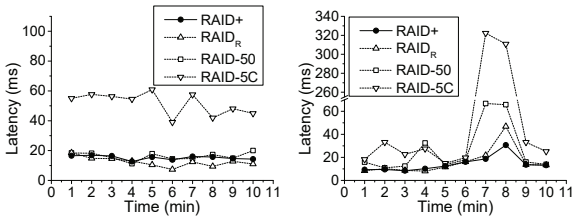
(a) `Fin2` (b) `prxy_1` (c) `src1_1` (d) `WebS_2`

Figure 6: Sample multi-workload performance w. online rebuild



(a) Sequential read (b) Sequential write

Figure 7: Normal `fio` sequential performance



(a) `prxy_1` (b) `usr_1`

Figure 8: Normal trace workload performance

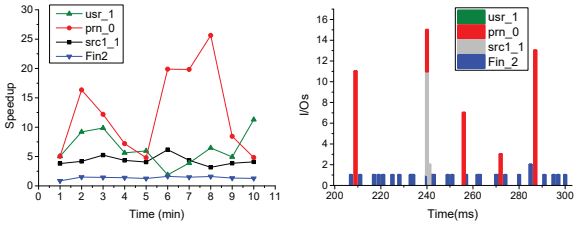
reasons, RAID$_R$ loses slightly to RAID+ under `usr_1`, and wins slightly under `prxy_1`, as it also uses all disks and the read-friendly addressing may bring minor side-effects for the more write-intensive `prxy_1` workload.

Finally, Table 4 compares application performance. Note that unlike other cases, TPC-C uses transactions per minute committed, the higher the better. With Facebook-like photo and MongoDB, both having primarily random accesses, all four RAID organizations have data distributed to all disks and report very similar performance results. Since GridGraph is primarily sequential, RAID-5C can mostly utilize only one or two underlying RAID-5 arrays. Therefore, all three other schemes have a more than 4-fold speedup over RAID-5C, and RAID+ has minor advantage over RAID-50 by using slightly more disks, and over RAID$_H$ by having better spatial locality. With TPC-C, which has both random and sequential accesses, RAID+ slightly outperforms both RAID-50 and RAID$_H$. Again RAID-5C clearly underperforms.

Multi-workload system throughput Now we examine normal performance with multiple workloads sharing the

	RAID-5C	RAID-50	RAID$_H$	RAID+
FaceBook (*s*)	168.18	165.85	176.20	168.8
MongoDB (*s*)	160.85	150.76	147.02	147.34
GridGraph (*s*)	1021.86	236.96	236.85	220.98
TPC-C (*TpmC*)	1345.2	2193	2192	2265

Table 4: Normal application performance



(a) RAID+ speedup over RAID-5 (b) Sample load variation

Figure 9: Case study with sample 4-workload mix

underlying disk pool, using the 29-disk, 4-volume setting similar to that in online reconstruction tests (Figure 6). We evaluated all unique 4-workload combinations from the 8 MSR/SPC traces, executing each of these 70 workload mixes on 4 RAID volumes built with RAID-5, RAID$_H$, and RAID+.

Here we adopt *weighted speedup* [14], a widely-used multi-workload performance metric in computer architecture, to measure the overall system throughput. As each workload is replayed at fixed speed (by timestamps given in traces), we use $1/latency$ to replace the typical IPC (Instructions Per Cycle) measurement in architecture studies, calculating the weighted speedup as $\frac{1}{n}\sum_{i=1}^{n}(L_i^c/L_i)$ for an n-application workload mix. Here L_i^c and L_i denote the average latency of the ith workload using conventional RAID (RAID-5) and the system to be evaluated, respectively. *I.e.*, RAID-5 is used as the baseline for measuring performance speedup.

To summarize the results, both RAID$_H$ and RAID+ deliver considerable weighted speedup in all 70 test cases, demonstrating their capability of consistently improving the overall system throughput by utilizing more disks simultaneously. More specifically, RAID$_H$ obtains an average weighted speedup of 1.83 (over 1.33 in 80% of cases) over the 4-volume RAID-5 baseline, while

RAID+ performs better, with average weighted speedup of 2.05 (over 1.5 in 80% of cases).

Figure 9 showcases one sample test case (running `src1_1`, `usr_1`, `prn_0`, and `Fin2`), showing the speedup (based on average latency in each one-minute episode) of RAID+ over RAID-5 along the timeline. All but one speedup data points are above 1, with `prn_0` reaching over 25 at one point. By zooming into the request patterns, we find such large profit comes from the burstiness in most workloads. Figure 9(b) illustrates this for a 100ms-long window, 200ms into the execution, showing the per-ms request count for each workload. At this granularity, one clearly sees that the workloads have sporadic requests and often form interleaving bursts. The most bursty workloads, `prn_0` and `src1_1`, benefit more from RAID+ and RAID_H, which use all disks to serve each volume. In particular, during request peaks these workloads see faster processing and lower I/O queue wait time, as confirmed by our detailed profiling, hence achieving the 25× (transient) speedup.

While the majority of the 70 mixes possess such "complementary" request patterns, there are cases where two or more workloads have sustained simultaneously intensive I/O activities, leading to slowdown of individual workload. However, among the total of 280 executions (70 mixed runs with 4 workloads per mix), there are only 39 cases of slowdown. Again, if a workload has to be guaranteed stronger performance isolation, RAID+ pools can be physically partitioned (such as building 41+19 volumes within a 60-disk enclosure).

6.3 Sensitivity to Internal Parameters

Finally, we study the impact of RAID+'s key parameters. Figure 10(a) shows both the aggregate random read and write throughput (left y axis) and the offline rebuild time (right y axis) with RAID+ pool size n, increased from 41 to 59, while fixing k at 7 and block size at 2MB.

These results show that the random read performance increases linearly with n (by up to 39%), due to uniform load distribution to all disks in the pool. The write throughput, though also growing steadily (by up to 24%), is much lower, as each of these 64KB write will bring at least four underlying I/O operations, for reading and writing back both the concerned data and parity blocks. In addition, such read-modify-write operations are synchronized, further lowering the aggregate throughput. The offline rebuild time, unsurprisingly, decreases as n grows and conforms to the model shown in Table 1.

Figure 10(b) shows similar experiments, with n fixed at 59 and varying k. With regard to user I/O performance, as modeled in Table 1, the aggregate throughput is mostly independent of k and the rebuild time grows linearly with it. One unexpected exception is with $k=3$, where the write bandwidth appears considerably higher than any other k values. By using the `iostat` tool, we find that with $k=3$ there are significantly fewer disk reads for parity calculation. Here with the 2+1 data-parity setup, there are higher chances for parity data to be reconstructed from cached data blocks.

Next, in Figure 10(c) we fix both n (59) and k (7) and change the block size. As expected, the block size has little impact on the random read/write performance. Meanwhile, the rebuild time decreases significantly, though not linearly. As RAID+'s rebuild access pattern introduces less regular access patterns compared with those of conventional RAID systems, larger block sizes improve performance by promoting sequential accesses.

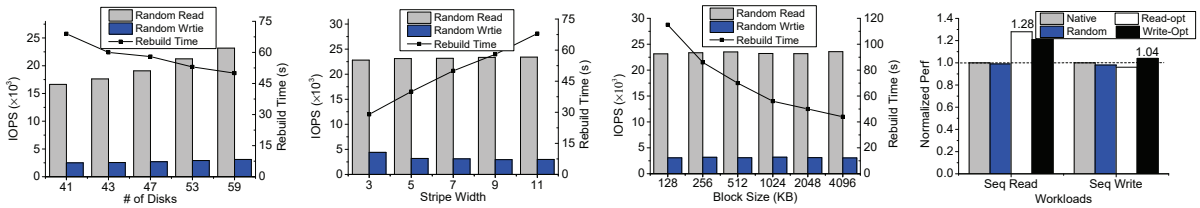
Last, to examine the effect of throughput-friendly block addressing (Section 4.3), we run the `fio` sequential read and write workloads, with I/O sizes of 2MB. Figure 10(d) shows the results using four stripe ordering strategies: 1) "native", original stripe ordering from a RAID+ template (stripes a to t in Figure 2), 2) "random", randomized stripe ordering using a pseudo-random function, 3) "read-opt", our proposed read-friendly ordering, and 4) "write-opt", our proposed write-friendly ordering. Bandwidths shown are normalized to "native". While the native and randomized strategies almost perform identically, the read-friendly strategy does generate a 28% improvement with sequential reads. Write-friendly ordering, on the other hand, brings a much smaller profit (4%). Again, unlike the "pure" sequential streams with reads, writes are not exactly sequential due to read-modify-write of both data and parity blocks.

7 Related Work

Data Layout Optimization Existing RAID layout optimizations roughly form two categories: 1) distributing either data blocks or parity blocks evenly across all the disks (*e.g.*, RAID-5 vs. RAID-4), and 2) exploiting spatial data locality (*e.g.*, left-symmetric RAID-5 [30]). Inspired by them, RAID+ spreads data and parity blocks in a much larger shared pool, with its throughput-friendly block addressing promoting access locality.

The parity declustering layout [36] utilizes as few disks as possible in data reconstruction, further analysed and extended/optimized by many [2, 10, 18, 20]. However, unlike with RAID+, rebuild is still capped by the write speed of the replacement disk, though these solutions do spread rebuild reads to remaining disks.

ZFS [6] uses "dynamic striping" to distribute load across "virtual devices", dynamically adjusting striping width and device selection to facilitate fast out-of-place updates and balanced capacity utilization. Systems such as IBM XIV [25] and the Flat Datacenter Storage (FDS) [38] use pseudo-random algorithms to distribute replicated data across all drives. If used for build-



(a) Different disk numbers (b) Different stripe widths (c) Different block sizes (d) Different stripe orders

Figure 10: Impact of several factors on RAID+'s performance

ing logical RAID volumes, in stripe placement ZFS and FDS would follow round-robin order (with optimizations on starting point selection), resulting in limited recovery bandwidth as a fixed "neighborhood" of disks would carry data relevant to recovery. XIV behaves similar to the $RAID_R$/$RAID_H$ schemes we evaluated.

Work also exists on designing data organizations sensitive to workload characteristics or application scenarios. *E.g.*, Disk Caching Disk (DCD) [23, 39] uses an additional disk as a cache to convert small random writes into large log appends. HP's AutoRAID [53] partitions RAID storage and differentiates the handling of hot and cold data. ALIS [22] and BORG [5] reorganize frequently accessed blocks (and block sequences) to place them sequentially in a dedicated area. These techniques are orthogonal to ours and can be incorporated by RAID+ to improve application performance.

Optimizations on RAID Reconstruction Prior studies have targeted improving reconstruction performance. Many of them focus on designing better data layout in a disk group [18, 29, 33, 52, 56], to minimize I/O for recovery or distribute rebuild I/O as evenly as possible. Other approaches optimize the RAID reconstruction workflow to make full use of higher sequential bandwidth, such as DOR (Disk-Oriented Reconstruction) [20], PR [31], and others [18, 42, 52, 55, 56]. In addition, PRO [45] rebuilds frequently-accessed areas first and S^2-RAID [48] optimizes reads and writes separately for faster recovery. Finally, task scheduling techniques optimize reconstruction rate control [32, 44, 47].

Except for WorkOut [54], which outsources part of user requests to surrogate disks during reconstruction, existing studies focus on improvement within one RAID group. RAID+ takes a different path from all, with built-in "backup" layouts to utilize all disks in a larger pool in reconstruction, while maintaining the fault tolerance and flexibility of smaller, logical RAID arrays.

RAID Scaling Adding disks to an array requires data movement to regain uniform distribution. Zhang et al. proposed batch movement and lazy metadata update to speed up data redistribution [57, 58]. FastScale [59] uses a deterministic function to minimize data migration while balancing data distribution. CRAID [34] uses a dedicated caching partition to capture and redistribute only hot data to incremental devices.

Another approach is *randomized RAID*, which randomly chooses a fraction of blocks to be moved to newly added disks. Prior work to this end [8, 16, 41] reduces migration, but produces unbalanced distribution after several expansions [34]. Also, existing randomized RAID systems require extra book keeping and look-up.

RAID+, in contrast, allows large disk enclosures to directly host user volumes, each using its own RAID configuration, with templates stamping out allocations in all shapes and sizes. Meanwhile, it is not designed for dynamic, heterogeneous distributed environments targeted by methods like CRUSH [51].

8 Conclusion

This paper proposes RAID+, a new RAID architecture that breaks the resource isolation between multiple co-located RAID volumes and allows the decoupling of stripe width k from disk group size n. It uses a novel Latin-square-based data template to guarantee uniform and deterministic data distribution of k-block stripes to all n disks, where n could be much larger than k. It also delivers near-uniform distribution in both user data and RAID reconstruction content even after one or several disk failures, as well as fast RAID rebuild.

With RAID+, users can deploy large disk pools with virtual RAID volumes constructed and configured dynamically, according to different application demands. By utilizing all disks evenly while maintaining spatial locality, it enhances both multi-tenant system throughput and single-workload application performance.

9 Acknowledgment

We thank all reviewers for their insightful comments, and especially our shepherd Gala Yadgar for her guidance during our camera-ready preparation. We also thank Mu Lin and Xiaokang Sang for helpful discussions. This work was partially supported by the National Natural Science Foundation of China (under Grant 61672315) and the National Grand Fundamental Research 973 Program of China (under Grant 2014CB340402).

References

[1] Umass trace repository. http://traces.cs.umass.edu/index.php/Storage/Storage, 2017.

[2] Guillermo A. Alvarez, Walter A. Burkhard, Larry J. Stockmeyer, and Flaviu Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of 25th International Symposium on Computer Architecture (ISCA'98)*, pages 109–120, 1998.

[3] Eitan Bachmat and Jiri Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'02)*, pages 55–65, 2002.

[4] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, pages 47–60, 2010.

[5] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization and self-optimization in storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, pages 183–196, 2009.

[6] Jeff Bonwick and Bill Moore. ZFS: The last word in file systems. https://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf, 2007.

[7] Raj Chandra Bose and Sharadchandra S Shrikhande. On the construction of sets of mutually orthogonal Latin squares and the falsity of a conjecture of Euler. *Transactions of the American Mathematical Society*, 95(2):191–209, 1960.

[8] André Brinkmann, Kay Salzwedel, and Christian Scheideler. Efficient, distributed data placement strategies for storage area networks. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*, pages 119–128, 2000.

[9] Ceph. libcrush. https://github.com/ceph/libcrush, 2017.

[10] Siu-Cheung Chau and Ada Wai-Chee Fu. A gracefully degradable declustered RAID architecture. *Cluster Computing*, 5(1):97–105, 2002.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SOCC'10)*, pages 143–154, 2010.

[12] Netapp Corporation. How long does it approximately take for a RAID reconstruction? https://kb.netapp.com/support/s/article/ka21A0000000jOzQAI/how-long-does-it-approximately-take-for-a-raid-reconstruction?language=en_US, 2017.

[13] Oracle Corporation. A better RAID strategy for high capacity drives in mainframe storage. http://www.oracle.com/technetwork/articles/systems-hardware-architecture/raid-strategy-hi-capacity-drives-170907.pdf, 2013.

[14] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3), 2008.

[15] fio. https://github.com/axboe/fio, 2017.

[16] Ashish Goel, Cyrus Shahabi, Shu yuen Didi Yao, and Roger Zimmermann. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 473–482, 2002.

[17] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 243–256, 2017.

[18] Mark Holland and Garth A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, pages 23–35, 1992.

[19] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *Proceedings of The Twenty-Third International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 422–431, 1993.

[20] Mark C. Holland. *On-line data reconstruction in redundant disk arrays*. PhD thesis, Pittsburgh, PA, USA, 2001.

[21] Robert Y. Hou, Jai Menon, and Yale N. Patt. Balancing I/O response time and disk rebuild time in a RAID5 disk array. In *Proceedings of the Twenty-sixth Hawaii International Conference on System Sciences (HICSS-26)*, pages 70–79 vol.1, 1993.

[22] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The automatic improvement of locality in storage systems. *ACM Transactions on Computer Systems (TOCS)*, 23(4):424–473, November 2005.

[23] Yiming Hu and Qing Yang. DCD - Disk Caching Disk: A new approach for boosting I/O performance. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA'96)*, pages 169–178, 1996.

[24] 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 Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 167–181, 2013.

[25] IBM. IBM XIV storage system architecture and implementation. `http://www.redbooks.ibm.com/redbooks/pdfs/sg247659.pdf`, 2017.

[26] Robert J. Jenkins. Hash functions for hash table lookup. `http://burtleburtle.net/bob/hash/evahash.html`, 1997.

[27] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, pages 61–74, 2014.

[28] Hannu H. Kari, Heikki K. Saikkonen, Nohpill Park, and Fabrizio Lombardi. Analysis of repair algorithms for mirrored-disk systems. *IEEE Transactions on Reliability*, 46(2):193–200, 1997.

[29] Osama Khan, Randal Burns, James Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, pages 251–264, 2012.

[30] Edward K. Lee and Randy H. Katz. The performance of parity placements in disk arrays. *IEEE Transactions on Computers (TOC)*, 42(6):651–664, Jun 1993.

[31] Jack Y. B. Lee and John C. S. Lui. Automatic recovery from disk failure in continuous-media servers. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 13(5):499–515, 2002.

[32] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David Nagle, and Erik Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI'00)*, pages 87–102, 2000.

[33] Jai Menon and Dick Mattson. Distributed sparing in disk arrays. In *Digest of Papers COMPCON Spring 1992*, pages 410–421, 1992.

[34] Alberto Miranda and Toni Cortes. CRAID: online RAID upgrades using dynamic hot data reorganization. In *Proceedings of the 12th USENIX conference on File and Storage Technologies (FAST'14)*, pages 133–146, 2014.

[35] MongoDB. `https://www.mongodb.com/`, 2017.

[36] Richard R. Muntz and John C. S. Lui. Performance analysis of disk arrays under failure. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB'90)*, pages 162–173, 1990.

[37] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10:1–10:23, 2008.

[38] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 1–15, 2012.

[39] Tycho Nightingale, Yiming Hu, Qing Yang, Tycho Nightingale Y, Yiming Hu Z, and Qing Yang Y. The design and implementation of a DCD device driver for Unix. In *Proceedings of the 1999 USENIX Technical Conference (ATC'99)*, pages 295–308, 1999.

[40] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM International Conference on Management of Data (SIGMOD'88)*, pages 109–116, 1988.

[41] Beomjoo Seo and Roger Zimmermann. Efficient disk replacement and data migration algorithms for large disk subsystems. *ACM Transactions on Storage (TOS)*, 1(3):316–345, August 2005.

[42] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'04)*, pages 15–30, 2004.

[43] Marc Staimer and Antony Adshead. Post-RAID alternatives address RAID's shortcomings. `http://www.computerweekly.com/feature/Post-RAID-alternatives-address-RAIDs-shortcomings`, 2010.

[44] Eno Thereska, Jiri Schindler, John S. Bucy, Brandon Salmon, Christopher R. Lumb, and Ganger R. Ganger. A framework for building unobtrusive disk maintenance applications. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, pages 213–226, 2004.

[45] Lei Tian, Dan Feng, Hong Jiang, Ke Zhou, Lingfang Zeng, Jianxi Chen, Zzhikun Wang, and Zhenlei Song. PRO: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, pages 277–290, 2007.

[46] tpcc mysql. `https://github.com/Percona-Lab/tpcc-mysql`, 2017.

[47] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Ganger R. Ganger. Argon: Performance insulation for shared storage servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, pages 5–5, 2007.

[48] Jiguang Wan, Jibin Wang, Changsheng Xie, and Qing Yang. S^2-RAID: Parallel RAID architecture for fast data recovery. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(6):1638–1647, 2014.

[49] Zhufan Wang. Reliability analysis on RAID+. `https://github.com/RAIDPLUS/Additional-materials/raw/master/reliability.pdf`, 2018.

[50] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 307–320, 2006.

[51] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*, 2006.

[52] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A. Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, pages 2:1–2:17, 2008.

[53] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer System (TOCS)*, 14(1):108–136, February 1996.

[54] Suzhen Wu, Hong Jiang, Dan Feng, Lei Tian, and Bo Mao. WorkOut: I/O workload outsourcing for boosting RAID reconstruction performance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, pages 239–252, 2009.

[55] Tao Xie and Hui Wang. MICRO: A multilevel caching-based reconstruction optimization for mobile storage systems. *IEEE Transactions on Computers (TOC)*, 57(10):1386–1398, 2008.

[56] Qin Xin, Ethan L. Miller, and Thomas J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of 13th International Symposium on High-Performance Distributed Computing (HPDC'04)*, pages 172–181, 2004.

[57] Guangyan Zhang, Jiwu Shu, Wei Xue, and Weiming Zheng. SLAS: An efficient approach to scaling round-robin striped volumes. *ACM Transactions on Storage (TOS)*, 3(1):3:1–3:39, 2007.

[58] Guangyan Zhang, Weiming Zheng, and Jiwu Shu. ALV: A new data redistribution approach to RAID-5 scaling. *IEEE Transactions on Computers (TOC)*, 59(3):345–357, March 2010.

[59] Weiming Zheng and Guangyan Zhang. FastScale: Accelerate RAID scaling by minimizing data migration. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, pages 149–161, 2011.

[60] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*, pages 375–386, 2015.

Logical Synchronous Replication in the Tintri VMstore File System

Gideon Glass [*], Arjun Gopalan, Dattatraya Koujalagi, Abhinand Palicherla, and Sumedh Sakdeo [†]

Tintri, Inc.

{gxglass, arjun91, dattatraya, abhinandh, sumedhsakdeo} @gmail.com

Abstract

A standard feature of enterprise data storage systems is synchronous replication: updates received from clients by one storage system are replicated to a remote storage system and are only acknowledged to clients after having been stored persistently on both storage systems. Traditionally these replication schemes require configuration on a coarse granularity, e.g. on a LUN, filesystem volume, or whole-system basis. In contrast to this, we present a new architecture which operates on a fine granularity—individual files and directories. To implement this, we use a combination of novel per-file capabilities and existing techniques to solve the following problems: tracking parallel writes in flight on independent storage systems; replicating arbitrary filesystem operations; efficiently resynchronizing after a disconnect; and verifying the integrity of replicated data between two storage systems.

1 Introduction

Synchronous replication in enterprise data storage systems allows customers to situate redundant storage systems at campus or metropolitan distances. This provides continuous availability in the event of hardware failures, power or cooling failures, or other disasters. One of the storage systems acts as *primary*, responsible for accepting IO from clients. The peer storage system acts as *secondary* and is responsible for accepting replicated IO from the primary system. In the common case when both storage systems are in a synced state, a cluster failover involving a role reversal between the primary and secondary can occur without loss of data.

To handle temporary outages (e.g. minor network glitches, non-disruptive software upgrades of either storage system), the primary and secondary coordinate to bring both the storage systems into a consistent state, without timing out client operations. To handle outages of arbitrary duration (e.g. power failure, lengthy network disruptions, or maintenance related activities) where the secondary is offline or unreachable by the primary, some mechanism for efficient resynchronization is required—once the secondary comes back on line, the primary system should be able to generate and replicate the delta changes that occurred while the secondary was offline.

A related feature that is often supported is *transparent failover*. In the event of a failure of the primary storage system, it enables the secondary system to take over in a way that does not disrupt client applications—with no loss of data and no loss of availability. This failover can be manually driven or automated. Figure 1 illustrates a typical deployment.

We have addressed the problems of synchronous replication and transparent failover by designing, implementing, and deploying a system that is flexible, efficient, and intuitive to use. Our system supports *logical* synchronous replication—the ability to only replicate a portion of the file system namespace, specified as a top-level directory. Designing this system involved solving several sub-problems:

- maximizing the overlap of write execution on the Primary and Secondary storage systems to minimize latency overhead of mirroring; a novel filesystem metadata mechanism is used for this purpose (Section 4).
- replicating complex, arbitrary filesystem operations; a two-phase commit protocol is used for this (Section 5).
- efficiently resynchronizing the two storage systems after extended disconnects (Section 6).

Finally, we have implemented a novel distributed integrity check that allows us to verify periodically or on demand that Primary and Secondary contain identical data (Section 7).

[*] Currently at Google.
[†] Currently at Lyft.

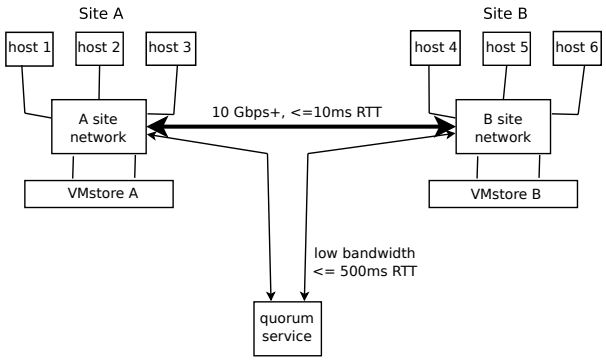


Figure 1: Generic synchronous replication deployment topology with multiple client hosts, a primary storage system, a secondary storage system, and a quorum service to facilitate automatic failover.

2 VMstore Background

In this section we discuss specific characteristics of our workloads and why they motivate logical replication, and our technology base.

2.1 Workload

As its name suggests, VMstore is a specialized storage system designed for virtualization workloads. By default the system exposes a single NFS (or SMB) mount point. On NAS storage, virtual machines (VMs) are typically stored in self-contained directories named after the VM. VM directories typically contain on the order of ten to twenty files, which are either small (e.g. text configuration files) or very large—virtual disk image files corresponding to the virtual disks associated with the VM. VMstore does not support general purpose NFS workloads. As a result, the system can be substantially simplified in one dimension: the number of files it is required to support. VMstore models vary, but support on the order of 100,000 files per system. The median usage is far below these limits, with most systems having under 10,000 files. This simplification affects our replication-related design choices and will be discussed in further sections. A final important aspect of our workload is that file writes comprise in excess of 99.9% of mutation events; file and directory creation, deletion, renames, etc are not uncommon but are generally associated with VM provisioning activities, not with ongoing application workloads running within VM's themselves.

Another aspect of the system is the desire to maximize simplicity for the user. A high priority is placed on making the system usable by IT generalists and virtualization administrators. As a result, the system does not expose traditional storage abstractions such as RAID groups, filesystem volumes, or LUNs to users. Consequently, apart from conceivably replicating the entire storage array, there is no obvious storage abstraction on which to expose synchronous replication for configuration purposes. [1]

From our prior experience with asynchronous replication, we know that many customers choose not to replicate significant portions of their workloads. As an example, a development/test customer may choose to replicate virtual machines housing important data (source code control system, bug database, etc) but not virtual machines running automated continuous integration test workloads. Customers generally are very aware of the relative differences in value of the data inside the different types of virtual machines in their environments. Commonly, a minority of VM's are actually configured for replication. This varies on a continuum, of course, but in rough numbers, it is common for, say, 25% of virtual machines to be configured for replication.

Based on these requirements—that enterprise customers be able to simply express replication policy on a subset of the files and their systems and efficiently replicate that data across geographically disparate locations, we introduce logical synchronous replication as a mechanism to continuously replicate fine-grained subsets of file system state.

2.2 Filesystem Architecture

The VMstore file system is a purpose-built storage system implementing all layers of the storage stack from RAID to file access protocols (NFS and SMB) in a user level filesystem process. Features expected of an enterprise storage system, such as durable writes, automatic crash recovery (from NVRAM), compression, deduplication, snapshots, writable clones, and asynchronous replication are present.

The VMstore file system is a log-structured file system [9]. As such, data is never overwritten in place. Additionally, it implements a transaction system allowing arbitrarily complex metadata updates with ACID semantics. (This system is based on Stasis [10] but with significant local enhancements to integrate with the VMstore log-structured storage system.) We use this facility to construct novel mechanisms to track file writes in progress and to implement per-file content checksums; these are discussed in further sections.

For high availability (HA) within a given system, a VMstore system contains two controllers (x86 servers), each with its own NVRAM and set of network interfaces. The controllers access shared storage devices (SSD and/or HDD, depending on model). High availability is implemented in an active/passive model. For pur-

[1] Per-VM synchronous replication was an option we considered. However, this does not work well with transparent failover, discussed in the next section, because it would require a customer-assigned Cluster IP address on a per-VM basis.

poses of synchronous replication, we do not assume local high availability; replication simply runs on whichever local controller is Active.

3 Replication System Overview

This section introduces our synchronous replication system, describes its configuration model for users, discusses error situations and how we handle them, and outlines how we support transparent failover.

3.1 Introduction

A requirement from our customer base is to support not just synchronous replication but also client-transparent failover across storage systems; customers do not want to perform any reconfiguration in the event of a total failure of one storage system, or a data center outage. To distinguish between local-system HA-related failover and failover across VMstores in a synchronous replication relationship, we refer to the former as *local failover* or *HA failover* and to the latter as *cluster failover*. By *transparent* we mean specifically that clients are not aware of failovers occurring, except for brief periods of disconnection. In particular, no reconfiguration of client hosts is needed—failover, either local failover or cluster failover—requires no manual intervention.

In our environment, virtualized guest operating systems (Linux and Windows) use internal I/O timeouts of 60 seconds or higher. Within VMstore, we require internal failovers to complete within 30 seconds. This gives the hypervisor clients enough time to reconnect and reissue I/Os so that guest operating systems do not time out.

We introduce the notion of a *mirrored datastore*—essentially an IP address and a mount point—as the basis for configuring synchronous replication. We require each mirrored datastore to have a dedicated *Cluster IP address* associated with it. This IP address is mounted by clients for IO operations and fails over between the two storage systems, similar to how local system Data IP addresses are failed over during local HA failover. To make this concrete, Table 1 provides an example showing a mix of synchronously replicated and unreplicated datastores.

Cluster IP	Mount Point	Replicated?
10.200.200.5	/tintri	No
10.300.100.60	/tintri/alpha	Yes
10.300.100.61	/tintri/beta	Yes

Table 1: Example client view of datastores on a VMstore.

The network requirements for Cluster IP addresses are simple: client hosts must be able to reach this address regardless of which VMstore happens to be the Primary at any point in time.[2]

In our system, synchronous replication is applied recursively to all content under the mirrored datastore. In practice this means that a virtual machine which the customer desires to replicate should simply be placed within a mirrored datastore top-level directory (e.g. /tintri/alpha/ImportantMachine), and one which is not desired to be synchronously replicated should be placed in the top-level directory (e.g. /tintri/LessImportantMachine). Note that the contents of *alpha* and *beta* from Table 1 may be replicated to different peer VMstores, or to the same peer VMstore; they are configured independently and are managed independently within VMstore. The cluster IP addresses associated with these datastores are also independently managed and are exported by whichever VMstore system is the replication primary. To simplify error checking and to avoid problems with conflicting policies, we allow only top-level subdirectories to be replicated. The root directory (exported as /tintri) cannot be configured for replication. Customers wanting to replicate their entire workload can simply place all VM's within one or more mirrored datastore subdirectories.

3.2 Failure Model and Operational Requirements

The failure model we assume is as follows. Machines may fail at any time, e.g. because of software crashes or power failures. Datacenters or the network may fail any time and for extended durations. The network may corrupt packets (and go undetected by the TCP checksum); we detect this via strong checksums in our messaging protocol and handle it as we would handle a transient TCP connection loss (i.e. by simply reconnecting). We assume peers are not malicious/Byzantine and VMstores always authenticate each other as the first step in each connection. Finally, individual devices (SSDs or HDDs) may fail, but lower levels of the filesystem insulate replication from having to handle device errors.

We now discuss the operational requirements of our system. Availability is important but consistency (data integrity) takes precedence over availability when there is a tradeoff. Availability, in turn, takes precedence over absolute redundancy. As discussed in detail in Section 3.3, this is done by taking the Secondary out of sync when necessary, and then resychronizing it when it comes back on line. An optional mode, which we have not implemented, would be for the Primary to *fail* I/Os which it could not replicate due to the Secondary (or network) being down. This might conceivably be useful for customers who prioritize absolute redundancy over availability.

[2] For cross-site replication, this requires the use of a stretched layer 2 or layer 3 network. Customers who deploy synchronous replication typically already have this in place.

3.3 Replication States

Figure 2 depicts a slightly simplified view of the state of a given mirrored datastore within a given VMstore system. Each VMstore maintains its state for a given datastore separately, so for a given mirrored datastore there are really two instances of this state machine operating in a loosely coupled manner.

The following invariants relate to the state machine and affect allowable state changes.

1. Only one VMstore may be Primary at any given time. Whichever system is Primary owns the Cluster IP address and advertises it on the network.
2. Upon initial configuration of synchronous replication, the Primary system may have a significant amount of data in the configured subdirectory. An initialization process is required to bring the Secondary into sync. The initialization process is essentially a special case of resynchronization (discussed in Section 6) in which the Secondary happens to be empty at the start of the process. This process may take a significant amount of time. The top row of the state machine contains states related to initialization/resync. The datastore is not in sync in these states.
3. During initialization and resync, the Secondary does not have a complete copy of data. Cluster failover is not possible until initialization/resync completes.
4. The normal state of operation is that both systems are connected and in sync (the Primary is in state 3; the Secondary is in state 7). In this state, client operations are fully replicated and are persisted to NVRAM on both systems prior to being acknowledged to clients (i.e., normal synchronous replication semantics are in effect).
5. Automatic cluster failover (of Secondary to Primary) requires a quorum—two of three systems. An external quorum service having storage independent of the two VMstores is provided for this purpose. Automatic cluster failover is initiated by an in-sync Secondary after a period of time (30 seconds) for which it has not heard from the nominal Primary, provided the Secondary can communicate with the quorum service.
6. Conversely, to handle a Secondary which is inaccessible, a Primary must undergo a transition to mark the datastore as being "out of sync" (not shown in Figure 2). There is a subtlety here: the Secondary, meanwhile, may have initiated a cluster failover in conjunction with the quorum service. As a result, a Primary must coordinate with the quorum service in order to mark a Secondary as being out of sync, and must be prepared for this to fail (i.e. if the Secondary already took over). If that fails, the former Primary must relinquish ownership of the datastore and must drop in-flight I/O requests and not return errors to clients.[3]

A design choice we enforce is that operations which can succeed on the Primary but which fail on the Secondary result in the Primary taking the Secondary out of sync immediately (a transition from state 3 $\rightarrow$ state 4 $\rightarrow$ state 2). The most likely example of this type would be the Secondary being out of space; less commonly, the Secondary might encounter some other limit (e.g., number of files, number of snapshots) that might prevent an operation that otherwise can succeed on the Primary. Again this policy reflects the choice to prioritize availability over absolute redundancy in some conditions. (To recover from this condition, the Primary will periodically reconnect to the Secondary and attempt to resync it; this will succeed if the user has freed up capacity or otherwise addressed the constraint that earlier had caused failure.)

The state machine also reflects a practical engineering consideration: taking the Secondary out of sync then later resyncing it has a non-trivial minimum cost, and we seek to avoid taking a Secondary out of sync if possible. This corresponds to two practical scenarios: brief network outages, and local HA failovers due to software crashes and restarts.

In practice this means that we attempt to recover from brief disconnects (up to approximately 30 seconds) by pausing client acknowledgments at the Primary, and attempting to reconnect to the Secondary in the background. If the reconnect attempt succeeds within the timeout, the Primary will resend buffered, unacknowledged updates (and only then ack the client), and the system will stay in Sync. The states in the second row of Figure 2 reflect these activities. "Catching Up" in the state descriptions refers to replicating (possibly re-replicating) buffered updates from the Primary; Sections 4 and 5 discuss this in detail.

The protocol between the VMstores and the quorum service solves a standard distributed consensus problem and will not be discussed in detail. The quorum service is provided by a standalone software application which can be installed by the customer in a virtual machine either on premises or in the public cloud; the main operational requirements are that the storage for the quorum service must be independent from the VMstores for which it is arbitrating, and network connectivity to the quorum service should be good.

[3] In practice, a soon-to-be-former Primary that becomes isolated on the network must relinquish ownership of the mirrored datastore, and must give up the Cluster IP, *before* the Secondary takes over, to prevent both systems from attempting to advertise the Cluster IP on the network. The Primary transitions out of the Primary/Synced/Connected state using a smaller timeout (e.g. 25 seconds) than the timeout driving the Secondary's attempt to initiate cluster failover. These transitions are not shown on the diagram for brevity.

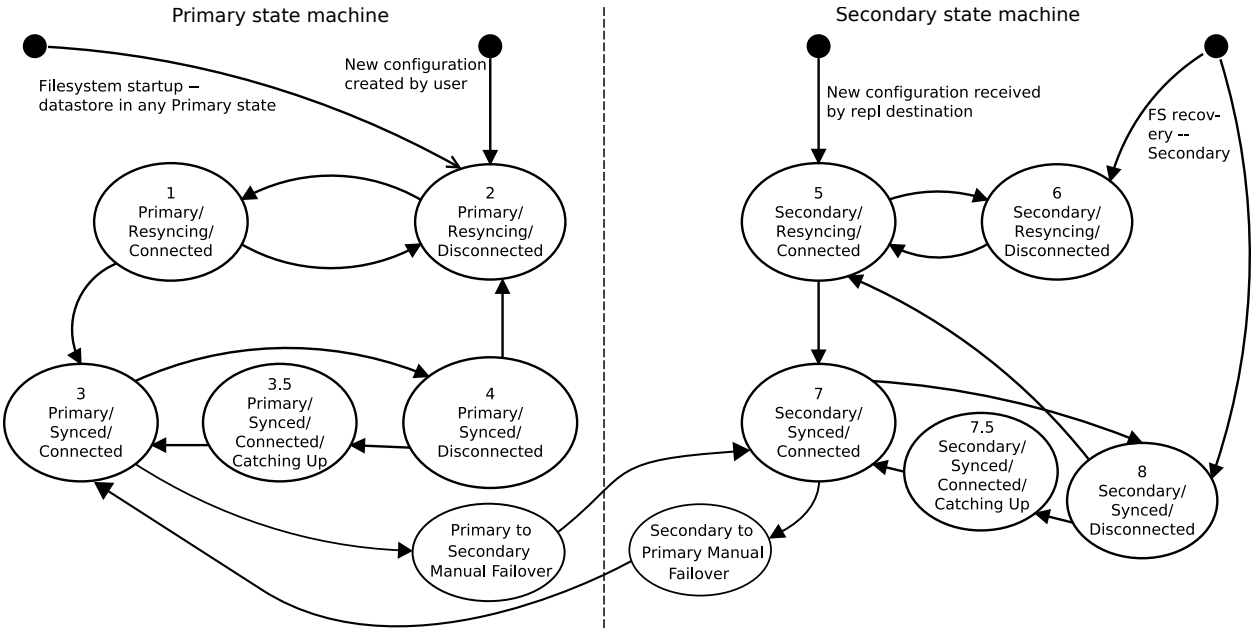


Figure 2: Replication System State Machine. This is shown from the point of view of a mirrored datastore in a single VMstore. The other VMstore will be in one of the other states. For brevity, states relating to automatic cluster failover are not shown.

3.4 Client Transparency

Implementing transparent cluster failover requires that clients of a failed storage system be able to reconnect to the Cluster IP address and see a view of the world exactly consistent with what they previously saw. This includes file content, user-visible metadata (path names, file attributes, etc) and client system-visible metadata (e.g., NFS file handles). To implement this, we assign internal file identifiers and NFS file handles as follows. The Primary makes all such assignments without having to coordinate with the Secondary—the fact that it is Primary gives it the right to assign these values.[4]

- **File Global Identification**. We identify each file in every mirrored datastore with a globally unique FileId. This consists of a 128-bit datastore-specific UUID and a datastore-relative 64-bit monotonically increasing counter. The global FileID value is used extensively within the replication system, as the values are the same on both systems.
- **NFS File Handles** must be stable across cluster failovers. File handles are based on the global FileId. Because only the Primary system assigns FileId's, and because FileId's are unique within a datastore (due to containing the datastore UUID as a prefix), we avoid any need for granular synchronization to negotiate assignment of these values.
- Operation sequencing: **Operation Sequence Numbers** (OSN's) are used to globally order all operations within a mirrored datastore. An OSN consists of a 64-bit cluster generation number, incremented whenever a cluster failover occurs, and a 64-bit local sequence number, assigned and incremented on the cluster Primary for every new incoming operation. Replicated operations are tagged with OSN's for bookkeeping purposes.

During a cluster failover in which a Primary loses ownership of a datastore, the system must be able to identify stale operations and drop them, rather than execute them. A full description of the solution to this problem is beyond the scope of this paper, but to summarize, we tag all requests entering the system with *tokens* which contain among other things the cluster generation number. Requests tagged with generation numbers older than the current generation number must be dropped.

[4] A side effect of this scheme is that at initialization, if the source datastore directory contains existing content, that content must be *re-assigned* new file identifiers—and consequently, file handles—to avoid possible conflicts with unrelated existing content on the Secondary. We require that this content be off-line while this occurs, as clients will encounter stale file handles if accessing content during this process. In practice, customers almost always configure synchronous replication on empty source directories and migrate data into the mirrored datastore by using live storage migration features (e.g., Storage vMotion) in the virtualization system.

4 Data Path: Writes

Minimizing write latency is an important consideration in primary storage systems. Users expect write latencies on unreplicated systems to be at most small numbers of milliseconds. Replication distances are typically small. Industry-wide guidance typically calls for not more than

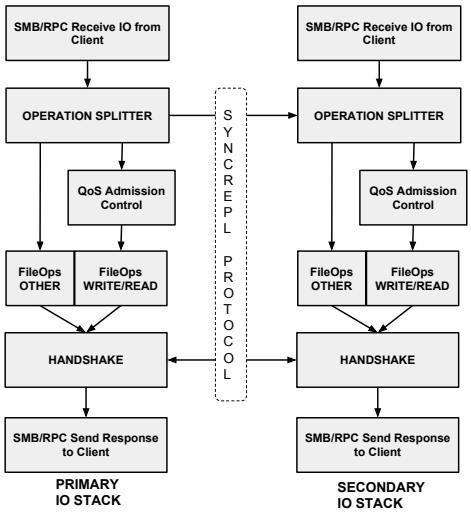


Figure 3: IO stack with operation splitting and handshake for synchronous replication between a pair of storage systems.

10ms RTT between storage systems; customers commonly deploy with 1-2ms RTT. In order to maximize overlap between write execution on the Primary, replication of writes to the Secondary, and write execution on the Secondary, we perform replication processing as early in the pipeline as possible.

Figure 3 depicts logical steps in write processing inside the VMstore file system. The top box indicates protocol processing for Microsoft SMB [1] or RPC/XDR [11, 12] processing for NFS. The module labeled "operation splitter" refers to front-end synchronous replication processing—it splits write operations for local processing and enqueues them for transmission to the Secondary. The operation can then traverse various stages of the pipeline in parallel on both the storage systems. In case one storage system is under contention, the operation would experience queueing delays only on that individual storage system.

The modules labeled "FileOps Write/Read", and "FileOps Other" correspond to existing IO stack processing for reads, writes, and all other operations (file creations, deletions, truncations, directory operations, etc). Once operations have executed locally within the Primary VMstore, flow returns to synchronous replication in the box labeled "Handshake Module". On the Primary, the handshake module will hold onto the write operation until the Secondary sends an acknowledgement for this write or until such time as synchronous replication allows the operation to be acknowledged to the protocol client, whichever is sooner. The latter corresponds to when the response from the Secondary takes longer than the allowed time forcing the Primary to go out of sync (as described in Section 3.3). On the Secondary, the handshake module will hold onto the write operation until the Primary has acknowledged completing the write to the Secondary. This step is required because the Secondary keeps track of writes that are still pending execution on the Primary and the acknowledgement from the Primary is used to free up metadata for the write on the Secondary (discussed below).

4.1 File Locks

The desire to achieve maximum parallelism within the system must be balanced against concerns for correctness. Here are some cases that illustrate these considerations:

1. Suppose a file is created and then is immediately written. What happens on the Secondary if, because of queueing at different points in processing within the system, the write operation is able to be processed on the Secondary before the file create operation completes its processing? The Secondary may have allowed the Primary to acknowledge the file creation to the client before the Secondary has actually finished *processing* the file creation—the operation may have been intent-logged only (this is discussed in Section 5).
2. What happens if a client issues a write to a given file, at a given [Offset, Byte Count], and then issues a separate, overlapping write before the first write has acknowledged? How do we ensure that these writes are processed in the same order on the two systems?

In designing our replication system, an important consideration was to avoid modifying a lot of the existing file system logic as far as possible. In VMstore, metadata operations (e.g. file creates) are executed by threads in a particular thread pool. Write operations are executed separately by a software pipeline involving a different set of threads.

To address these issues, the operation splitter module (Figure 3) imposes two kinds of file locks: *per-file operation locks* and *range locks*. The per-file operation locks are acquired by reads, writes, and metadata operations prior to their execution. Reads and writes acquire operation locks in a shared mode, whereas metadata operations like truncate, file create, rename, etc acquire them in an exclusive mode. This causes metadata operations to be executed in isolation on both the storage systems and solves case 1 above. To address case 2, we introduce range locks maintained on a tuple consisting of $<$ FileID, StartOffset, RequestSize $>$. Write and read operations acquire locks using the respective offset and byte count values specified in the operation. This allows reads and writes to non-overlapping byte ranges to execute in parallel. In practice lock contention is not a problem because clients do not issue update patterns that inherently

race. This is because guest operating systems are generally careful to avoid issuing multiple outstanding writes to overlapping ranges of disk blocks.

4.2 Metadata Support and Distributed Recovery of Writes

As noted earlier, one of the challenges of any synchronous replication system is to keep track of all writes in progress. We leverage a combination of NVRAM and our flexible metadata transactional system for this purpose. For each synchronous replication datastore, we maintain a persistent hash table that contains one entry per in-flight write. Entry tuples are <FileId, OSN, byte offset, length> with OSN being the key. We refer to these tuples as *PAW Entries* ("Primary Acknowledgement Waiting") on the Primary. On the Secondary, the same set of tuples are maintained for writes which have been received and executed and we refer to them as *SAW entries* ("Secondary Acknowledgement Waiting"). To summarize, the presence of a PAW entry means that the Primary is waiting for an acknowledgement from the Secondary and the presence of a SAW entry means that the Secondary is waiting for an acknowledgement from the Primary.

PAW/SAW metadata is associated with each write operation as it proceeds through various write processing stages within VMstore. As part of the metadata transaction which updates file metadata to point to a newly-written block, the transaction also commits a PAW entry identifying the block in question. This is done both on the Primary and the Secondary. Identifying all blocks which may be dirty on both sides requires iterating over all PAW/SAW records in the mirrored datastore. The cost of this is proportional to the number of in-flight IOs in the system, which is bounded. If the system undergoes a local HA failover before the transaction commits and persists, the PAW/SAW information can be retrieved from NVRAM along with the data and other information about the write.

The ***PAW/SAW update sequence*** across the two systems is as follows:

1. When processing a write, the Primary will create a local PAW entry;
2. After receiving the write, the Secondary will have stabilized the write to NVRAM and will subsequently create a SAW entry for itself; the Secondary then acks the write to the Primary;
3. At this point (and after its local NVRAM update is finished), the Handshake Module on the Primary acks the write to the client;
4. Simultaneously, the Primary acks the Secondary's ack and releases its PAW entry;
5. Upon receipt of the second ack, the Secondary can free its SAW entry.

Note that the Secondary commits writes to NVRAM and SSD storage independent of the Primary. As a result, scenarios like the following are possible:

1. Primary receives writes W1 and W2 (could be to the same or different files).
2. These writes are mirrored to Secondary and are enqueued for local processing on Primary.
3. Primary writes W2 persistently. However, before writing W1 to NVRAM, the Primary crashes and performs a local HA failover.
4. Secondary writes W1 persistently. However, before writing W2 to NVRAM, the Secondary crashes and also performs a local HA failover.
5. The Primary and Secondary complete local HA failovers independently.
6. After recovering, the Primary is able to connect to the Secondary and must now reconcile W1 and W2.

To handle this situation and really any situation where a Primary and Secondary have become disconnected (and one or both may have restarted), after every reconnect we perform what we call *distributed recovery*. This happens regardless of whether the systems are in-sync or not. To handle in-progress unacknowledged writes, both sides iterate over all the PAW/SAW entries in the datatstore. The Secondary sends its set to the Primary which merges it with its own. Then, the Primary simply reads out its copy of data for all blocks involved and sends the data to the Secondary to rewrite the relevant blocks. This ensures that the datastore contents are identical upon the completion of distributed recovery (for the in-sync case). Once this completes, new writes are allowed into the system on the Primary.

In the scenario described above, W2 is persisted on Secondary for the first time as part of distributed recovery. With respect to W1, the data on Secondary undergoes a rollback to the contents as dictated by the Primary. This is correct because neither W1 nor W2 was acknowledged to the client prior to the sequence of crashes.[5]

At some point it is necessary to remove PAW/SAW entries to bound the work involved in distributed recovery. If the write has persisted on the Secondary, and the Primary has an acknowledgement of that, the corresponding PAW entry is deleted on the Primary. Similarly, if the write has persisted on the Primary, and the Secondary has an acknowledgement for that, the corresponding SAW entry is deleted on the Secondary. PAW and SAW entry deletions correspond to Steps 4 and 5 respectively in the PAW/SAW update sequence described above.

[5] Presumably the client will reconnect per its normal logic and reissue both writes to the Primary. However, if the client also crashes—which is fine—then both the Primary and Secondary end up with only write W2 written persistently. This is also fine: there is no guarantee about whether the storage systems stabilized an unacknowledged write, and there is no ordering guarantee between simultaneously executing unacknowledged writes.

5 General Filesystem Operations

We use the term *metadata operations* (or just *metadata ops*) to refer to all filesystem modifications other than file writes. Many of these operations are familiar POSIX/NFSv3 operations: file creation and deletion, directory creation and deletion, rename, setattr, link creation, and so on. Additionally, we implement several proprietary operations. A full description of these operations is beyond the scope of this paper, but to summarize, the operations are space reservation (in which the system attempts to reserve physical capacity for the full logical size of a given file); file-level snapshot creation and deletion; and file-level clone creation.[6]

Metadata ops differ from writes in several important ways. While they are not infrequent (they may occur tens to hundreds of times per second, in active provisioning workloads), they are much less frequent than writes, giving us more implementation flexibility. Second, whereas writes can be undone simply by reading out data from the Primary and overwriting whatever data may exist on the Secondary (Section 4.2), there is no equivalent mechanism available to undo metadata operations. Thus, a more general mechanism is required to track in-flight metadata ops.

5.1 Operation Logging and States

We implement a scheme similar to two-phase commit to ensure that both systems track metadata ops and agree that they can be executed prior to executing them. Prior to being executed, metadata operations along with their respective OSN's, operation-specific arguments, etc., are *intent logged* on both sides. Physically, the log is simply a space-reserved file in a hidden, unreplicated portion of the file system. One intent log is maintained for each mirrored datastore. Log updates are efficient: we utilize the existing file write path which provides low-latency stable writes via NVRAM. The log size is not large because the log can be logically truncated regularly with no impact on performance. Note that the log is not used when the datastore is out of sync, so there are no limitations arising from log storage capacity.

Figure 4 depicts the general flow for metadata operations. To summarize, both sides log each operation along with an operation state:

- PENDING
- COMMITTED

[6] VMstore implements VM level snapshots as point-in-time snapshots over the set of files comprising the physical embodiment of the virtual machine: various metadata files and the virtual disk files, and possibly files capturing dynamic state, e.g. memory and swap. Within VMstore, a VM level snapshot consists of a set of file-level snapshots, taken atomically, and a certain amount of snapshot-wide metadata. VM level clones are implemented by instantiating a set of file-level clones, writable files which reference an underlying base snapshot in a read-only manner.

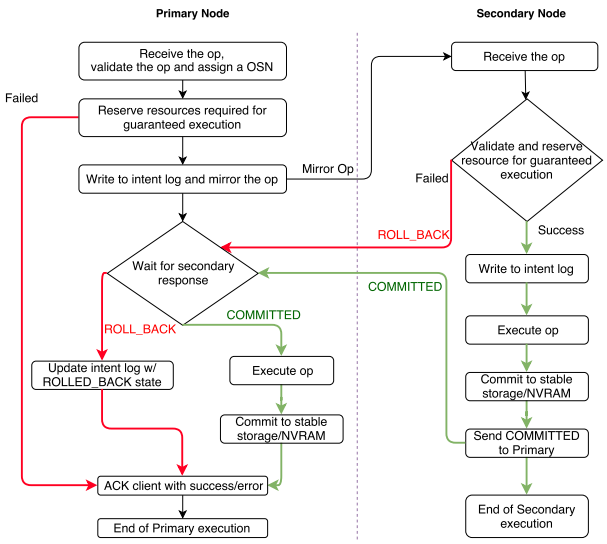


Figure 4: Metadata operation execution scheme that implements a two-phase commit protocol.

- ROLLED_BACK

Either side may decide that an operation may not succeed, for a variety of operation-specific reasons. For example, space reservation may fail on the Secondary but not on the Primary. In general, the protocol gives the Secondary the opportunity to determine if an operation should fail.

The first step in processing is for the Primary to validate the request (arguments are valid, resources are available, etc). If this fails, an immediate failure is returned to the client. If this succeeds, the operation is logged in the PENDING state and the operation is replicated to the Secondary. This Secondary then decides whether the operation can succeed. For operations which the Secondary cannot execute, it simply sends a ROLLED_BACK reply to the Primary. For successful operations, the Secondary first logs them in the COMMITTED state, then executes them, then sends a COMMIT reply back to the Primary. At this point the Primary executes the operation.

As part of executing metadata operations, the existing code paths all utilize the VMstore transaction mechanism. We augment these code paths to tag each file with the OSN of each completed operation for reasons discussed in the next section. Finally, when both sides finish executing the operation, file locks are released, and the Primary is allowed to acknowledge the operation to the client.

5.2 Distributed Recovery For Metadata Operations

Crash recovery for metadata operations must handle the same sort of considerations that were discussed above for write operations: messages may be lost; local HA

failovers may occur at any time; etc. To recover in-flight metadata operations, we adopt an approach conceptually similar to what we used for in-flight write operations. We consolidate intent log entries on both sides by finding all entries in the COMMITTED state in both systems' intent logs. Note that, as it is implemented currently, the Secondary always commits a given entry first: it does this before sending a reply to the Primary that, upon receipt, allows the Primary to commit the operation. Thus, distributed recovery for metadata ops involves scanning the live portion of the intent log on the Primary for committed operations and simply sending them all to the Secondary. By definition, every log entry in the intent log that is in the COMMITTED state needs to be reapplied if it has not already been applied. Both the Primary and Secondary also take care of this during system start up.

In general, metadata operations are not idempotent. Some are, but we handle the general case and ensure that all metadata operations are executed exactly once. Log replay handles this simply and efficiently by comparing each operation's OSN with the last-executed OSN on the respective files. Operations which have already been done are simply ignored. The same OSN based comparison is also used in the replay of write operations where these writes are just discarded if the corresponding files have subsequently been deleted.

Because of the need to correctly interleave metadata operations with file writes, the relationship between metadata distributed recovery and file write distributed recovery is simple: metadata distributed recovery is done first, then file writes are recovered. This ensures that files are created prior to writes being recovered.

6 Data Path: Resync

Efficient resynchronization is important in any synchronous replication scheme, because the alternative is basically untenable: rereplicate the entire copy of data from the Primary, possibly tens to hundreds of terabytes. This section describes how we perform resynchronization using file-level snapshots.

As discussed in Section 3, our threshold for extended disconnects is 30 seconds, after which one of two things can happen—the Secondary takes over and becomes Primary (in conjunction with a Quorum Server), or the Primary marks the Secondary as being "out of sync". In the latter case, the Primary stops replicating operations to the Secondary but continues to execute them locally. This will be the state of the datastore on the Secondary until the Secondary becomes reachable again, at which point we begin the process of resync. To allow for efficient resync some method is needed to track incremental changes that occurred after the systems went out of sync.

In VMstore we leverage efficient per-file snapshots that are implemented internally as a linked list of persistent delta B-trees by the filesystem metadata layer. At the time of going out of sync, the Primary will create a special *resync snapshot* on all file(s) within the affected mirrored datastore. (This is possible because our snapshots are relatively cheap, and the number of files, as mentioned earlier, is bounded and small.) The state of the Secondary can be determined by the Primary in the future when it is time to perform resync, from the combination of the data captured in the resync snapshots and the metadata about writes in-flight tracked via the PAW/SAW entries. When resync begins, the Primary is able to efficiently identify data written after going out of sync by observing the delta between the time at which the resync snapshots were created and the current filesystem state. No work is required to materialize these deltas; they are maintained directly by the underlying filesystem metadata layer and can simply be read out on a per-file basis.

When the systems are out of sync, arbitrary filesystem manipulations may occur on the Primary—files and directories may be deleted, renamed, created from scratch, etc. One of the goals for resync is to avoid replicating updates to files which have subsequently been deleted on the Primary. Of course, the basic requirement is that resync must bring the Secondary into a state of being identical with the Primary. With this in mind, we perform resync processing in three steps:

1. Bring the Secondary into a state of being identical *with the content in the resync snapshots*. This applies to files which existed at the time the Secondary went out of sync, and is skipped for files created after the systems went out of sync. Arbitrary metadata operations that were in-flight when going out of sync are also reapplied on the Secondary.
2. Bring the Secondary directory namespace into sync with the Primary. This handles all deletions, renames, and file creations that occurred while out of sync. This also enables us to replicate new namespace manipulation operations while resyncing.
3. Resync file content on a file by file basis. Within each file, resync on an offset range by offset range basis.

Step (1) is similar to the distributed recovery procedure discussed above that we run immediately after connecting in-sync systems that have been briefly disconnected; the difference is that with resync, the file content must be read from the file-level resync snapshots on the Primary, not from the current live version of the file. As with the distributed recovery scheme, the PAW/SAW persistent metadata identifies blocks which were subject to in-flight writes at the time the systems went out of sync. Note that after going out of sync, the PAW/SAW metadata is essentially frozen to preserve the knowledge of which blocks had ongoing writes until the time we can use this information in resync. For files which exist on

both Secondary and Primary (this is the normal case for long lived workloads), after this step, the Secondary is now identical in content to the Primary at the time the resync snapshot was taken on the Primary.

Step (2) allows us to optimize out writes that occurred to files which were subsequently deleted on the Primary, and to generally reclaim these files on the Secondary as early as possible. This reduces pressure for filesystem capacity on the Secondary and avoids scenarios where the Secondary may run out of space simply because it hasn't yet deleted files that we know have been deleted from the Primary.

Finally in Step (3) we iterate over all files on the Primary. The delta between the current file's content and the resync snapshot can be extracted efficiently on the Primary and the data read out and sent. Internally in VMstore, files are identified by a local FileId value, a 64-bit monotonically increasing sequence number. Files are resynced in increasing order of local FileId. This makes it fairly simple to persistently track resync progress; within a given mirrored datastore, we store a single local FileId value persistently during resync. Similarly, the offset within the resync snapshot is checkpointed as well. This allows resync to resume without performing a large amount of re-replication of data in the event of a local crash and restart on the Primary while it is performing resync. Checkpointing resync progress at a granular level is important because large virtual disk files (e.g., in excess of 10TiB) are not uncommon.

New writes to files which have been created after the systems begin resync and writes to offset ranges in files that have already been resynced are handled by mirroring them synchronously. This ensures that resync converges toward completion, i.e. it does not run the risk of falling behind incoming live writes and never completing.

6.1 Handling user-created snapshots

VMstore implements VM-level snapshots (scheduled or manual) using per-file snapshots. These per-file snapshots are atomically created across the set of files comprising a given VM. This complicates resync. At the start of resync, there may exist file level snapshots on the Primary which were created while the systems were out of sync. Conversely, there may exist file-level snapshots on the Secondary which were deleted from the Primary while the systems were out of sync. Similar to how file deletions are replicated during resync prior to sending incremental data, we replicate discrete snapshot deletion operations first, prior to replicating snapshot contents.

With the exception of clone create, snapshot create, and snapshot delete operations, most of the metadata operations that the Primary receives while resyncing are replicated to the Secondary immediately because the namespaces are in-sync. New snapshot creates are only replicated if the files involved are in-sync on the Secondary. Snapshot deletes are only replicated if the snapshot has been resynced to the Secondary. Clone creates are only replicated if the required backing snapshot is present on the Secondary. The resync process must eventually take care of replicating any of these operations if they are delayed from being replicated at the time they were issued to the Primary. Note that these operations themselves are not logged; the resulting file system state (the set of snapshots and clones) is discovered by the local FileId-based iteration described above.

7 Distributed Integrity Verification

The VMstore file system implements an incrementally-updated per-file content checksum for purposes of data integrity verification. The checksum is neither a cryptographic checksum nor a guarantee that corruption has not occurred; rather it is a probabilistic mechanism designed as an extra check on top of many other mechanisms (transactions, crash recovery, NVRAM, careful design, code review, thorough testing, etc) used collectively to ensure data integrity.

The checksum physically comprises approximately 1KB of metadata; file writes update portions of this checksum based on the file offset being written and the block content itself. Each block write updates the checksum using 7 bits derived from data in the write. The checksum metadata updates are performed efficiently using the transactional metadata mechanism noted in Section 2 (essentially the related metadata updates—system-wide statistics, file statistics, B-Tree updates to point to new blocks, etc—are logged together). Additionally, at the time of snapshot creation, a file's current checksum is stored with the associated file-level snapshot metadata.

The checksum values are used in several places. During file deletion, each block's checksum contribution is logically subtracted from the remaining file checksum value, and at the end of deletion, the checksum must be logically zero. Similar logic is used when truncating a file to zero bytes in size.

In synchronous replication, the basic requirement to maintain identical copies of files on both systems (as long as the systems are in sync) enables us to leverage the file content checksums for integrity verification. Integrity verification involves the following steps. First, writes and other operations are temporarily paused using the exclusive file-level lock mechanism described in Section 4.1. Next, in-flight operations are flushed through the system. Following this, the Primary reads out its per-file checksum values and sends them to the Secondary, which reads out its values and compares them. These checksums are expected to match across the two systems.

In order to avoid blocking file operations for an extended period of time, which could be the case if the data-

store contains several thousands of files, distributed integrity checking is done in a batched manner. This allows us to acquire file-level locks one batch at a time as opposed to for the entire datastore. The batch size is chosen such that integrity checking is transparent to clients—it lasts at most a few seconds for any given file—and such that it minimizes network communication.

Content checksum mismatches are expected never to occur in practice. However, if they are encountered, the system takes the Secondary out of sync and lets the Primary continue servicing client IOs, to avoid interruption of service. Additionally the system logs the affected files and their checksum values on both sides. The differences in the checksums allow us to identify a set of candidate file blocks that may be different, and if the number of blocks in this set is below a threshold, the systems additionally read out and save off the affected blocks for later inspection. This mechanism has been occasionally useful in debugging the system during development.

In production we run the verification procedure such that each file is checked once every 24 hours, provided that the datastore is in-sync. Additionally, we also proactively perform checksum verifications at certain points, e.g. just prior to user-initiated cluster failover and at the end of resync.

8 Evaluation

We have implemented a heavily multithreaded write pipeline, each stage of which does asynchronous processing. This improves performance and also isolates processing of mirrored and non-mirrored datastore requests. We evaluated our implementation to answer the following questions:

- What is the overhead of synchronous replication on read and write throughput?
- What is the impact of the VMstore network RTT on client latency for various write workloads?

CPU	Xeon E5-2630 v2 (2x6 cores, 2.60GHz)
RAM	64GB DDR3 at 1600 Mhz
Flash	11x480GB SATA SSDs
Disks	13x4TB SAS SED HDDs
NIC	Intel X540-T2 at 10Gbps

Table 2: VMstore hardware configuration used in experiments.

Experiment setup: We used two VMstores running Tintri OS 4.3 (see Table 2 for hardware configuration[7]); one as the Primary and the other as the Secondary connected to each other through a 10Gbps ethernet link. We also had a Linux-based physical client machine which was connected to the Primary through a 10Gbps ethernet link. A tool that drives synthetic IO traffic over NFS was used to generate random read and write IO traffic with 8KiB, 64KiB and 256KiB block sizes. These IO sizes were chosen because they represent the majority of workload sizes observed in VM workloads. The network RTT between both the VMstores and between the client and the primary VMstore as measured by *ping* using a packet size of 64 bytes was observed to be 100μs on average. In some experiments, we used the *tc* (traffic control) Linux utility to vary the RTT between the two VMstores.

[7] As it happens, we used model T850, introduced in 2014, for these experiments. Two generations of newer hardware families have succeeded this model, so performance on current systems will be higher.

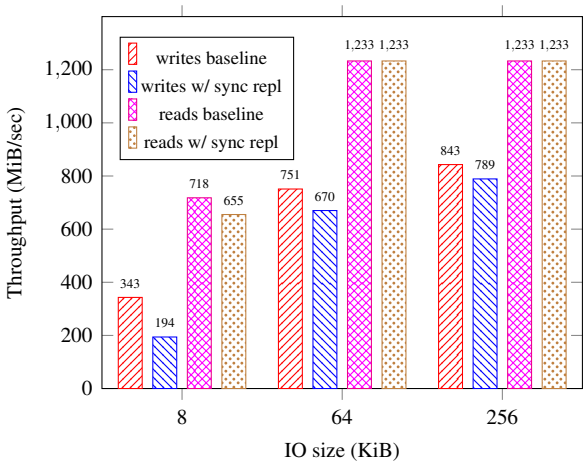


Figure 5: Throughput comparison between baseline performance and performance with synchronous replication for different IO sizes. The RTT was 100 μs; no additional delay was induced in the network.

8.1 Throughput

Figure 5 graphs read and write throughput for various IO sizes—8KiB, 64KiB and 256KiB under two scenarios: i) baseline performance when synchronous replication is not enabled, and ii) performance when synchronous replication is enabled.

Synchronous replication imposes a 43% overhead in throughput for 8KiB writes, 11% for 64KiB writes and 6-7% for 256KiB writes. The difference is significant for 8KiB writes because of the per-request processing overhead of replication in our system. This is due to file range locks, sending the request through various queues, memory allocation, and other assorted software overhead.

Synchronous replication imposes a very minor overhead on read performance; about 8% for 8KiB reads. This is because reads to files in synchronously replicated datastores also have to acquire shared file locks. Larger 64KiB and 256KiB IO size reads end up saturating the 10 Gbps network link even when synchronous replica-

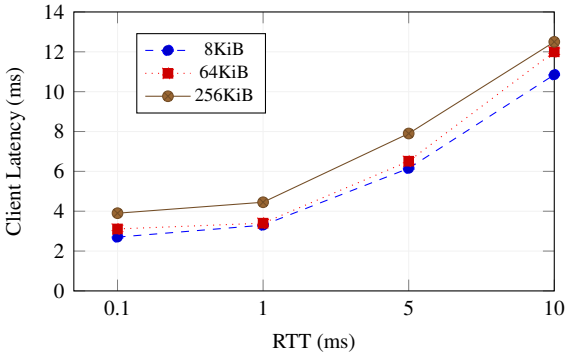


Figure 6: Impact of VMstore network RTT on client latency for various write workloads. For each workload, the client-side load was kept constant for all RTT values.

Workload	Throughput(MiB/sec)			
	0.1 ms	1 ms	5 ms	10 ms
8KiB	205	167	80	50
64KiB	725	660	340	250
256KiB	754	671	380	240

Table 3: Write throughput for different RTT values. The client-side load was fixed for each workload as the RTT was varied.

tion is enabled because bulk data movement dominates the fixed processing costs in the file system.

Performance in our system is subject to continuous improvement; with techniques like batching of small writes and write acknowledgements over the network, tuning of TCP connection performance and optimizing read-only workloads, we are confident that we can improve the performance of small reads and writes in subsequent releases of our software.

8.2 Latency

Figure 6 graphs the average client visible latency for various write workloads and for different values of RTT between the two VMstores. For all IO sizes, the trend observed is expected. The client visible latency increases as the RTT increases because every write has to be synchronously replicated to the Secondary. Additionally, as discussed in Section 4, the execution of writes on the Primary and Secondary is allowed to overlap. So, for lower values of RTT, the individual VMstore IO processing times will dominate the client latency and there is a value of RTT beyond which the RTT will start to dominate the client latency. This RTT crossover point depends on the cost of IO processing in the file system as well as the cost of mirroring and is hence workload dependent.

We also observe that the client latencies for 8KiB writes and 64KiB writes are close to each other at lower RTT values even though the latter has a higher mirroring cost. This is because various parts of the Tintri VMstore file system are optimized for 64KiB IO requests.

The difference in the client latency and the RTT gives the average overhead from synchronous replication and IO processing. From Figure 6, we observe that this overhead remains constant at around 3-4 ms for all workloads. This is expected because for a fixed client-side queue depth, any increase in the RTT should only affect the end-to-end client latency and not the latency overhead from synchronous replication. Of course, another consequence of this is reduced throughput. Table 3 captures the actual throughput observed at different RTT values.

9 Implementation Experience and Lessons Learned

Currently the system is in production use at dozens of sites globally. This section discusses our experiences in designing, building, testing, and deploying the system.

Usability: Space limitations prevent us from presenting our user interfaces for configuring the system, operational monitoring, and latency visualization. However, it is fair to say that the feedback from customers about the usability of the system has been extremely positive. The one area where there is a usability challenge relates to functionality which we deferred implementing, discussed next.

Functionality: From the beginning we designed for automated cluster failover. However, there was acute pressure to deliver *some* functionality to customers as quickly as possible. As a result, we elected to deliver functionality in a phased manner, and did not make automatic cluster failover available initially. In retrospect, demand for automatic cluster failover was higher than anticipated, and lack of this support has delayed adoption of the system to some extent.

Performance: Apart from the up-front design work to integrate replication carefully with the existing write pipeline (Section 4) to allow maximal parallelization, we did a modest amount of performance tuning specifically on the replication data paths. There is more that could be done to reduce the throughput gap between unreplicated writes and replicated writes, especially at 8KiB. However, as anticipated, the ability to let customers easily *not replicate* large portions of their workloads, combined with the performance-related work that we did do, has had the net result that there have been minimal performance problems in practice.

Complexity: Prior to undertaking synchronous replication, we had implemented asynchronous snapshot-based replication in VMstore. Synchronous replication is significantly more complex for a number of reasons. First, it necessarily involves fairly significant surgical modifications to various write paths and high level

filesystem operation implementations. By comparison, asynchronous replication on the source system only has to consume snapshots after their creation, and the snapshot abstraction generally insulates asynchronous replication from the dynamic churn of ongoing filesystem operations. Second, synchronous replication must attempt to ensure both low latency and high throughput; asynchronous replication only needs to deliver sufficient throughput. Third, asynchronous replication has no equivalent of client transparent failover or automated cluster failover; failovers involve external reconfiguration of the customer's virtualization environment, implemented by higher level disaster recovery orchestration software, not by the filesystem replication system.

As a result, the synchronous replication implementation required roughly three times as many lines of code compared to asynchronous replication (approximately 100,000 and 35,000, respectively). Additionally, the asynchronous replication code is more self-contained and hence simpler to reason about. To a first approximation synchronous replication took perhaps five times the number of person-months of engineering effort, spread over roughly twice as much calendar time.

Correctness: The distributed data integrity verification mechanism (Section 7) proved invaluable during development and testing. The per-file content checksums on which this mechanism depends had previously been implemented a number of years before we began the synchronous replication project, and had been used extensively in internal testing. We had not enabled file content checksums in production due to lingering performance issues in certain scenarios. As part of the synchronous replication project, we decided early on to do the work necessary to allow us to enable the file content checksums in production. This allowed us to build the distributed integrity checking mechanism on top of the file content checksum mechanism. This took several months of effort on the part of several engineers, but was surely worth it. This mechanism caught a handful of subtle bugs during development and internal testing. However, in a year of shipping the system to dozens of customers, we have not experienced a single data path related customer found defect; the distributed data integrity check has not failed in production.

10 Related Work

In general, synchronous replication schemes in commercial enterprise storage systems are not well described in the literature. Seneca [7] describes a detailed taxonomy of design choices for remote mirroring, and a design for a remote mirroring protocol with correctness validation using I/O automata-based simulation. It also presents some details of existing systems as of 2003, many of which are still in use. Snapmirror [8] discusses an asynchronous replication scheme of using self-consistent snapshots of the data mirrored from a source to a destination volume. The focus is on system performance at the cost of data loss. The tolerance to data loss is proportional to the frequency of taking and mirroring snapshots.

For resynchronization, some enterprise data storage systems (e.g., Symmetrix [4] and Linbit [2]) use bitmaps to keep track of writes that have been processed on the primary but not yet replicated to the secondary. Other systems (e.g., MetroCluster [6]) use filesystem volume level snapshots or system-wide snapshots to achieve this. In contrast, our system uses granular per-file metadata and file-level snapshots.

Some enterprise storage systems also implement synchronous replication to guarantee zero divergence in data between a pair of storage systems. EMC RecoverPoint [4] supports synchronous replication over IP or over FibreChannel network. Their host-based I/O splitting technology is used to mirror application writes with minimal perceivable impact on host performance. HP 3PAR Peer Persistence [5] maintains a synchronized copy of data between a pair of storage nodes, with the host maintaining an active path to one array and a standby path to the other array. A transparent failover and failback between this pair of storage nodes is made possible using a Quorum Witness. These systems operate on the basis of LUNs and thus require significantly more operational expertise compared to our system.

Veritas Volume Replicator [3] is a host-based software system. It makes use of Storage Replicator Log which is essentially a circular buffer to persistently remember writes to be queued for replication to the secondary. Writes have to be first written to this storage replicator log, then replicated to the secondary for persistence. This serialization of IOs is suboptimal compared to our scheme where writes occur in parallel on the primary and secondary.

11 Conclusion

We have implemented logical synchronous replication, a new approach to solving an old problem. We have introduced novel mechanisms to track writes in-flight and reconcile them across systems after reconnects. Additionally, we leverage two-phase commit to replicate complex filesystem operations, and granular per-file snapshots to implement efficient resynchronization. A datastore-wide distributed data integrity verification procedure built on a novel per-file checksum scheme ensures that the system is operating correctly. The flexibility of replicating only a selected portion of a filesystem has proven intuitive and easy to use by users.

12 Acknowledgments

We would like to thank Fred Douglis, Mark Gritter, Tyler Harter, Ed Lee, and Ashok Sudarsanam for their helpful comments on early drafts of this paper. The anonymous reviewers and our shepherd, Andy Warfield, provided insightful feedback that greatly enhanced the presentation of our material. We would like to thank our management at Tintri (Ashok Sudarsanam, Tom Theaker, Tony Chang, and Kieran Harty) for their support in publishing this paper. Finally, we would like to thank the many engineers at Tintri, past and present, who contributed to building this system.

References

[1] [MS-SMB2]: Server Message Block (SMB) Protocol Versions 2 and 3. https://msdn.microsoft.com/en-us/library/cc246482.aspx.

[2] The quick-sync bitmap. https://docs.linbit.com/doc/users-guide-83/s-quick-sync-bitmap/.

[3] Veritas volume replicator option by symantec. http://eval.symantec.com/mktginfo/products/White_Papers/Storage_Server_Management/sf_vvr_wp.pdf, 2006. White paper guide to understanding volume replicator.

[4] EMC VNX Replication Technologies an overview. https://www.emc.com/collateral/white-papers/h12079-vnx-replication-technologies-overview-wp.pdf, 2015. White paper highlighting EMC VNX replication technology.

[5] Implementing vsphere metro storage cluster using hpe 3par peer persistence. https://www.hpe.com/h20195/V2/GetPDF.aspx/4AA4-7734ENW.pdf, 2016. White paper highlighting HPE 3PAR Peer Persistence.

[6] MetroCluster management and disaster recovery guide. https://library.netapp.com/ecm/ecm_download_file/ECMLP2495113, 2017. White paper highlighting MetroCluster.

[7] Minwen Ji, Alistair C. Veitch, and John Wilkes. Seneca: remote mirroring done write. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 253–268. USENIX, 2003.

[8] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. Snapmirror®: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST'02, pages 9–9, Berkeley, CA, USA, 2002. USENIX Association.

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

[10] Russell Sears and Eric Brewer. Stasis: Flexible transactional storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 29–44, Berkeley, CA, USA, 2006. USENIX Association.

[11] Raj Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, August 1995.

[12] Raj Srinivasan. XDR: External Data Representation Standard. RFC 1832, August 1995.

ALACC: Accelerating Restore Performance of Data Deduplication Systems Using Adaptive Look-Ahead Window Assisted Chunk Caching

Zhichao Cao, Hao Wen, Fenggang Wu, and David H.C. Du
Department of Computer Science, University of Minnesota, Twin Cities
{*caoxx380, wenxx159, wuxx0835, du*}*@umn.edu*

Abstract

Data deduplication has been widely applied in storage systems to improve the efficiency of space utilization. In data deduplication systems, the data restore performance is seriously hindered by read amplification since the accessed data chunks are scattered over many containers. A container consisting of hundreds or thousands data chunks is the data unit to be read from or write to the storage. Several schemes such as forward assembly, container-based caching, and chunk-based caching are used to reduce the number of container-reads during the restore process. However, how to effectively use these schemes to get the best restore performance is still unclear.

In this paper, we first study the trade-offs of using these schemes in terms of read amplification and computing time. We then propose a combined data chunk caching and forward assembly scheme called ALACC (Adaptive Look-Ahead Chunk Caching) for improving restore performance which can adapt to different deduplication workloads with a fixed total amount of memory. This is accomplished by extending and shrinking the look-ahead window adaptively to cover an appropriate data recipe range and dynamically deciding the memory to be allocated to forward assembly area and chunk-based caching. Our evaluations show the restore throughput of ALACC is higher than that of the optimum case of various configurations using the fixed amount of memory allocated to forward assembly and to chunk-based caching.

1 Introduction

Coming into the second decade of the twenty-first century, social media, cloud computing, big data, and other emerging applications are generating an extremely huge amount of data daily. Data deduplication is thus widely used in both primary and secondary storage systems to eliminate the duplicated data at chunk-level or file-level. In chunk-level data deduplication systems, the original data stream is segmented into data chunks and the duplicated data chunks are eliminated (not to store). The original data stream is then replaced by an ordered list of references, called recipe, to the unique data chunks. A unique chunk is a new data chunk which has not appeared before. At the same time, only these unique data chunks are stored in the persistent storage. To maximize the I/O efficiency, instead of storing each single data chunk separately, these unique chunks are packed into containers based on the order of their appearances in the original data stream. A container which may consist of hundreds or thousands of data chunks is the basic unit of data read from or written to storage with a typical size of 4MB or larger.

Restoring the original data is the reverse process of deduplication. Data chunks are accessed based on their indexing order in the recipe. The recipe includes the metadata information of each data chunk (e.g., chunk ID, chunk size, container address and offset). The corresponding data chunks are assembled in a memory buffer. Once the buffer is full, it will be sent back to the requesting client such that one buffer size data is restored. Requesting a unique or duplicate data chunk may trigger a container read if the data chunk is not currently available in memory, which causes a storage I/O and impacts restore performance. Our focus is specifically on restore performance in secondary storage systems.

In order to reduce the number of container-reads, we may read ahead the recipe and allocate the data chunks to the buffers in the Forward Assembly Area (FAA). We can also cache the read-out container (container-based caching) or a subset of data chunks (chunk-based caching) for future use. If a requested data chunk is not currently available in memory, it will trigger a container-read. It is also possible that only a few data chunks in the read-out container can be used in the current FAA. Therefore, to restore one container size of the original data stream, several more containers may have to be read from the storage causing read amplification. Read amplification causes low throughput and long completion time for the restore process. Therefore, the major goal of improving the restore performance is to reduce the number of container-reads [1]. For a given data stream, if

its deduplication ratio is higher, its read amplification is potentially more severe.

There are several studies that address the restore performance issues [1, 2, 3, 4, 5, 6, 7]. The container-based caching scheme is used in [3, 4, 5, 7]. To use the memory space more efficient, a chunk-based LRU caching is applied in [1, 6]. In the restore, the sequence of future accesses is precisely recorded in the recipe. Using a look-ahead window or other methods can identify the future information to achieve a more effective caching policy. Lillibridge *et al.* [1] propose a forward assembly scheme. The proposed scheme reserves and uses multiple container size buffer in FAA to restore the data chunks with a look-ahead window which is the same size as FAA. Park *et al.* [7] use a fixed size look-ahead window to identify the cold containers and evict them first in a container-based caching scheme.

The following issues are not fully addressed in the existing work. First, the performance and efficiency of container-based caching, chunk-based caching and forward assembly vary as the workload locality changes. When the total size of available memory for restore is fixed, how to use these schemes in an efficient way and make them adapt to the workload changing are very challenging. Second, how big is the look-ahead window and how to use the future information in the look-ahead window to improve the cache hit ratio are less explored. Third, acquiring and processing the future access information in the look-ahead window requires computing overhead. How to make better trade-offs to achieve good restore performance, but limit the computing overhead is also an important issue.

To address these issues, in this paper we design a hybrid scheme which combines chunk-based caching and forward assembly. We also propose new ways of exploiting the future access information obtained in the look-ahead window to make better decisions on which data chunks are to be cached or evicted. The sizes of the look-ahead window, chunk cache, and FAA are dynamically adjusted to reflect the future access information in the recipe.

In this paper, we first propose a look-ahead window assisted chunk-based caching scheme. A large Look-Ahead Window (LAW) provides the future data chunk access information to both FAA and chunk cache. Note that the first portion of the LAW (as the same size as that of FAA) is used to place chunks in FAA and the second part of the LAW is used to identify the caching candidates, evicting victims and accessing sequence of data chunks. We will only cache the data chunks that appear in the current LAW. Therefore, a cached data chunk is classified as either an *F-chunk* or a *P-chunk*. *F-chunks* are the data chunks that will be used in the near future (appear in the second part of LAW). *P-chunks* are the data chunks that only appear in the first part of the LAW. If most of the cache space is occupied by the *F-chunks*, we may want to increase the cache space. If most of the cache space is occupied by *P-chunks*, caching is not very effective at this moment. We may consider to reduce the cache space or to enlarge the LAW.

Based on the variation of the numbers of *F-chunks*, *P-chunks*, and other measurements, we then propose a self-adaptive algorithm (ALACC) to dynamically adjust the sizes of memory space allocated for FAA and chunk cache, and the size of LAW. If the number of *F-chunks* is low, ALACC extends the LAW size to identify more *F-chunks* to be cached. If the number of *P-chunks* is extremely high, ALACC either reduces the cache size or enlarges LAW size to adapt to the current access pattern. When the monitored measurements indicate that FAA performs better, ALACC increases the FAA size, thus reduces the chunk caching space, and gradually shrinks LAW. Since we consider a fixed amount of available memory, a reduction of chunk cache space will increase the same size of FAA or vice versa. For the reason that LAW only involves meta-data information which takes up a smaller data space, we ignore the space required by LAW, but focus more on the computing overhead caused by the operations of LAW in this paper.

Our contributions can be summarized as follows:

- We comprehensively investigate the performance trade-offs of container-based caching, chunk-based caching and forward assembly in different workloads and memory configurations.
- We propose ALACC to dynamically adjust the sizes of FAA and chunk cache to adapt to the changing of chunk locality to get the best restore performance.
- By exploring the cache efficiency and overhead of different LAW size, we propose and implement an effective LAW with its size dynamically adjusted to provide essential information for FAA and chunk cache and avoid unnecessary overhead.

The rest of the paper is arranged as follows. Section 2 reviews the background of data deduplication and the current schemes of improving restore performance. Section 3 compares and analyzes different caching schemes. We first present a scheme with the pre-determined and fixed sizes of the forward assembly area, chunk cache, and LAW in Section 4. Then, the adaptive algorithm is proposed and discussed in Section 5. A brief introduction of the prototype implementation is in Section 6 and the evaluation results and analyses are shown in Section 7. Finally, we provide some conclusions and discuss the future work in Section 8.

2 Background and Related Work

In this section, we first review the deduplication and restore process. Then, the related studies of improving restore performance are presented and discussed.

2.1 Data Deduplication Preliminary

Data deduplication is widely used in the secondary storage systems such as archiving and backup systems to improve the storage space utilization [8, 9, 10, 11, 12, 13, 14, 15]. Recently, data deduplication is also applied in the primary storage systems such as SSD array to make better trade-offs between cost and performance [16, 17, 18, 19, 20, 21, 22, 23]. To briefly summarize deduplication, as a data stream is written to the storage system, it is divided into data chunks, which are represented by a secure hash value called a fingerprint. The chunk fingerprints are searched in the indexing table to check their uniqueness. Only the new unique chunks are written to containers, and the original data stream is represented with a recipe consisting of a list of data chunk meta-information including the fingerprint.

Restoring the original data stream back is the reverse process of deduplication. Starting from the beginning of the recipe, the restore engine identifies the data chunk meta-information sequentially, accesses the chunks either from memory or from storage, and assembles the chunks in an assembling buffer in memory. To get a chunk from storage to memory, the entire container holding the data chunk will be read, and the container may be distant from the last accessed container. Once the buffer is full, data is flushed out to the requested client.

In the worst case, to assemble N duplicated chunks, we may need N container-reads. A straightforward solution to reduce the container-reads is to cache some of the containers or data chunks. Since some data chunks will be used very shortly after they are read into memory, these cached chunks can be directly copied from cache to the assembling buffer which can reduce the number of container-reads. Another way to reduce the number of container-reads is to store (re-write) some of the duplicated data chunks together with the unique chunks during the deduplication process in the same container. Therefore, the duplicated chunks and unique chunks will be read out together in the same container and thus avoids the needs of accessing these duplicated chunks from other containers. However, this approach will reduce the effectiveness of data deduplication.

2.2 Related Work on Restore Performance Improvement

Selecting and storing some duplicated data chunks during the deduplication process and designing efficient caching policies during the restore process are two major research directions to improve the restore performance. In the remaining of this subsection, we first review the studies of selectively storing the duplicated chunks. Then, we introduce the container-based caching, chunk-based caching and forward assembly.

There have been several studies focusing on how to select and store the duplicated data chunks to improve the restore performance. The duplicated data chunks have already been written to the storage when they first appeared as unique data chunks and dispersed over different physical locations in different containers, which creates the chunk fragmentation issue [3]. During the restore process, restoring these duplicated data chunks causes potential random container-reads which lead to a low restore throughput. Nam *et al.* [3, 4] propose a way to measure the chunk fragmentation level (CFL). By storing some of the duplicated chunks to keep the CFL lower than a given threshold in a segment of the recipe, the number of container-reads is reduced.

Kaczmarczyk *et al.* [2] use the mismatching degree between the stream context and disk context of the chunks to make the decision of storing selected duplicated data chunks. The container capping is proposed by Lillibridge *et al.* [1]. The containers storing the duplicated chunks are ranked and the duplicated data chunks in the lower ranking containers are selected and stored again. In a historical-based duplicated chunk rewriting algorithm [5], the duplicated chunks in the inherited sparse containers are rewritten. Due to the fact that rewriting some selected duplicated data chunks again sacrifices the deduplication ratio and the selecting and rewriting can be applied separately during the deduplication process, we will consider only the restore process in this paper.

Different caching policies are studied in [1, 2, 3, 4, 6, 7, 24, 25]. Kaczmarczyk *et al.* [2] and Nam *et al.* [3, 4] use container-based caching. Other than using recency to identify the victims in the cache, Park *et al.* [7] propose a future reference count based caching policy with the information from a fixed size look-head window. Belady's optimal replacement policy can only be used in a container-based caching schema [5]. It requires extra effort to identify and store the replacement sequence during the deduplication. If a smaller caching granularity is used, a better performance can be achieved. Instead of caching containers, some of the studies directly cache data chunks to achieve higher cache hit ratio [1, 6]. Although container-based caching has lower operating cost, chunk-based caching can better filter out the data chunks that are irrelevant to the near future restore and better improve the cache space utilization.

However, the chunk-based caching with LRU also has some performance issues. The historical based LRU may

Table 1: Data Sets

Data Set Name	ds_1	ds_2	ds_3
Deduplication Ratio	1.03	2.35	2.11
Reuse Distance (# containers)	24	18	26

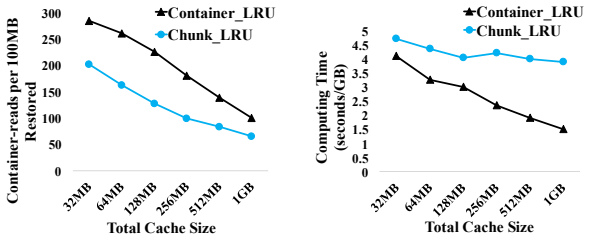
fail to identify data chunks in the read-in container which are not used in the past and current assembling buffers, but they will be used in the near future. This results in cache misses. To address this issue, a look-ahead window which covers a range of future accesses from the recipe can provide the crucial future access information to improve the cache effectiveness.

A special fashion of chunk-based caching proposed by Lillibridge *et al.* is called forward assembly [1]. Multiple containers (say k) are used as assembling buffers called Forward Assembly Area (FAA) and a look-ahead window of the same size is used to identify the data chunks to be restored in the next k containers. FAA can be considered as a chunk-based caching algorithm. It caches all the data chunks that appear in the next k containers and evicts any data chunk which does not appear in these containers. Since data chunks are directly copied from container-read buffer to FAA, it avoids the memory-copy operations from the container-read buffer to the cache. Therefore, forward assembly has lower overhead comparing with the chunk-based caching. Forward assembly can be very effective if each unique data chunk will reappear in a short range from the time it is being restored.

As discussed, there is still a big potential to improve the restore performance if we effectively combine forward assembly and chunk-based caching using the future access information in the LAW. In this paper, we consider the total amount memory available to FAA and chunk cache is fixed. If the memory allocation for these two can vary according to the locality changing, the number of container-reads may be further reduced.

3 Analysis of Cache Efficiency

Before we start to discuss the details of our proposed design, we first compare and analyze the cache efficiency of container-based caching, chunk-based caching, and forward assembly. The observations and knowledge we learned will help our design. The traces used in the experiments of this section are summarized in Table 1. ds_1 and ds_2 are the last version of EMC_1 and FSL_1 traces respectively which are introduced in detail in Section 7.1. ds_3 is a synthetic trace based on ds_2 with larger re-use distance. The container size is 4MB. Computing time is used to measure the management overhead. It is defined as the total restore time excluding the storage I/O time which includes the cache adjustment time, memory-copy time, CPU operation time, and others.



(a) # Containers-reads per 100MB restored as cache size varies from 32MB to 1GB

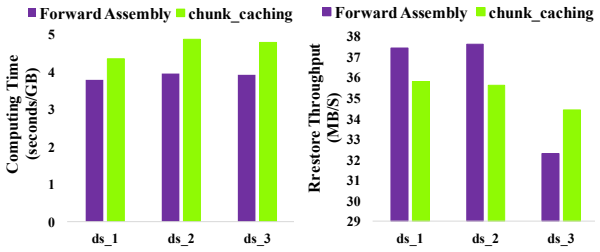
(b) Computing time per 1GB restored as cache size varies from 32MB to 1GB

Figure 1: The cache efficiency comparison between chunk-based caching and container-based caching

3.1 Caching Chunks vs. Caching Containers

Technically, caching containers can avoid the memory-copy from the container-read buffer to the cache. If the entire cache space is the same, the cache management overhead of container-based caching is lower than that of chunk-based caching. In most cases, some data chunks in a read-in container are irrelevant to the current and near-future restore process. Container-based caching still caches these chunks and wastes the valuable memory space. Also, some of the useful data chunks are forced to be evicted together with the whole container which increases the cache miss ratio. Thus, without considering managing overhead, caching chunks can achieve better cache hit ratio than caching containers if we apply the same cache policy in most workloads. This is especially true if we use LAW as a guidance of future accesses. Only in the extreme cases that most data chunks in the container are used very shortly and there is very high temporal based locality, the cache hit ratio of caching containers can be better than that of caching chunks.

We use the ds_2 trace to evaluate and compare the number of container-reads and the computing overhead of caching chunks and caching containers. We implemented the container-based caching and chunk-based caching with the same LRU policy. The assembling buffer used is one container size. As shown in Figure 1(b), the computing time of restoring 1GB data of caching chunks is about 15-150% higher than that of caching containers. Theoretically, the LRU cache insertion, lookup and eviction are $\mathcal{O}(1)$ time complexity. However, the computing time drops in both designs as the cache size increases. The reason is that with larger memory size more containers or data chunks can be cached and the cache eviction happens less frequently. There will be fewer memory-copy of containers or data chunks, which leads to less computing time.



(a) Computing time per 1GB restored (b) Restore throughput

Figure 2: The cache efficiency comparison between forward assembly and chunk-based caching

Also, due to fewer cache replacements in the container-based caching, the computing time of caching containers drops quicker than that of caching chunks. However, as shown in Figure 1(a), the number of container-reads of caching chunks is about 30-45% fewer than that of caching containers. If one container-read needs 30ms, about 3% fewer container-reads can cover the extra computing time overhead of caching chunks. Therefore, caching chunks is preferred in the restore process, especially when the cache space is limited.

3.2 Forward Assembly vs. Caching Chunks

Comparing with chunk-based caching, forward assembly does not have the overhead of cache replacement and the overhead of copying data chunks from container-read buffer to the cache. Thus, the computing time of forward assembly is much lower than that of chunk-based caching. As for the cache efficiency, the two approaches have their own advantages and disadvantages with different data chunk localities. If most of the data chunks are unique or the chunk re-use locality is high in a short range (e.g., within the FAA range), forward assembly performs better than chunk-based caching. In this scenario, most of the data chunks in the read-in containers will only be used to fill the current FAA. In the same scenario, if chunk-based caching can intelligently choose the data chunks to be cached based on the future access information in LAW, it can achieve a similar number of container-reads but it has larger computing overhead.

If the re-use distances of many duplicated data chunks are out of the FAA range, chunk-based caching performs better than forward assembly. Since these duplicated data chunks could not be allocated in the FAA, extra container-reads are needed when restoring these data chunks via forward assembly. In addition, a chunk in the chunk cache is only stored once while in forward assembly it needs to be stored multiple times in its appearing locations in the FAA. Thus, chunk-based caching can potentially cover more data chunks with larger re-use distances. When the managing overhead is less than the time saved by the reduction of container-reads, chunk-based caching performs better.

We use the aforementioned three traces to compare the efficiency of forward assembly with that of chunk-based caching. The memory space used by both schemes is 64MB. For chunk-based caching, it uses the same size LAW as FAA to provide the information of data chunks accessed in the future. It caches the data chunks that appear in the LAW first, then adopts the LRU as its caching policy to manage the rest of caching space. As shown in Figure 2(a), the computing time of forward assembly is smaller than that of chunk-based caching. The restore throughput of forward assembly is higher than that of chunk based caching except in ds_3 as shown in Figure 2(b). In ds_3, the re-use distances of 77% of the duplicated data chunks are larger than the FAA range. More cache misses occur in forward assembly, while chunk-based caching can effectively cache some of these data chunks. In general, if the deduplication ratio is extremely low or most of the chunk re-use distances can be covered by the FAA, the performance of forward assembly is better than that of chunk-based caching. Otherwise, chunk-based caching can be a better choice.

4 Look-Ahead Window Assisted Chunk-based Caching

As discussed in the previous sections, combining forward assembly with chunk-based caching can potentially adapt to various workloads and achieve better restore performance. In this section, we design a restore algorithm with the assumption that the sizes of FAA, chunk cache and LAW are all fixed. We call the memory space used by chunk-based caching *chunk cache*. We first discuss how the FAA, chunk-based caching and LAW cooperate together to restore the data stream. Then, a detailed example is presented to better demonstrate our design.

FAA consists of several container size memory buffers and they are arranged in a linear order. We call each container size buffer an FAB. A restore pointer pinpoints the data chunk in the recipe to be found and copied to its corresponding location in the first FAB at this time. The data chunks before the pointer are already assembled. Other FABs are used to hold the accessed data chunks if these chunks also appeared in these FABs. LAW starts with the first data chunk in the first FAB (FAB1 in the example) and covers a range bigger than that of FAA in the recipe. In fact, the first portion of the LAW (equivalent to the size of FAA) is used for the forward assembly purpose and the remaining part of LAW is used for the purpose of chunk-based caching.

An assembling cycle is the process to completely restore the first FAB in the FAA. That is, the duration after the previous first FAB is written back to the client

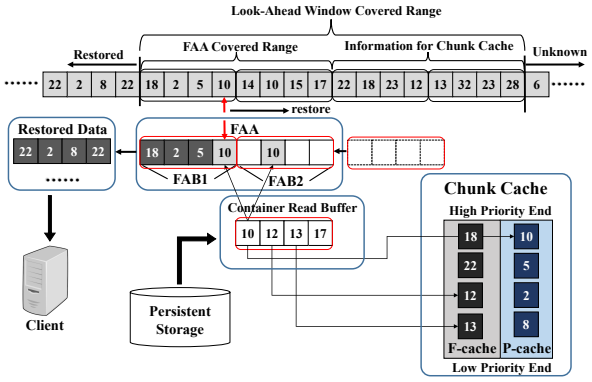


Figure 3: An example of look-ahead window assisted chunk-based caching

and a new empty FAB is added to the end of FAA to the time when the new first FAB is completely filled up. At the end of one assembling cycle (i.e., the first FAB is completely assembled), the content of the FAB is written back to the requested client as the restored data and this FAB is removed from the FAA. At the same time, a new empty FAB will be added to the end of the FAA so that FAA is maintained with the same size. For the LAW, the first portion (one container size) corresponding to the removed FAB is dropped and one more segment of recipe (also one container size) is appended to the end of LAW. Then, the restore engine starts the next assembly cycle to fill in the new first FAB of the current FAA.

During the assembling cycle, the following is the procedure of restoring the data chunk pointed by the restore pointer. If the data chunk at the restore pointer has already been stored by previous assembling operations, the pointer will be directly moved to the next one. If the data chunk has not been stored at its location in the FAB, the chunk cache is checked first. If the chunk is found in the cache, the data chunk is copied to the locations where this data chunk appears in all the FABs including the location pointed by the restore pointer. Also, the priority of the chunk in the cache is adjusted accordingly. If the chunk is not in the chunk cache, the container that holds the data chunk will be read in from the storage. Then, each data chunk in this container is checked with LAW to identify all the locations it appears in the FAA. Then, the data chunk is copied to the corresponding locations in all FABs if they exist. Next, we have to decide which chunks in this container are to be inserted to the chunk cache according to a caching policy, and it will be described later. After this data chunk is restored, the restore pointer moves to the next data chunk in the recipe. The read-in container will be replaced by the next requested container. An example is shown in Figure 3, and it will be described in detail in the last part of this section.

When inserting data chunks from the read-in container to the chunk cache, we need to identify the potential usage of these chunks in the LAW and treat them accordingly. Based on the second portion of LAW (i.e., the remaining LAW after the size of FAA), the data chunks from the read-in container can be classified into three categories: 1) *U-chunk* (Unused chunk) is a data chunk that does not appear in the current entire LAW, 2) *P-chunk* (Probably used chunk) is a data chunk that appears in the current FAA but does not appear in the second portion of the LAW, and 3) *F-chunk* (Future used chunk) is a data chunk that will be used in the second portion of the LAW. Note that a *F-chunk* may or may not be used in the FAA.

F-chunks are the data chunks that should be cached. If more cache space is still available, we may cache some *P-chunks* according to their priorities. That is, *F-chunks* have priority over *P-chunks* for caching. However, each of them has a different priority policy. The priority of *F-chunks* is defined based on the ordering of their appearance in the second portion of the LAW. That is, an *F-chunk* to be used in the near future in the LAW has a higher priority over another *F-chunk* which will be used later in the LAW. The priority of *P-chunks* is LRU based. That is, the most recently used (MRU) *P-chunk* has a higher priority over the least recently used *P-chunks*. Let us denote the cache space used by *F-chunks* (*P-chunks*) as F-cache (P-cache). The boundary between the F-cache and P-cache is dynamically changing as the number of *F-chunks* and that of *P-chunks* vary.

When the LAW advances, new *F-chunks* are added to the F-cache. An *F-chunk* that has been restored and no longer appeared in the LAW will be moved to the MRU end of the P-cache. That is, this chunk becomes a *P-chunk*. The priorities of some *F-chunks* are adjusted according to their future access sequences. The new *P-chunks* are added to the P-cache based on the LRU order of their last appearance. Some *P-chunks* may become *F-chunks* if they appear in the newly added portion of LAW. When the cache eviction happens, data chunks are evicted from the LRU end of the P-cache first. If there is no *P-chunk* in P-cache, eviction starts from the lowest priority end of the F-cache.

Figure 3 is an example to show the entire working process of our design. Suppose one container holds 4 data chunks. The LAW covers the data chunks of 4 containers in the recipe from chunk 18 to chunk 28. The data chunks before the LAW have been assembled and written back to the client. The data chunks beyond the LAW is unknown to the restore engine at this moment. There are two buffers in the FAA denoted as FAB1 and FAB2. Each FAB has a size of one container and also holds 4 data chunks. FAA covers the range from chunk 18 to chunk 17. The rest information of data chunks in the LAW (from chunk 22 to chunk 28) are used for the chunk

cache. The red frames in the figure show the separations of data chunks in containers. The labeled chunk number represents the chunk ID and is irrelevant to the order of the data chunk appearing in the recipe.

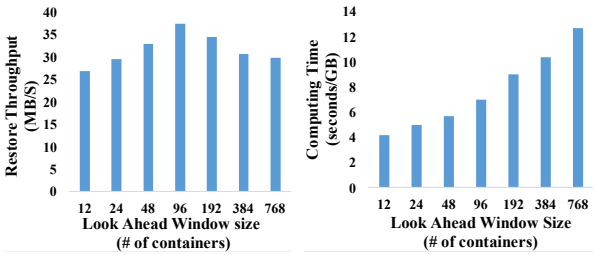
In FAB1, chunks 18, 2, 5 have already been stored and the current restore pointer is at data chunk 10 (pointed by the red arrow). This data chunk has neither been allocated nor been cached. The container that stores chunk 10, 12, 13 and 17 is read out from the storage to the container read buffer. Then, the data of chunk 10 is copied to the FAB1. At the same time, chunk 10 also appears in the FAB2 and the chunk is stored in the corresponding position too. Next, the restore pointer moves to the chunk 14 at the beginning of the FAB2. Since FAB1 has been fully assembled, its content is written out as restored data and it is removed from FAA. The original FAB2 becomes the new FAB1 and a new FAB (represented with dotted frames) is added after FAB1 and becomes the new FAB2.

All the data chunks in the container read buffer are checked with the LAW. Data chunk 10 is used in the FAA but it does not appear again in the rest of LAW. So chunk 10 is a *P-chunk* and it is inserted to the MRU end of the P-cache. Chunk 12 and chunk 13 are not used in the current FAA but they will be used in the next two assembling cycles within the LAW. They are identified as *F-chunks* and added to the F-cache. Notice that chunk 12 appears after chunk 22 and chunk 13 is used after chunk 12 as shown in the recipe. Therefore, chunk 13 is inserted into the low priority end and chunk 12 has the priority higher than chunk 13. Chunk 17 has neither been used in the FAA nor appeared in the LAW. It is a *U-chunk* and it will not be cached. When restoring chunk 14, a new container will be read into the container read buffer and it will replace the current one.

5 The Adaptive Algorithm

The performance of the look-ahead window assisted chunk-based caching is better than that of simply combining the forward assembly with LRU-based chunk caching. However, the sizes of FAA, chunk cache and LAW are pre-determined and fixed in this design. As discussed in the previous sections, FAA and chunk cache have their own advantages and disadvantages for different workloads. In fact, in a given workload like a backup trace, most data chunks are unique at the beginning. Later in different sections of the workload, they may have various degrees of duplication and re-usage.

Therefore, the sizes of FAA and chunk cache can be dynamically changed to reflect the locality of the current section of a workload. It is also important to figure out what the appropriate LAW size is to get the best restore performance given a configuration of the FAA and chunk cache. We propose an adaptive algorithm called



(a) Restore throughput (b) Computing time per 1GB restored

Figure 4: The restore performance and computing overhead variation as the LAW size increases

ALACC that can dynamically adjust the sizes of FAA, chunk cache and LAW according to the workload variation during the restore process. First, we evaluate and analyze the restore performance of using different LAW sizes. Then, we present the details of ALACC.

5.1 Performance Impact of LAW Size

We did an experiment that varies the LAW sizes for a given workload and compared the restore throughput and required computing overhead. We use an 8MB (2 container size) FAA and a 24MB (6 container size) chunk cache as the memory space configuration and increase the LAW size from 12 container size to 768 container size (it covers the original data stream size from 48MB to 3GB). As shown in Figure 4, the computing time continuously increases as the LAW size increases due to higher overhead to process and maintain the information in the LAW. When the size of LAW is larger than 96, the restore throughput starts to decrease. The performance degradation is caused by the increase of computing overhead and less efficiency of chunk cache. Thus, using an appropriate LAW size is important to make better trade-offs between cache efficiency and computing overhead.

We can explain the observation by analyzing the chunk cache efficiency. Suppose the LAW size is S_{LAW} chunks, the FAA size is S_{FAA} chunks, and the chunk cache size is S_{cache} chunks. Note that we use a container size as the basic unit for allocation and the number of containers can be easily translated to the number of chunks. Assume one data chunk C_i is used at the beginning of the FAA and it will be reused after D_{C_i} chunks (i.e., the reuse distance of this chunk). If $D_{C_i} < S_{FAA}$, it will be reused in the FAA. If $S_{FAA} < D_{C_i}$, this chunk should be either an *F-chunk* or a *P-chunk*. However, the chunk category highly depends on the size of S_{LAW}. If the LAW only covers the FAA range ($S_{FAA} = S_{LAW} < D_{C_i}$), the proposed algorithm degenerates to the LRU-based chunk caching algorithm. If $S_{FAA} < D_{C_i} < S_{LAW}$, we can definitely identify this chunk as an *F-chunk* and decide to cache this chunk. However, if $S_{LAW} < D_{C_i}$, this chunk will be identified as

a *P-chunk* and it may or may not be cached (this depends on the available space in cache). Therefore, at least we should ensure $S_{FAA} + S_{cache} \leq S_{LAW}$.

If the LAW size is large enough and most of the chunk cache space are occupied by *F-chunks*, the cache efficiency is high. However, once the cache is full of *F-chunks*, the newly identified *F-chunk* may or may not be inserted into the cache. This depends on its reuse order. Thus, continuing increase the LAW size would not further improve the cache efficiency. What is worse, a larger LAW size requires more CPU and memory resources to identify and maintain the future access information. As we discuss before, if S_{LAW} is the same as S_{FAA}, all cached chunks are *P-chunks* and chunk caching becomes an LRU-based algorithm.

Thus, the best trade-off between cache efficiency and overhead is when the total number of *P-chunks* is as low as possible but not 0. However, it may be hard to maintain the number of *P-chunks* low all the time, especially when there is a very limited number of *F-chunks* identified with a large LAW size. In this case, the size of LAW should not be extended or it should be decreased slightly.

5.2 ALACC

Based on the previous analysis, we propose an adaptive algorithm, which dynamically adjusts the memory space ratio of FAA and chunk cache, and the size of LAW. We apply the adjustment at the end of each assembling cycle right after the first FAB is restored. Suppose at the end of i_{th} assembling cycle, the sizes of FAA, chunk cache and LAW are S^i_{FAA}, S^i_{cache} and S^i_{LAW} container size respectively. $S^i_{FAA} + S^i_{cache}$ is fixed. To avoid an extremely large LAW size, we set a maximum size of LAW (Max_{LAW}) that is allowed during the restore.

The algorithm of ALACC optimizes the memory allocation to FAA and chunk cache first. On one hand, if the workload has an extremely low locality or its duplicated data chunk re-use distance is small, FAA is preferred. On the other hand, according to the total number of *P-chunk* in the cache, the chunk cache size is adjusted. Then, according to the memory space allocation changing and the total number of *F-chunk* in the cache, the LAW size is adjusted to optimize the computing overhead. The detailed adjustment conditions and actions are described in the following part of this section.

In our algorithm, the conditions of increasing the FAA size are examined first. If any of these conditions is satisfied, FAA will be increased by 1 container size and the size of chunk cache will be decreased by 1 container accordingly. Otherwise, we will check the conditions of adjusting the chunk cache size. Notice that the size of FAA and chunk cache could remain the same if none of the adjustment conditions are satisfied. Finally, the LAW size will be changed.

FAA and LAW Size Adjustment. As discussed in Section 3.2, FAA performs better than chunk cache when 1) the data chunks in the first FAB are identified mostly as unique data chunks and these chunks are stored in the same or close containers, or 2) the re-use distance of most duplicated data chunks in this FAB is within the FAA range. Regarding the first condition, we consider that the FAB can be filled in by reading in no more than 2 containers and none of the data chunks needed by the FAB is from the chunk cache. When this happens, we consider this assembling cycle FAA effective. For Condition 2, we observed that if the re-use distances of 80% or more of the duplicated chunks in this FAB is smaller than the FAA size, forward assembly performs better.

Therefore, based on the observations, we use either of the following two conditions to increase the FAA size. First, if the number of consecutive FAA effective assembling cycles becomes bigger than a given threshold, we increase the FAA size by 1 container. Here, we use the current FAA size, S^i_{FAA}, as the threshold to measure this condition at the end of i_{th} assembling cycle. When S^i_{FAA} size is small, the condition is easier to satisfy and the size of FAA can be increased faster. When S^i_{FAA} is large, the condition is more difficult to satisfy. After increasing the FAA size by one, the count of consecutive FAA effective assembling cycles is reset to 0. Second, the data chunks used during the i_{th} assembling cycle to fill up the first FAB in FAA are examined. If the re-use distances of more than 80% of these examined chunks during the i_{th} assembling cycle are smaller than $S^i_{FAA} + 1$ container size, the size of FAA will be increased by 1 container. That is, $S^{i+1}_{FAA} = S^i_{FAA} + 1$.

If the FAA size is increased by 1 container, the size of chunk cache will decrease by 1 container accordingly. Originally, $S^i_{LAW} - S^i_{FAA}$ container size LAW information is used by S^i_{cache} container size cache. After the FAA adjustment, $S^i_{LAW} - S^i_{FAA} + 1$ container size LAW is used by $S^i_{cache} - 1$ container size cache, which wastes the information in the LAW. Thus, the LAW size is decreased by 1 container size to avoid the same size LAW used by a now smaller size chunk cache. After the new sizes of FAA, chunk cache and LAW are decided, two empty FABs (one to replace the re-stored FAB and the other reflects the increasing size of FAA) will be added to the end of FAA and the chunk cache starts to evict data chunks. Then, the $(i+1)_{th}$ assembling cycle will start.

Chunk Cache and LAW Size Adjustment. If there is no adjustment to the FAA size, we now consider the adjustment of chunk cache size. After finished the i_{th} assembling cycle, the total number ($N_{F-chunk}$) of *F-chunks* in the F-cache and the total number ($N_{P-chunk}$) of *P-chunks* in the P-cache are counted. Also, the number of *F-chunks* that are newly added to the F-cache during the i_{th} assembling cycle is denoted by $N_{F-added}$. These

newly added *F-chunks* either come from the read-in containers in the i_{th} assembling cycle or are transformed from *P-chunks* due to the extending of the LAW. We examine the following three conditions.

First, if $N_{P-chunk}$ becomes 0, it indicates that all the cache space is occupied by *F-chunks*. The current LAW size is too large and the number of *F-chunks* based on the current LAW is larger than the chunk cache capacity. Therefore, the chunk cache size will be increased by 1 container. Meanwhile, the size of LAW will decrease by 1 container to reduce the unnecessary overhead. Second, if the total size of $N_{F-added}$ is bigger than 1 container, it indicates that the total number of *F-chunks* increases very quickly. Thus, we increase the chunk cache size by 1 container and decrease LAW by 1 container. Notice that a large $N_{F-added}$ can happen when $N_{P-chunk}$ is either small or large. This condition will make our algorithm quickly react to the changing of the workload.

Third, if $N_{P-chunk}$ is very large (i.e., $N_{F-chunk}$ is very small), the chunk cache size will be decreased by 1 container. In this situation, the LAW size is adjusted differently according to either of the following two conditions: 1) In the current workload, there are few data chunks in the FAB that are reused in the future, and 2) The size of LAW is too small, it cannot look ahead far enough to find more *F-chunks* for the current workload. For Condition 1, we decrease the LAW size by 1 container. For Condition 2, we increase the LAW size by K containers. Here, K is calculated by $K = (Max_{LAW} - S^i_{LAW})/(S^i_{FAA} + S^i_{cache})$. If LAW is small, its size is increased by a larger amount. If LAW is big, its size will be increased slowly.

LAW Size Independent Adjustment If none of the aforementioned conditions are satisfied (the sizes of FAA and chunk cache remain the same), the LAW size will be adjusted independently. Here, we use the $N_{F-chunk}$ to decide the adjustment. If $N_{F-chunk}$ is smaller than a given threshold (e.g., 20% of the total chunk cache size), the LAW size will be slightly increased by 1 container to process more future information. If $N_{F-chunk}$ is higher than the threshold, the LAW size will be decreased by 1 to reduce the computing overhead.

ALACC makes the trade-offs between computing overhead and the reduction of container-reads, such that a higher restore throughput can be achieved. Instead of using a fixed value as the threshold, we tend to dynamically change FAA size and LAW size. It can slow down the adjustments when the overhead is big (when the LAW size is large) and it can speed up the adjustment when the overhead is small to quickly reduce the container-reads (when FAA size is small).

6 Prototype Implementation

We implemented a prototype of a deduplication system (a C program with 11k LoC) with several restore designs: FAA, container-based caching, chunk-based caching, LAW assisted chunk-based caching, and ALACC. For the deduplication process, it can be configured to use different container size and chunk size (fixed size chunking and variable size chunking) to process the real world data or to generate deduplication traces.

To satisfy the flexibility and efficiency requirements of ALACC, we implemented several data structures. All the data chunks in the chunk cache are indexed by a hashmap to speed up the searching operation. *F-chunks* are ordered by their future access sequence provided by the LAW and *P-chunks* are indexed by an LRU list. The LAW maintains the data chunk metadata (chunk ID, size, container ID, address and offset in the container) in the same order as they are in the recipe. Although the insertion and deletion in the LAW is $\mathcal{O}(1)$ by using the hashmap, identifying the *F-chunk* priority is $\mathcal{O}(\log(N))$, where N is the LAW size.

A Restore Recovery Log (RRL) is maintained to ensure reliability. When one FAB is full and flushed out, the chunk ID of the last chunk in the FAB, the chunk location in the recipe, the restored data address, FAA, chunk cache and LAW configurations are logged to the RRL. If the system is down, by using the information in the RRL, the restore process can be recovered to the latest restored cycle. The FAA, chunk cache and LAW will be initiated and reconstructed.

7 Performance Evaluation

To comprehensively evaluate our design, we implement five restore engines including ALACC, LRU-based container caching (Container_LRU), LRU-based chunk caching (Chunk_LRU), forward assembly (FAA), and fixed combination of forward assembly and chunk-based caching. However, in the last case, we show only the Optimal Fix Configuration (Fix_Opt). Fix_Opt is obtained by exhausting all possible fixed combinations of FAA, chunk cache, and LAW sizes.

7.1 Experimental Setup and Data Sets

The prototype is deployed on a Dell PowerEdge R430 server with a 2.40GHz Intel Xeon with 24 cores and 32GB of memory using Seagate ST1000NM0033-9ZM173 SATA hard disk with 1TB capacity as the storage. The container size is configured as 4MB. All five implementations are configured with one container size space as container read buffer and memory size of S containers. Container_LRU and Chunk_LRU use one container for FAA and $S-1$ for container or chunk cache.

Table 2: Characteristics of datasets

Dataset	FSL_1	FSL_2	EMC_1	EMC_2
Size	103.5GB	317.4GB	29.2GB	28.6GB
ACS[1]	4KB	4KB	8KB	8KB
DR[2]	3.82	4.88	1.04	4.8
CFL[3]	13.3	3.3	14.7	19.3

[1] ACS stands for Average Chunk Size

[2] DR stands for the Deduplication Ratio, which is the original data size divided by the deduplicated data size.

[3] CFL stands for the Chunk Fragmentation Level, which is the average number of containers that stores the data chunks from one container size of original data stream. High CFL value leads to low restore performance.

FAA uses LAW of S container size and all S memory space as the forward assembly area. The specific configuration of Fix_Opt is given in Section 7.2. In all experiments, ALACC is initiated with $S/2$ container size FAA, $S/2$ container size chunk cache and $2S$ container size LAW before the execution. For the reason that ALACC requires to maintain at least one container size space as FAB, the FAA size varies from 1 to S and the chunk cache size varies from $(S-1)$ to 0 accordingly. After one version of backup is restored, the cache is cleaned.

We use four deduplication traces in the experiments as shown in Table 2. FSL_1 and FSL_2 are two different backup traces from FSL /home directory snapshots of the year 2014 [26]. Each trace has 6 full snapshot backups and the average chunk size is 4KB. EMC_1 and EMC_2 are the weekly full-backup traces from EMC and each trace has 6 versions and 8KB average chunk size [27]. EMC_1 was collected from an exchange server and EMC_2 was the */var* directory backup from a revision control system.

To measure the restore performance, we use the *speed factor*, *computing cost factor* and *restore throughput* as the metrics. The speed factor (MB/container-read) is defined as the mean data size restored per container read. Higher speed factors indicate higher restore performance. The computing cost factor (second/GB) is defined as the time spent on computing operations (subtracting the storage I/O time from the restore time) per GB data restored and the smaller value is preferred. The restore throughput (MB/second) is calculated from the original data stream size divided by the total restore time. We run each test 5 times and present the mean value. Notice that, for the same trace using the same caching policy, if the restore machine and the storage are different, the computing cost factor and restore throughput can be different while the speed factor is the same.

Table 3: The FAA/chunk cache/LAW configuration (# of containers) of Fix_Opt for each deduplication trace

	FSL_1	FSL_2	EMC_1	EMC_2
Size	4/12/56	6/10/72	2/14/92	4/12/64

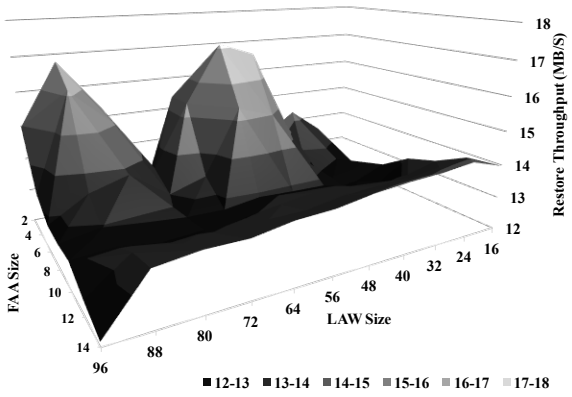


Figure 5: The restore throughput of different FAA, chunk cache and LAW size configurations

7.2 Optimal Performance of Fixed Configurations

In LAW assisted chunk-based caching design, the sizes of FAA, chunk cache and LAW are fixed and are not changed during the restore process. To find out the best performance of a configuration for a specific trace with a given memory size, in our experiments we run all possible configurations for each trace to discover the optimal throughput. This optimal configuration is indicated by Fix_Opt. For example, for 64MB memory, we vary the FAA size from 4MB to 60MB and the chunk cache size from 60MB to 4MB. At the same time, the LAW size increases from 16 containers to 96 containers. Each test tries one set of fixed configuration and finally, we draw a three-dimensional figure to find out the optimal results.

An example is shown in Figure 5, for FSL_1, the optimal configuration has 4 containers of FAA, 12 containers of chunk cache and a LAW size of 56 containers. One throughput peak is when the LAW is small. The computing cost is low while the container-reads are slightly higher. The other throughput peak is when the LAW is relatively large. With more future information, the container-reads are lower but the computing cost is higher. The optimal configuration of each trace is shown in Table 3, the sizes of FAA and chunk cache can be calculated by the number of containers times 4MB.

7.3 Restore Performance Comparison

Using the same restore engine and storage as discovering the Fix_Opt, we evaluate and compare the speed fac-

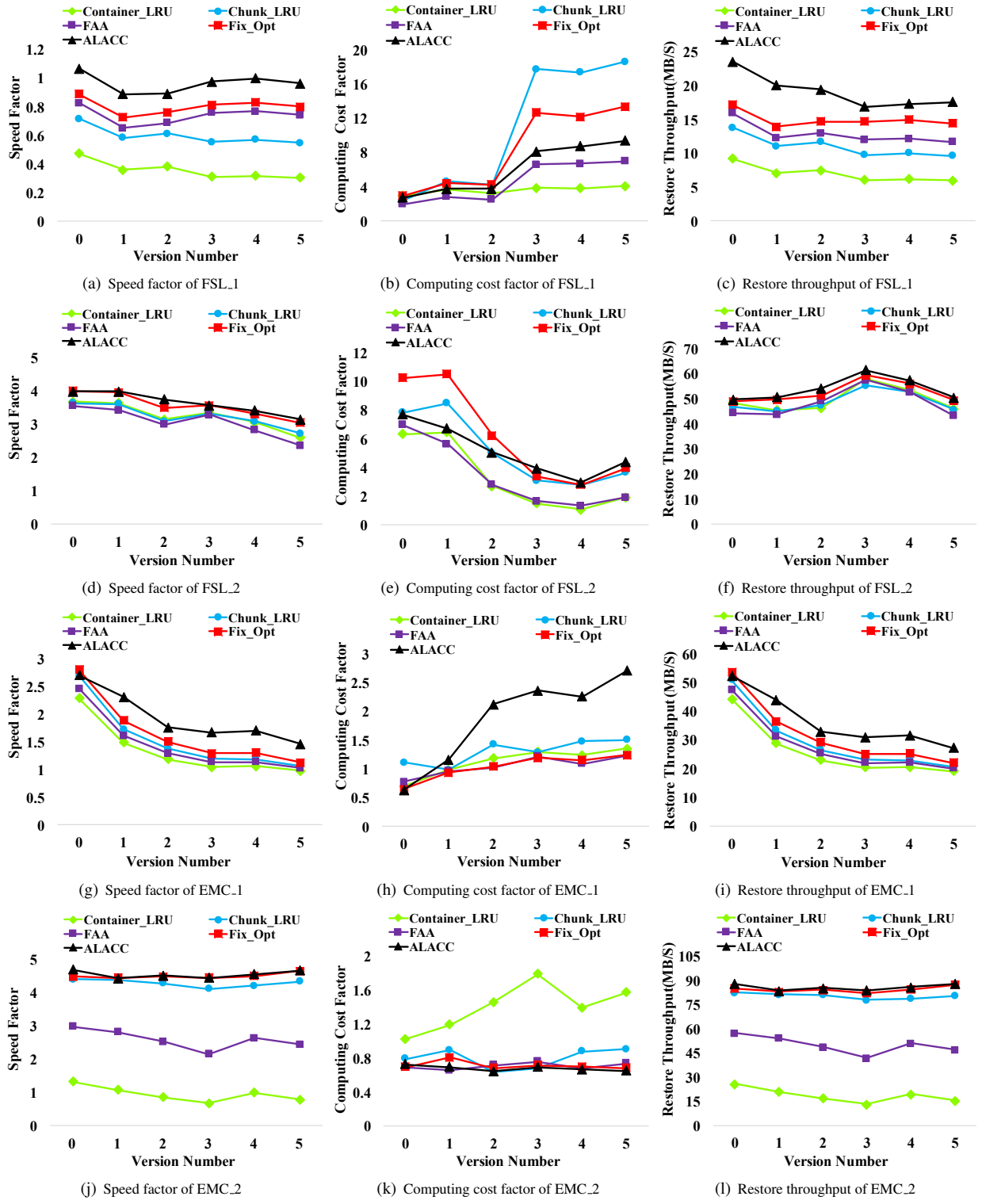


Figure 6: The restore performance results comparison of Container_LRU, Chunk_LRU, FAA, Fix_Opt and ALACC. **Notice** that the speed factor, computing cost factor and restore throughput vary largely in different traces, we use different scales among subfigures to show the relative improvement or difference of the five designs in the same trace.

Table 4: The percentage of memory size occupied by FAA of ALACC in each restore testing case

Version #	0	1	2	3	4	5
FSL_1	50%	38%	39%	38%	40%	38%
FSL_2	67%	67%	64%	69%	64%	57%
EMC_1	26%	24%	14%	17%	16%	16%
EMC_2	7%	8%	8%	8%	8%	7%

Table 5: The average LAW size (# of containers) of ALACC in each restore testing case

Version #	0	1	2	3	4	5
FSL_1	31.2	30.5	31.4	32.2	32.0	30.7
FSL_2	44.6	44.1	40.1	32.3	32.8	36.9
EMC_1	77.1	83.1	88.7	84.2	76.1	82.6
EMC_2	95.3	95.2	95.1	95.3	94.8	95.2

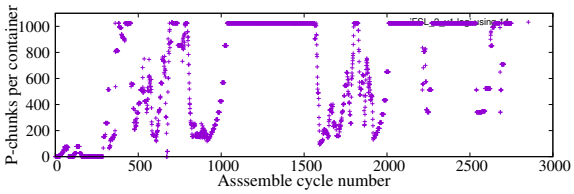
tor, computing cost and restore performance (throughput) of the four traces with 64MB total memory. The evaluation results are shown in Figure 6. As indicated in Table 2, the CFL of FSL_2 is much lower than the others, which leads to a very close restore performance of all the restore designs. However, when the CFL is high, ALACC is able to adjust the sizes of FAA, chunk cache, and LAW to adapt to the highly fragmented chunk storing, and achieves higher restore performance.

The computing cost varies in different traces and versions. In most cases, the computing cost of FAA is relatively lower than the others, because it avoids the cache insertion, look-up, and eviction operations. As expected, the computing overhead of Chunk_LRU is usually higher than that of Container_LRU due to more management operations for smaller caching granularity. However, in EMC_2, the computing overhead of Container_LRU is much higher than the other four designs. The overhead is caused by the high frequent cache replacement and extremely high cache miss ratio. The cache miss ratio of Container_LRU is about 3*X* higher than that of Chunk_LRU. The time of reading containers from storage to the read-in buffer dominates the restore time. Thus, the speed factor can nearly determine the restore performance. Comparing the speed factor and restore throughput of the same trace, for example, trace FS_1 in Figure 6(a) and 6(c), the curves of the same cache policy are very similar.

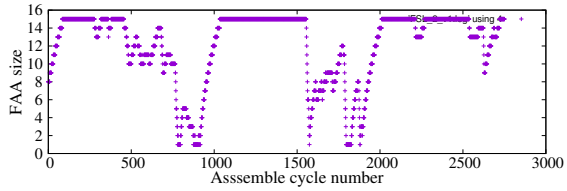
For all 4 traces, the overall average speed factor of ALACC is 83% higher than Container_LRU, 37% higher than FAA, 12% higher than Chunk_LRU and 2% higher than Fix_Opt. The average computing cost of ALACC is 27% higher than that of Container_LRU, 23% higher than FAA, 33% lower than Chunk_LRU and 26% lower than Fix_Opt. The average restore throughput of ALACC is 89% higher than that of Container_LRU, 38% higher than FAA, 14% higher than Chunk_LRU and 4% higher than Fix_Opt. In our experiments, the speed factor of ALACC is higher than those of Container_LRU, Chunk_LRU, and FAA. More importantly, ALACC achieves at least a similar or better performance as Fix_Opt. By dynamically adjusting the sizes of FAA, chunk cache and LAW, the improvement of the restore throughput is higher than the speed factor. Notice that we need tens of experiments to find out the optimal configurations of Fix_Opt which is almost impossible to carry out in a real-world production scenario.

The main goal of ALACC is to make better tradeoffs between the number of container-reads and the required computing overhead. The average percentage of the memory space occupied by FAA of ALACC is shown in Table 4 and the average LAW size is shown in Table 5. The percentage of chunk cache in the memory can be calculated by $(1 - FAA$ percentage$)$. The mean FAA size and LAW size vary largely in different workloads. The restore data with larger data chunk re-use distance usually needs smaller FAA size, larger cache size, and larger LAW size, like in traces FSL_1, FSL_2, and EMC_2. One exception is trace EMC_1. This trace is very special, only about 4% data chunks are duplicated chunks and they are scattered over many containers. The performances of Container_LRU, Chunk_LRU and FAA are thus very close since extra container-reads will always happen when restoring the duplicated chunks. By adaptively extending the LAW to a larger size (about 80 containers and 5 times larger than FAA cover range) and using larger chunk cache space, ALACC successfully identifies the data chunks that will be used in far future and caches them. Therefore, ALACC can outperform others in such an extreme workload. Assuredly, ALACC has the highest computing overhead (about 90% higher than others in average) as shown in Figure 6(h).

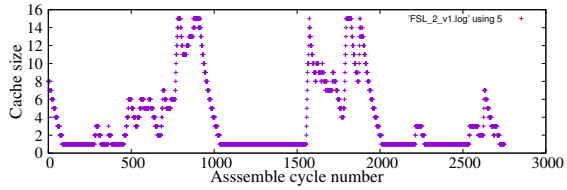
Comparing the trend of varying FAA and LAW sizes of Fix_Opt (shown in Table 3) with that of ALACC (shown in Tables 4 and 5), we can find that ALACC usually applies smaller LAW and larger cache size than Fix_Opt. Thus, ALACC achieves lower computing cost and improves the restore throughput as shown in Figures 6(b), 6(e) and 6(k). In EMC_2, ALACC has a larger cache size and a larger LAW size than those of Fix_Opt. After we exam the restore log, we find that the *P-chunks* occupied 95% the cache space in more than 90% of the assembling cycles. A very small portion of data chunks is duplicated many times, which can explain why the Chunk_LRU performs close to Fix_Opt. In such an extreme case, ALACC makes the decision to use a larger cache space and a larger LAW size such that it can still adapt to the workload and maintain a high speed factor



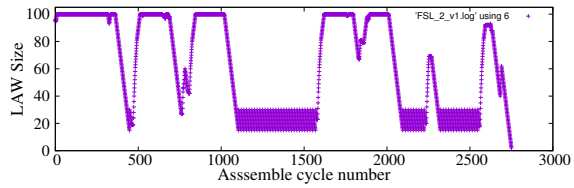
(a) P-chunk numbers per container size chunk cache



(b) FAA size (# of containers)



(c) Chunk cache size (# of containers)



(d) LAW size (# of containers)

Figure 7: The variation of P-chunk numbers per container size chunk cache, FAA size, chunk cache size, and LAW size during the restore of FSL_1 in version 0

and high restore throughput as shown in Figures 6(j) and 6(l). In general, ALACC successfully adapts to the locality changing and delivers high restore throughput for a given workload.

7.4 The Adaptive Adjustment in ALACC

To verify the adaptive adjustment process of ALACC, we write the log at the end of each assembling cycle and using *P-chunk* as an example to show the size adjustment of FAA, chunk cache and LAW by ALACC. The log records the *P-chunk* numbers per container size cache , the sizes of FAA, chunk cache, and LAW. We use FSL_2 version 1 as an example and the results are shown in Figure 7. The number of *P-chunks* per container size cache is very low at the beginning and varies sharply as assembly cycle increases as shown in Figure 7(a). One container size cache can store about 1000 data chunks in average. During the assembling cycle range (1000–1500), most of the chunk cache space is occupied by the *P-chunks* and there are few duplicated data chunks. Thus, ALACC uses a larger FAA and a smaller LAW.

If the number of *P-chunk* is relatively low, more caching space is preferred. For example, in the assembling cycle range (700–900), the number of *P-chunks* is lower than 200 (i.e., more than 80% of the chunk cache space is used for *F-chunks*). As expected, the FAA size drops quickly and the chunk cache size increases sharply and stays at a high level. Meanwhile, since the cache space is increased, the LAW size is also increased to cover larger recipe range and to identify more *F-chunks*. In general, ALACC successfully monitors the workload variation and self-adaptively reacts to the number of *P-chunks* variation as expected, and thus, delivers higher restore throughput without manual adjustments.

8 Conclusion and Future Work

Improving restore performance of deduplication system is very important. In this paper, we studied the effectiveness and the efficiency of different caching mechanisms applied to the restore process. Based on the observations of the caching efficiency experiments, we design an adaptive algorithm called ALACC which is able to adaptively adjust the sizes of the FAA, chunk cache and LAW according to the workload changes. By making better trade-offs between the number of container-reads and computing overhead, ALACC achieves much better restore performance than container-based caching, chunk-based caching and forward assembly. In our experiments, the restore performance of ALACC is slightly better than the best performance of restore engine with all possible configurations of fixed sizes of FAA, chunk cache and LAW. In our future work, duplicated data chunk rewriting will be investigated and integrated with ALACC to further improve the restore performance of data deduplication systems for both primary and secondary storage systems.

Acknowledgments

We thank all the members in CRIS group to provide the useful comments to improve our design. We thank Dongchul Park for assistance with the trace exploring and pre-processing, and Baoquan Zhang for the specific reviewing comments. We would like to thank our shepherd, Philip Shilane, for his useful comments and suggestions. This work is partially supported by the following NSF awards: 1305237, 1421913, 1439622 and 1525617.

References

[1] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 183–198, 2013.

[2] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 15. ACM, 2012.

[3] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 581–586. IEEE, 2011.

[4] Young Jin Nam, Dongchul Park, and David HC Du. Assuring demanded read performance of data deduplication storage with backup datasets. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 201–208. IEEE, 2012.

[5] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *USENIX Annual Technical Conference (USENIX ATC 14)*, pages 181–192, 2014.

[6] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. Sar: Ssd assisted restore optimization for deduplication-based storage systems in the cloud. In *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*, pages 328–337. IEEE, 2012.

[7] Dongchul Park, Ziqi Fan, Young Jin Nam, and David HC Du. A lookahead read cache: Improving read performance for deduplication backup storage. *Journal of Computer Science and Technology*, 32(1):26–40, 2017.

[8] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, volume 8, pages 1–14, 2008.

[9] Wei Zhang, Tao Yang, Gautham Narayanasamy, and Hong Tang. Low-cost data deduplication for virtual machine backup in cloud storage. In *HotStorage*, 2013.

[10] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011.

[11] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 331–344, 2015.

[12] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[13] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*, pages 1–9. IEEE, 2009.

[14] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, volume 2, pages 89–101, 2002.

[15] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: A scalable secondary storage. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*, volume 9, pages 197–210, 2009.

[16] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, volume 12, pages 1–14, 2012.

[17] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cachededup: in-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, 2016.

[18] Biplob K Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *USENIX annual technical conference (USENIX ATC 10)*, 2010.

[19] Yoshihiro Tsuchiya and Takashi Watanabe. Dblk: Deduplication for primary block storage. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–5. IEEE, 2011.

[20] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *2014 Ottawa Linux Symposium*, 2014.

[21] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, 2016.

[22] Zhuan Chen and Kai Shen. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 291–299, 2016.

[23] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, 2012.

[24] Ziqi Fan, David HC Du, and Doug Voigt. H-arc: A non-volatile memory based cache policy for solid state drives. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–11. IEEE, 2014.

[25] Ziqi Fan, Fenggang Wu, Dongchul Park, Jim Diehl, Doug Voigt, and David HC Du. Hibachi: A cooperative hybrid cache with nvram and dram for storage arrays. In *Mass Storage Systems and Technologies (MSST), 2017 IEEE 33th Symposium on*. IEEE, 2017.

[26] http://tracer.filesystems.org/.

[27] Nohhyun Park and David J Lilja. Characterizing datasets for data deduplication in backup applications. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

UKSM: Swift Memory Deduplication via Hierarchical and Adaptive Memory Region Distilling

Nai Xia[†] *Chen Tian*[†] *Yan Luo*[‡] *Hang Liu*[‡] *Xiaoliang Wang*[†]
[†]*State Key Laboratory for Novel Software Technology, Nanjing University, China*
[‡]*Department of Electrical and Computer Engineering, University of Massachusetts Lowell, USA*
{*xianai, tianchen, waxili*}*@nju.edu.cn,* {*Yan_Luo, Hang_Liu*}*@uml.edu*

Abstract

In cloud computing, deduplication can reduce memory footprint by eliminating redundant pages. The responsiveness of a deduplication process to newly generated memory pages is critical. State-of-the-art Content Based Page Sharing (CBPS) approaches lack responsiveness as they equally scan every page while finding redundancies. We propose a new deduplication system UKSM, which prioritizes different memory regions to accelerate the deduplication process and minimize application penalty. With UKSM, memory regions are organized as a distilling hierarchy, where a region in a higher level receives more CPU cycles. UKSM adaptively promotes/demotes a region among levels according to the region's estimated deduplication benefit and penalty. UKSM further introduces an adaptive partial-page hashing scheme which adjusts a global page hashing strength parameter according to the global degree of page similarity. Experiments demonstrate that, with the same amount of CPU cycles in the same time envelop, UKSM can achieve up to 12.6× and 5× more memory saving than CBPS approaches on static and dynamic workloads, respectively.

1 Introduction

In cloud computing, multiple virtual machines (VMs)/ containers (*e.g.*, dockers)/ processes are consolidated to share a physical sever. For a public cloud, the more VMs that can be packed into one host, the more VMs can be sold to tenants. For a private cluster, the more processes that can be packed into one host, the fewer the number of hosts needs to be purchased and maintained. In this context, available memory space can be a major bottleneck which limits the number of VMs/containers/processes that can be consolidated [1].

Memory deduplication can reduce memory footprint by eliminating redundant pages. This is particularly true when similar OSes/applications/data are used across different VMs. For instance, Chang *et al.* [2] observed as much as 86% redundant pages in real-world applications. Essentially, memory deduplication detects those redundant pages, and merges them by enabling transparent page sharing. It is important to mention that deduplication has penalty besides benefit. These shared pages are managed in a copy-on-write (COW) fashion, that is, when a write request happens to one of the transparently shared pages, this specific page can not be shared any more. A new copy of this page will be generated in the memory so that the write request is applied there, which is called page *COW-broken*.

The responsiveness of the deduplication process to newly generated pages is critical. For a production system, the memory is always dynamic, where pages come and go. As demonstrated by our typical cloud computing workload experiment (Section 8), if an approach cannot catch up with the generation speed of memory redundancy, memory pages would be swapped out to the disk, and the whole system is slowed down.

State-of-the-art Content Based Page Sharing (CBPS) approaches lack responsiveness as they equally scan every page to find redundancies. CBPS is a major deduplication method in Linux, Xen and VMware [3, 4, 5]. It is capable of full memory scan and it is easy to be integrated into main stream systems. For example, Linux's Kernel Same-page Merging (KSM) is a kernel feature that deduplicates pages for both virtualized and non-virtualized environments. In short, CBPS uses a scanner to calculate the hash value for every candidate page. If two pages share the same hash value, a byte-by-byte memory comparison is performed. If duplication confirmed, one page is merged to the other. It should be noted that: *NOT* all pages are created equal. Due to their applications' nature, some pages have little chance of being identical to others. These so called *sparse* pages should be tested in the last place. Some pages, although identical to others at the very beginning, can quickly become either COW-broken or freed. We refer to them as *COW-broken* pages and *short-lived* pages respectively. An ideal candidate page for deduplication should remain

static (*i.e.*, not COW-broken or freed) for a reasonable period of time. A deduplication approach should prioritize these *statically-duplicated* pages. Further, deduplication operations performed on different pages may have different degrees of performance impacts on applications. We should also minimize deduplication's penalty on running applications due to operations such as page table locks and recovery of COW-broken pages.

Our observation is that *pages within the same memory region present similar duplication patterns* (Section 3). Here a memory region refers to a continuous virtual memory region allocated by an application (*i.e.*, allocated by *malloc*, *brk*, *mmap* etc). In some regions, most pages are statically-duplicated. In other regions, most identical pages may quickly become COW-broken or freed. According to the dominant page pattern, we can label a region as one of the four types of *sparse*, *COW-broken*, *short-lived* and *statically-duplicated* (*i.e.*, with a high duplication ratio, long-lived and seldom-changed). Intuitively, if we can prioritize pages in statically-duplicated regions for testing redundancy, the deduplication speed could be significantly accelerated. However, the challenge is how to distill these regions without testing every page at the first place?

Our key insight is that for each memory region, we can estimate its duplication ratio by sampling only a portion of all pages, at the same time monitor its degree of dynamics and lifetime. We can then distinguish sparse, short-lived and frequently COW-broken regions from statically-duplicated regions. To this end, we propose a new deduplication system *Ultra KSM* (UKSM). Build on top of KSM, UKSM improves traditional CBPS designs by prioritizing statically-duplicated regions over other regions to accelerate the deduplication process and minimize application penalty (Section 4).

UKSM introduces a hierarchy of sampling levels, each of which maintains a linked-list of memory regions. Each time an application *mmap*-s a new memory region, this region is immediately inserted into the list of the bottom level, which has the lowest scanning speed hence the lowest sampling density. A single thread iterates over levels to sample and deduplicate pages in each level. After each round of sampling, the duplication ratio and COW ratio of each region are compared with a set of threshold values. Once a memory region is identified as a potential statically-duplicated region, it is promoted from the current level to the next higher level which has a higher scanning speed hence a higher sampling density. This hierarchical architecture ensures system responsiveness by investing more CPU resources in regions in higher levels (Section 5).

To minimize the computational cost, we further develop a new partial-page hashing scheme called Adaptive Partial Hashing (APH). Let *page hash strength* denotes the number of bytes hashed in each sampling page. We define *profit* as the time saved compared to the strongest page hash strength and *penalty* as the wasted time of futile memory comparison due to hash collision. APH adaptively selects a global page hash strength to maximize the overall benefit which is *profit* subtracting *penalty*. Our novel progressive hash algorithm can support hash strength adaptation with incremental cost. Note that APH can improve other deduplication approaches as well since they are mostly hash-based (Section 6).

UKSM is implemented in both Linux kernel and Xen. The approach can detect and merge duplicated memory pages in real-time without intruding other parts of a system (*e.g.*, I/O, file system, etc). Experiments demonstrate that, with the same amount of CPU cycles in the same time envelop, UKSM can achieve up to 12.6×/5× more memory saving than CBPS approaches (*e.g.*, KSM) on static/dynamic workloads, respectively. UKSM also significantly outperforms XLH (*i.e.*, 50% more memory saving with the same amount of CPU consumption), a state-of-the-art I/O hint based approach. UKSM introduces negligible CPU consumption (around 0.2% of one core) when the host has no more page to be deduplicated, at the same time can respond to emerging duplicated pages rapidly (Sections 7 and 8).

UKSM is an open source project and benefits a wide range of applications [6]. Its patches for Linux kernel were first released in 2012 and have been kept synchronized with upstream kernel releases ever since. UKSM has been downloaded for over 30,000 times (at our site [6] alone, not including those re-distributed by other developers) at the time of the paper's publication. Besides the default versions, UKSM was also ported to kernels for desktop/server Linux systems [7, 8, 9, 10, 11] and Android systems [12, 13, 14, 15, 16] by third-party developers.

2 Related Work

Content-based Page Sharing (CBPS) VMWare ESX server [5] is the pioneer of content based page sharing approaches, where memory pages are scanned one-by-one. To control realtime CPU overhead, pages are randomly selected at a fixed scanning speed. A hash function is applied to each page for checking the similarity among pages. Pages that hash to the same value are byte-by-byte fully compared before they can be shared through copy-on-write. IBM Active Memory Deduplication [17] uses a similar approach for hypervisors in Power systems. CBPS for Xen was proposed by Kloster et al. [4] and later extended by XDE [18]. They detect page similarity by SuperFastHashing 64-byte blocks at two fixed locations in each page [19].

Linux Kernel Same-page Merging (KSM) [20] allows applications (including KVM [21]) to share identical

memory pages via full page comparison. KSM works well for deduplicating fairly static pages. Singleton [22] extends KSM to consider host disk cache in a VM environment and improves the scanner from full-page comparison to SuperFastHash-based hash comparison. Red Hat Enterprise Linux uses a dedicated user space daemon named *ksmtuned* [23] to adjust KSM scanning speed under certain circumstances. For example, it increases the scanning speed when memory usage exceeds some threshold and the system is starting virtual machines. It is a very limited approach that simply adjusts scanning speed according to coarse grained system information which may not always imply page duplication. KSM would waste CPU resources if this kind of implication fails. It is hard for *ksmtuned* to achieve maximum saving across different workload patterns [3], although it does improve performance if optimized case by case.

Instead of treating every page equally, UKSM prioritizes different memory regions to accelerate the deduplication process. APH shares partial page hashing ideas [18] but can adapt the global page hash strength according to page similarity in the whole system.

Catalyst [24] offloads page hashing computation to GPU to improve deduplication performance. The need of special hardware support increases deployment complexity. SmartMD [25] uses page access information monitored by lightweight schemes to improve the efficiency of large page (e.g. 2M-pages) deduplication. This work is orthogonal to UKSM since we address the more general problem of page deduplication.

I/O hint based page sharing KSM++ [26] proposed a deduplication scanner based on I/O hints. XLH [27] utilizes cross layer I/O hints in the host's virtual file system to find sharing opportunities earlier without raising the deduplication overhead. A generalized memory deduplication was proposed in [28] that leverages the free memory pool information in guest VMs. It treats free memory pages as duplicates of an all-zero page to improve the efficiency of deduplication. I/O-hinted approaches cannot detect dynamically created duplicated pages (*e.g.*, anonymous pages created by applications in Docker containers).

CMD [29] is a classification-based deduplication approach. Pages are classified according to their access characteristics. Comparison trees introduced in KSM are subsequently divided into multiple trees dedicated to each class. Thus, page comparisons are performed only in the same class which reduces futile comparison among different classes. However, the above strategies require dedicated *hardware* monitors to capture system I/O or page access characteristics, which incurs significant deployment complexity.

In this paper, we focus on improving CBPS because of its capability of full memory scan and easy integration to all existing systems, neither of which is the case for I/O hint based page sharing option.

Storage deduplication is different Deduplication projects in disk storage systems [30, 31, 32, 33, 34] are important related works. However, there exist two significant differences.

First, UKSM faces the challenge of responsiveness which is not the case for disk storage deduplication projects. For instance, when a large volume of duplicated pages are generated, memory deduplication system needs to quickly identify and remove these duplicates before they exhaust available physical memory and cause memory swap out.

Second, since memory is dynamically updated while disk storage is relatively static, memory deduplication pays attention to more characteristics than just a duplication ratio that is the centerpiece for disk deduplication As reflected in this work, UKSM also considers COW ratio and lifetime characteristics of memory regions.

3 Observations

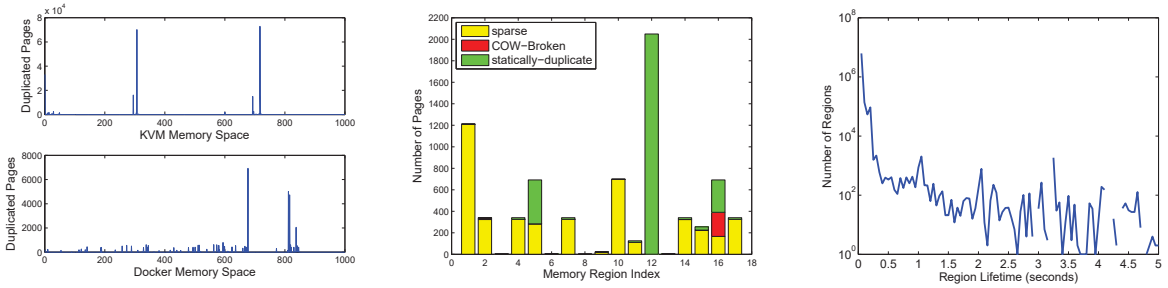
This section discusses two key observations that motivate the design of UKSM.

Observation # 1: Most pages within the same region present similar duplication patterns.

All heap memory allocation operations end up relying on *mmap* to claim memory spaces. For each call, mmap allocates a memory region that encompasses one or multiple virtual pages with continuous virtual addresses. Our intuition is that pages in the same memory region might exhibit same characteristics for deduplication. For instance, KVM exploits mmap to allocate memory space for each guest VM's OS. If two memory regions from different VMs store the same disk content for a long term, pages in them are friendly to deduplication. As a comparison, if a region of a network program serves as its busy network socket buffer, pages in it may not worth to be deduplicated even if many of them are identical. It will lead to frequent COW-broken operations.

Settings We use KVM and Docker as workloads for analysis of duplicated, COW-broken and short-lived pages. For the container workload, we make a Docker image from a Ubuntu based system with Apache web server and MySQL database serving a WordPress website. We then start three Docker containers from this image. For the KVM workload, we start three KVM virtual machines all installed with Ubuntu 16.04.

Results Page duplications demonstrate strong locality with respect to application memory regions. For both KVM and Docker, we evenly divide their virtual memory spaces (each contains many small memory regions) to 1,000 buckets. The number of duplicated pages in each bucket is presented in Figure 1(a). It is clear that most duplicated pages concentrate on a portion of memory



(a) Number of duplicated pages (b) A snapshot of Docker memory fragment (c) Distribution of short-lived regions

Figure 1: Memory regions in VM and container environments.

regions. We randomly demonstrate a bucket, which contains 18 memory regions from different processes, of one Docker's memory space. Note that some regions are so small in size (*e.g.*, regions 3, 6, 8) that they are almost invisible in the bar illustration. As shown in Figure 1(b), most pages in the same region share similarity in redundancy. Some regions are sparse and contain little duplications (*e.g.*, regions 1 and 2). Some are highly statically-duplicated (*e.g.*, region 12). Regions 5 and 16 are more complicated, where different kinds of pages coexist in the same region. Figure 1(c) presents the region distribution, whose lifetime is less than 5 seconds, of the Docker workload. We can see that a huge number of regions are short-lived.

Observation # 2: Partial page hashing need to be adaptive

XDE [18] has demonstrated that partial page hashing can improve scan performance. We further observe that hashing a fixed number of bytes for each page, albeit partially, can limit the benefits of partial page hashing because different scenarios may have drastically different workloads.

For example, image display application renders a dotted image with the same color background. In this case, we need to hash more bytes in order to differentiate highly similar (but not identical) pages to avoid time-consuming byte-by-byte page comparison. While for other workloads, pages may be quite different to each other. An crypto application tends to hold memory regions with encrypted data as content. Hashing one or two bytes is already enough to identify the difference between pages.

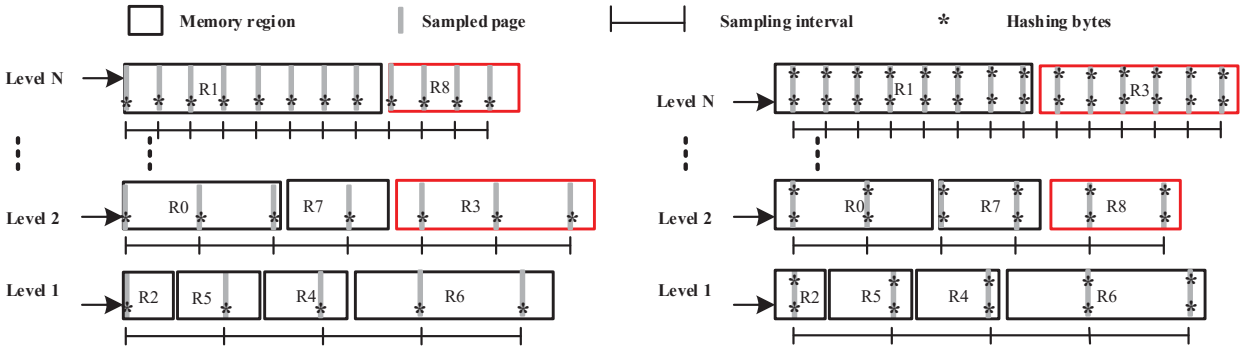
Real world systems may be filled with all kinds of workloads. The workload might even evolve with time. For example, a container may hold a remote desktop, the user may close a paint application and open a https browser.

4 Overview

UKSM consists of two unique components, that is, memory region hierarchical distilling (in Section 5) and adaptive partial page hashing (in Section 6). Figure 2 demonstrates how these two components identify duplicated pages with minimal scanning overhead through a simple example. There are nine memory regions (R_0 - R_8). Figure 2(a) and 2(b) demonstrate two whole memory sampling rounds (*i.e.*, round 1 and 10).

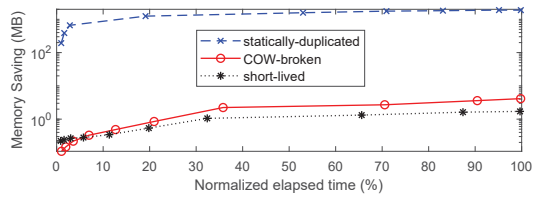
Memory region hierarchical distilling manages memory regions by levels. There are N levels as shown in Figure 2, and every region falls into one of the N levels. Level N is the highest level and level 1 is the lowest level. A higher level has a higher scanning speed hence a higher sampling density. Let each gray bar represents a sampled page. Demonstrated in the figures, the sampling interval decreases as the level increases. Each newly allocated memory region is first inserted into level 1 of the hierarchy. Newly added pages may not be statically-deduplicated. Computing power should not be invested on these regions before they are proved worthwhile. That is why we put them into the lowest level of the hierarchy. During each sampling round, every memory region is sampled and filtered with a group of distilling parameters to decide whether it should be "promoted" to the next higher level or "demoted" to the next lower level . If all duplicated pages in a region are merged at some scan level, it goes back to level 1. If a region is unmapped it will be tagged and removed from the linked level later by the scanner. For example, in round 1 of Figure 2(a), regions R_3 and R_8 reside in level 2 and N respectively. When the scan thread proceeds to round 10, regions R_3 has been promoted to level N while R_8 has been demoted to level 2. Further elaboration of this technique are discussed in Section 5.

To further minimize the computational overhead, UKSM introduces the Adaptive Partial page Hashing (APH) approach. The key idea is that we will adjust a global hash strength after each global page sampling round in order to achieve a more cost-effective scanning. For example, in Figure 2, each star represents a hashing byte in each page. In round 1 of Figure 2(a), the hash strength is one byte per page. Based on the feedback from preceding sampling rounds, in round 10 of Figure 2(b), we increase the hash strength to two bytes per

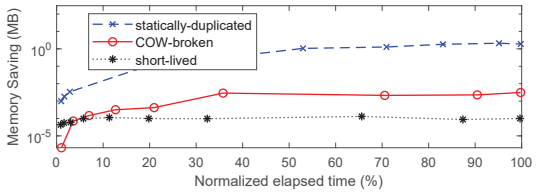


(a) round 1 with a smaller hash strength **(b) round 10 with a larger hash strength**

Figure 2: Memory region hierarchical distilling in two sampling rounds with different hash strength.



(a) Memory saving



(b) Mix with sparse background

Figure 3: Results with different workloads (Y-axis in each figure is exponential).

page. This design significantly improves per-page scan speed so that the overall system can respond to emerging duplicated pages rapidly and at the same time remains very low CPU consumption when no good candidates for deduplication exist. Details about hashing strength and feedback controls are discussed in Section 6.

5 Hierarchical Region Distilling

This section introduces design details of memory region hierarchical distilling. By analyzing the deduplication gain and lose of each kind of memory regions, we discuss distilling and scan principals (Section 5.1). Then whether a memory region is "promoted" or "demoted" is dominated by a set of threshold values (Section 5.2). At last, we present the hierarchical sampling procedure (Section 5.3).

5.1 Memory region characterization

Intuitively, a memory region should contain many statically-duplicated pages in order to be deduplication-friendly. In contrast, the unfriendly one could be frequently COW-broken, short-lived, or contain little identical content.

We uses three metrics to study the deduplication effects over a specific kind of regions. The first two metrics measure the gain and lose associated with deduplication, which are *memory saving* and *CPU consumption*, respectively. The third metric is *performance impact*, which reflects the slowdown ratio a deduplication method brought to a running application. Note this parameter is more comprehensive than the CPU consumption metric because a slowdown can result from cache/memory contention even if the deduplication worker (*e.g.*, *ksmd* – the kernel thread worker of KSM) executes on a dedicated CPU core. Ideally, we would like to maximize the first metric and minimize the other two.

This section utilizes four application configurations to emulate different workloads. Particularly, we use a memory footprint of about 2 GB for each of the *statically-duplicated*, *COW-broken* and *sparse* workloads. The compiling of the Linux kernel serves as an benchmark for *short-lived* workload which consumes about 30 MB memory space (only the anonymous pages, not including the file cache). We take KSM as the representative of existing CBPS approaches to demonstrate the complexity of balancing among three metrics. We record the maximum time needed for deduplicating all eligible pages of the *statically-duplicated* workload when using 100% capacity of a single CPU core. Then we normalize the time in x-axes of Figure 3, to demonstrate the progress of deduplicating each workload.

Memory saving vs. CPU consumption. Figure 3(a) shows how average memory saving progresses with time for *statically-duplicated*, *COW-broken* and *short-lived* workloads. Firstly, more CPU consumption does bring more memory saving, until the last duplicated page is merged. The deduplication speed, which is the slope of each line, drops rapidly as time goes on for *statically-duplicated* workload. Secondly, the memory saving of *statically-duplicated* workload is two orders of magnitude higher than that of *COW-broken* and *short-lived*

Table 1: Distilling and sampling parameters

V_{cow}	Only regions whose COW-broken ratios are lower than this threshold can be promoted.
V_{dup}	Only regions whose duplication ratios are larger than this threshold can be promoted.
V_{life}	Only regions lived longer than this threshold can be effectively scanned.
T_s	The sleep time in each sleep-scan cycle of the scan thread.
t_l	The expected time of sampling round for level l (in seconds).
t	The expected time of a global sampling round (in seconds).
p	The invested CPU percentage).
s	The estimated CPU cost of sampling one page.

workloads, which is consistent with our expectation. So we conclude that, for *statically-duplicated* workloads, invested computation is effective at the beginning. After all candidate pages are merged, further scan needs to be slowed down. For dynamic workloads (*COW-broken* and *short-lived*), higher CPU consumption is required to save the same amount of memory. Users may need to decide if the trade-off is worthwhile.

In Figure 3(b), we let each workload mixed with a *sparse* workload. The amount of memory saving decreases significantly. It is clear that scan of *sparse* regions in a system should be delayed, if not totally avoided, as much as possible.

Performance impact. We further study the performance impact to CPU intensive workloads brought by this hash-based KSM. One workload is a full SPEC-CPU2006 benchmark, and the other is the COW-broken Linux compiling workload mentioned above. If the scanning thread works at full speed with enough CPU resources (*i.e.*, scanning thread and workload threads each has its dedicated CPU core), the performance impact to *COW-broken* workload is 29.7%, and the impacts to other workloads range from 1.5% to 22.9%.

In-depth profiling shows that: 1) even with abundant CPU cores to separate workloads and the scanner, intensive scanning of CPU bound workloads makes the scanning thread contending more for memory management locking (*i.e.*, VMA locks, page table locks, etc), which introduces higher overhead for these workloads; 2) deduplication on frequently COW-broken pages may not bring much memory saving, but will bring many COW-broken page faults on merged pages, thus deteriorate performance.

5.2 Candidate region identification

The key characteristics (*i.e.*, *COW ratio*, *duplication ratio*, and *average page lifetime*) that indicate the duplication qualities of each memory region should be obtained first. This section introduces corresponding quantitative threshold values that can decide whether we "promote" or "demote" a memory region. Table 1 details these three thresholds, i.e., V_{dup}, V_{cow} and V_{life}. In particular, a regions with duplication ratio above V_{dup}, COW-broken ratio below V_{cow} and life longer than V_{life} can be identified as a good candidate for *statically-duplicated*. To control CPU overhead, we make the scanner work in a sleep-scan cyclic pattern with sleep time T_s. This parameter is related to the life time threshold.

For a memory region, the first parameter duplication ratio is estimated by dividing the duplicated page counter by the number of the pages sampled in this round. To compute its COW ratio, we need to obtain the number of COW-broken page faults on merged pages during each sampling round. This information can be easily obtained by hooking the page fault handler function. The last parameter lifetime is decided by the sleep time T_s and the sampling round time t (sum of t_l for each level in Table 1). Only those regions which live across this sleep-scan cycle time may get sampled.

How to choose threshold values? Threshold values of *duplication ratio, COW-broken threshold, lifetime* are critical parameters for UKSM. We design UKSM as a general system and it targets a wide range of scenarios. The default settings of *10%, 50%, 100ms* are obtained empirically and are shown to work well for a wide range of systems. The global sampling round time can be configured in the range of 2 - 20s with further details explained in Section 7. However, we also leave these parameters configurable for expert users who can tune UKSM to meet their application-specific needs. As far as we know, many follow-up production systems extend various configurations of UKSM to meet their particular needs [7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. The auto-tuning of these parameters could be a future work.

Why use the same set of threshold values across all levels? The higher the level is, the higher the page scanning frequency is. There is a larger chance of *false positive* in a lower level, where a low duplication ratio region may accidently get promoted. Hence, even with the same set of threshold values at all levels, we can successfully demote false positive regions and promote real positive regions. Further, UKSM performs merge together with the scanning. Every time a new duplicated page is found, besides being counted in the duplication ratio calculation, the page is merged directly. Statistically, even with the same threshold, regions in a higher level should have a larger duplication ratio than those in a lower level.

```
For(;;){
    A global sampling round {
        Sleep Ts;
        promote/demote regions;
        For(l = 1, l<= N, l++) {
            scan level l with budget time tl;
        }
    }
}
```

Figure 4: Workflow of UKSM hierarchical sampling.

5.3 Hierarchical sampling procedure

Now that UKSM has the information of the key features of each memory region and the promotion criteria, this section discusses our hierarchical sampling approach which manages memory regions by levels and each region only belongs to one level. For instance, Figure 2(a) and 2(b) manage 9 regions by N levels. Figure 4 shows the workflow.

Scan a level When sample a specific level, all memory regions are grouped to be one flat linear space. The memory scanner starts at page offset of zero in this linear space and picks sample points by the length of *interval*. Note that a higher level possesses a smaller interval. If a sample point falls in a region, one page will be selected from this region. Particularly, we introduce a region specific offset permutation scheme to avoid sampling the same page repeatedly. For instance, although R_2 from level 1 is sampled in both rounds of Figure 2(a) and Figure 2(b), different pages are picked.

Once the page is selected, our scanner will get the page's hash value according to current hash strength (*i.e.*, bytes hashed in each page), and looks it up in two red-black trees trying to find a collision. One of the red-black trees (Tr_s) tracks the "merged" pages whereas the other one (Tr_{us}) records the "unmerged" ones. If the sampled page has an identical page in Tr_s, we increase the region's counter by one. If the sampled page is found to be identical to one of the pages in Tr_{us}, we move the page to Tr_s and increase the counters of both regions. Eventually, we update the page table and release redundant pages accordingly. UKSM keeps "merged" and "unmerged" page hashes in separate trees because merged pages should be managed in a read-only tree. Write to any node in this tree causes a COW operation.

This scan continues until the sample point reaches the boundary of the linear space. We call it a sampling round in this level. The scanner then proceeds to the next level.

A global sampling round A global round is finished after the level N sampling. then the scanner restarts from level 1. After each global round, the scanner estimates each region's duplication and COW-broken ratios. It is easy to see that with sufficient lifetime, every page of a memory region will be scanned. If a region is *unmapped* before every page is scanned, it will be removed from that level.

Sampling time control For each level, we can easily get the number of pages in one level as L_l. With invested CPU computation p and the estimated time of sampling one page s, we get the average page processing speed as p/s. Assuming the expected sampling round time for this level is t_l, the number of sample points in one round is $n = t \cdot p/s$. The sampling interval in each level is determined by $L_l/n = L_l \cdot s/(t_l \cdot p)$. The sleep time is T_s, so the active time of each sleep-active cycle is $T_s \cdot p/(1-p)$. Then we can get the number of pages to scan during each active cycle as $T_s \cdot p/(s \cdot (1-p))$.

In summary, users can configure two parameters which are p and t, as the invested CPU computation time and global sampling round time, respectively. According to our empirical study, p and t can be configured in the ranges of 0.2% - 95% and 2 - 20s, respectively.

6 Adaptive Partial Hashing

We propose a new page hashing function to reduce per-page scan and deduplication cost. The key idea is to *partially hash* a page. if the hash value is already sufficient to distinguish different pages, we do not need to hash a full page. Generally, the new hash function should have the following features:

- The hash strength (i.e. bytes hashed in a page) should be adjustable. If the memory pages are "quite different", a weaker strength is used. Otherwise, a stronger strength is applied.
- With the strongest strength, the hashing function should have a comparable speed and collision rate to SuperFastHash for arbitrary workloads.
- With weak strength values, the hash function should be significantly faster than SuperFastHash.
- The hash function should be bidirectional progressive with cost proportional to the delta of strength, hence the page hash values with an updated strength can be incrementally computed from previous values.

6.1 Hash strength adaptation

A weak hash strength may increase the possibility of false positive, which can result in additional overhead on *memcmp*. For each sampling round, we quantify the *profit* for using some hash strength by the time saved compared to that of using the strongest strength. We quantify its *penalty* by the additional time of *memcmp* due to collision. The calculation for both *profit* and *penalty* is instrumented in the scan functions. The aim of hash strength adaptation is to maximize the overall benefit of *profit-penalty*. In what follows, we explain how our adaptive algorithm finds the optimal strength for the hash function.

When the system starts up, the hash strength is initialized with half of the strongest strength. After the

```
#define STREN_FULL (4096/sizeof(u32))
u32 shiftr, shiftl;
u32 random_offsets[STREN_FULL];
u32 random_sample_hash(u32 hash_init,
    void *page_addr, u32 strength) {
    u32 hash = hash_init;
    u32 i, pos, loop = strength;
    u32 *key = (u32*)page_addr;

    if (strength > STREN_FULL)
        loop = STREN_FULL;
    for (i = 0; i < loop; i++) {
        pos = random_offsets[i];
        hash += key[pos];
        hash += (hash << shiftl);
        hash ^= (hash >> shiftr);
    }
    return hash;
}
```

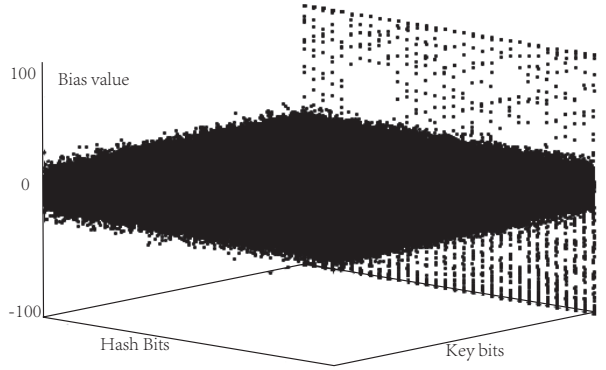
Figure 5: Progressive hash procedure.

first sampling round, the system enters a "probing" state trying to search for a strength that leads to a better overall benefit and finally stays in a dynamically "stable" state. The searching in the probing state simulates the TCP slow start process. The system firstly decreases the hash strength by a size variable named *delta* (initialized with 1) and checks if this change results in larger benefit. If it does, the system goes on trying until the benefit begins to decrease. During this process, *delta* will be doubled each time till the max value 32. Then the system records the maximum benefit point achieved and reset *delta* to 1. Similarly, the system will search in the other direction when increasing the hash strength. Once the system reaches an optimal point, it enters a stable state.

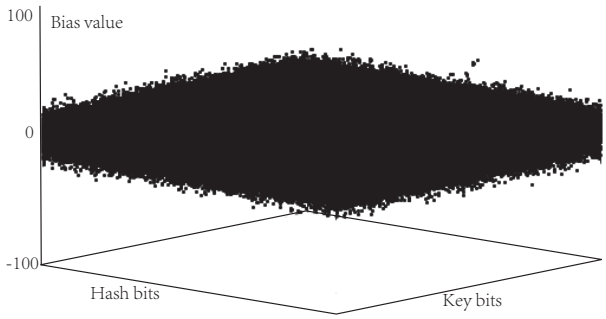
The state changes from *stable* to *probing* is triggered by one of the following conditions: 1) The benefit of the last sampling round deviates more than 50% from the benefit value when the system enters a stable state; 2) There is no *memcmp* caused by hash collision in the last two sampling rounds; 3) Every 1000 sampling rounds have been passed.

6.2 Progressive hash algorithm

We decide to use random sampling to fulfill the feature of dynamically adjustable strength. A universal random permutation of all the 32-bit-aligned offsets in a page is computed when the deduplication system is initialized. This is important because randomization is necessary in cases where some pages have specific patterns (*e.g.*, leading zeros). When a page is hashed with strength *I*, only the first *I* 32-bit data units are read and calculated from the page with the corresponding offsets in the permutation. In order to limit the execution time for the strongest strength, we derive the hash algorithm based on Jenkins's "one-at-a-time hash" [35] which is also the ancestor of SuperFastHash. The algorithm framework is shown in Figure 5. In the code, *random_offsets* is the buffer holding the random permutation of offsets; *shiftl* and *shiftr* are the two values we need to parameterize to further satisfy other features required for collision rate and incremental/decremental calculation; *STREN_FULL* is the strength for hashing a full 4 KB page content.



(a) SuperFastHash avalanche



(b) Random_sample_hash avalanche

Figure 6: The avalanche effects over a 4KB page.

Achieve low collision To ensure a low collision rate, we study the avalanche effect [36, 37, 38] of the hash function in our algorithm when hashing a full page with different *shiftl* and *shiftr* values. Avalanche is a desirable property of hash algorithms to achieve low collision rate wherein if the input is changed slightly the output can change significantly in pseudo-random manner. We evaluate the avalanche effect with an initially zeroed two dimensional matrix which we call avalanche *bias_matrix*. Given a randomly generated key of page size, we flip the *i*-th bit. If this operation leads to the flipping of the *j*-th bit of the hash value, we increase point (i, j) in *bias_matrix* by one, and if the *j*-th bit of hash value is not affected, we decrease *bias_matrix(i, j)* by one. This process is repeated for multiple times, then we calculate the average value of $bias\_matrix(i, j)$ for all $i \in [0, 32767], j \in [0, 31]$. Ideally, one bit changes in the key will affect the output of hash value with 50% probability. Therefore, the corresponding *bias_matrix* entry should approximate 0 on average.

Figure 6(a) is the 3D visualization for such an avalanche *bias_matrix* of SuperFastHash. We can see that most of the points are closed to the $bias\_value = 0$

```
u32 reverse_addeq_shiftl(u32 n) {
    u32 ret = n, turn = 1;
    n <<= shiftl;
    while (n != 0) {
        if (turn)
                ret -= n;
        else
                ret += n;
        turn = !turn;
        n <<= shiftl;
    }
    return ret;
}

u32 reverse_xoreq_shiftr(u32 n) {
    u32 ret = n;
    n >>= shiftr;
    while (n != 0) {
        ret ^= n;
        n >>= shiftr;
    }
    return ret;
}
```

Figure 7: Reverse functions for progressive hash.

plane except for the last several key bytes. We therefore evaluate the avalanche effect of the hash algorithm by the number of the "bad points" whose deviation from the $bias\_value = 0$ plane exceeds a threshold (for our case, we take 50). We conduct an exhaustive search of all possible (*shiftl*, *shiftr*) value pairs and generate a priority list of them (omitted due to space limitation). The avalanche behavior of our hash algorithm with maximum strength is illustrated in Figure 6(b). It is better than that of SuperFastHash as illustrated.

Achieve progressive hashing Assume the recorded hash value of a page is achieved at strength S_1 but the current strength is S_2, the updated hash value can be achieved with additional computation using the recorded result for S_1. If $S_2 > S_1$, this can be done by filling the *hash_init* parameter (in Figure 5) with the hash value at strength S_1. If $S_2 < S_1$, the hash calculation must be reversed. The "+=" operation can be reversed with "-=". The "+=" and "^=" operations combined with "<<" and ">>" can be reversed by the code in Figure 7.

Small values of *shiftl* and *shiftr* will increase the cost of reverse operations. We choose the pair of (19, 16) from the priority list for (*shiftl*, *shiftr*) which brings very good avalanche effect and at the same time makes the cost of the reverse operation comparable to that of progressive hash operation. We compare the speed of *random_sample_hash* with maximum strength and SuperFastHash and find that our algorithm is only about 2% slower than SuperFastHash. The final avalanche effect result of our hash algorithm with maximum strength is slightly better (fewer "bad points" as we state above) than that of SuperFastHash.

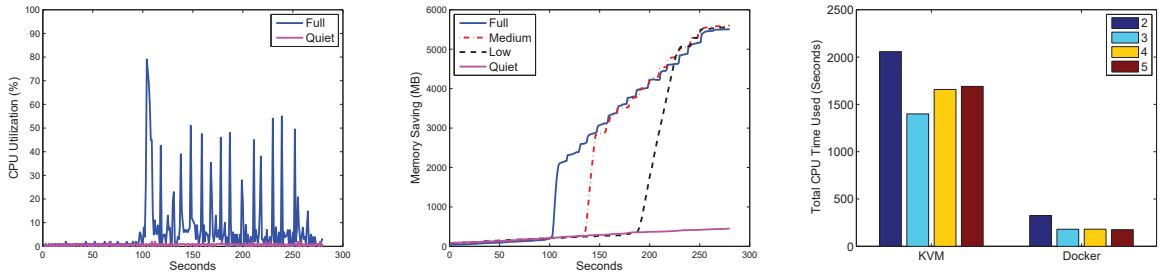
Why use a global hash strength design instead per-region or per-app hash strength? Here hash strength denotes how many 32-bit words are hashed to generate a fixed length hash value. If we use different numbers of 32-bit words for two different pages, these two pages cannot be compared directly. That is why we use a global hash strength, so that every pair of pages can compare their hash values directly.

Why develop APH based on SuperFastHash? There are some newly developed fast hash algorithms, such as Spooky [39], xxHash [40], and Murmur [41], which are much faster than SuperFastHash. Whether an algorithm can derive an adaptive version depends on its design details. Using one of those hashes in our adaptive hash framework could be an interesting future work.

7 Implementation and Configuration

UKSM is implemented in both Linux kernel and Xen, each with more than 6,000 lines of C code. In Linux, UKSM hooks the Linux kernel memory management subsystem for monitoring the creation and termination of application memory regions. The kernel page fault routine is also hooked to log COW-broken events in each region. UKSM scanner is created as a kernel thread *uksmd*. In Xen, UKSM scanner is implemented as a softirq service routine of the Xen hypervisor. The Xen memory management subsystem is also hooked in the same way as in Linux kernel.

To facilitate drop-in utilization of UKSM, we borrow the idea of "CPU governors" with which the Linux kernel simplifies the configuration for Intel CPU frequency [42]. We define several default parameter sets named as "governors" to represent "how aggressive" the scanner should be. These "governors" are *Full*, *Medium*, *Low*, and *Quiet*. With the *Full* governor, it can use up to 95% CPU and finishes one global sampling round in 2 seconds. From *Full* to *Low*, each governor doubles the global sampling round time and reduces the top CPU usage by half. The *Quiet* governor is designed to be used in battery powered systems where workloads are static most of the time (*e.g.*, Android). It has a top CPU consumption of 1% and a global sampling round time up to 20 seconds. We use the workload of *booting 25 VMs* in Section 8 to depict the performance metrics of UKSM under different governors. The results are shown in Figure 8(a) (plotting only *Full* and *Quiet* for clarity) and Figure 8(b). We can see that with the *Low* governor, UKSM can already catch up with the booting process (about 260 seconds). The main difference is how fast a governor can catch up. We also observe that the CPU cycles consumed by the governors are proportional to the number of pages they deduplicate. It is consistent with our design purpose. Since the *Full* governor is more responsive, we choose *Full* as the default governor and use it for all later evaluations unless specified otherwise.



(a) CPU consumption for governors (b) Memory saving for governors (c) Performance with different level numbers

Figure 8: *Performance metrics under different configurations*

For the scan levels, the bottom level serves as a baseline sampling with CPU consumption as low as possible. We believe 0.2% should be acceptable for general systems. The CPU consumption of the top level is given by the "governor". The CPU consumption for each intermediate level is halved. The sampling round time for each level is evenly derived from global sampling round time. The number of scan levels determines how smooth the promoting process could be: more levels will make a memory region more carefully sampled before it is intensively scanned. On the other hand, less levels will make the system response more quickly to emerging duplication but may suffer false positives caused by sampling singularity (*i.e.*, a region is falsely identified as "good" after one scanning round). We tested the configuration from 2 levels to 5 levels with real world benchmarks used in Section 8. As shown in Figure 8(c), for larger regions of the KVM workload, sampling singularities are less likely to happen. So 3-level-sampling is the best choice. For smaller regions of the docker workload, 5 level is the best choice. We choose 4 levels as UKSM default.

Till the time of this paper being written, the feedbacks from different sources have demonstrated that while the system design stems from a server environment, its design and parameters are shown to work in a wide range of environments such as a mobile system (*e.g.* Android). Only very few people adjusted the individual parameters according to their specific requirement. We leave comprehensive parameters tuning under different types of workload as our future work.

8 Evaluation

We evaluate our UKSM implementation in comparison with the Linux kernel KSM. The operating system for our benchmarks is CentOS 7 with vanilla Linux kernel 4.4. The hardware setting is Intel(R) Core(TM) i7 CPU 920 with four 2.67GHz cores and with 12 GB RAM. The benchmarks include emulated workloads and real-world workloads. For fair comparison, the native Linux KSM scanner is upgraded to use SuperFastHash, which has a better performance. Our evaluation centers around five key questions:

How efficient is UKSM on different workloads? Using emulated workloads each focusing on a single type, we show that UKSM can be up to 12.6× more efficient than KSM on densely/sparsely 1:1 mixed workloads and can be up to 5× more efficient than KSM on frequently COW-broken workloads (Section 8.1).

How flexible is UKSM with customization? On the same set of workloads, we show that UKSM can filter different types of memory regions with different thresholds. With UKSM, users can customize their trade-offs, while previous approaches like KSM cannot (Sections 8.1.2 and 8.1.3).

What is the performance v.s. overhead tradeoff of UKSM on production workloads? By experiments on KVM VMs and Docker containers, we show that UKSM significantly outperforms *ksmtuned*-enhanced KSM in VM benchmarks. It can deduplicate the typical setup of Docker containers (which cannot be handled by KSM) with negligible CPU consumption (less than 1% of one core). The results also prove that our approach outperforms XLH even without I/O hints. The experiments on desktop servers with mixed workloads shows that UKSM can deduplicate newly generated pages in seconds (Section 8.2).

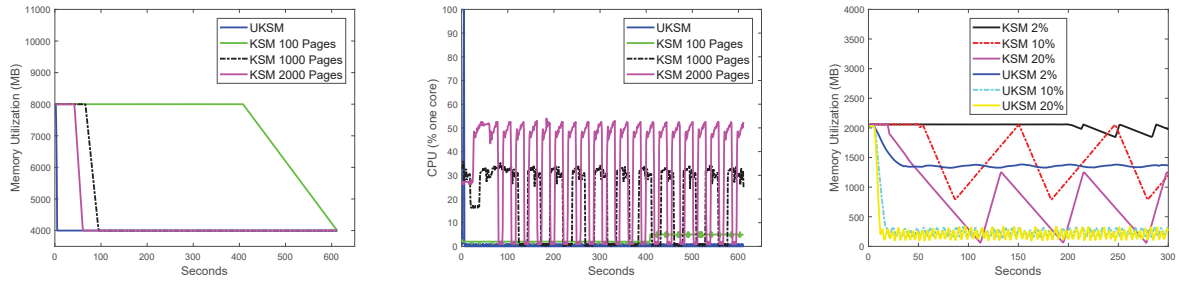
How does Adaptive Partial Hashing perform compared to non-adaptive algorithms? We analyze the effectiveness of APH on densely and sparsely duplicated pages. We find that APH alone can make the scanning speed of UKSM up to 7× that of KSM on typical cloud workloads (Section 8.3).

How large is the application penalty of UKSM? For native environments, UKSM's penalty is less than 3%. For virtualized environments, UKSM's penalty is less than 1.8% (Section 8.4).

8.1 Deduplication efficiency analysis

8.1.1 Statically mixed workload

We first evaluate the deduplication efficiency of UKSM and KSM on a static workload. This workload is composed of two programs. Each program creates 4 GB memory. One fills memory with identical page data. The other program fills memory with random data. After they complete filling pages, we start the UKSM/KSM



(a) Deduplication speed and memory saving (b) CPU consumption (c) Memory Utilization for COW-broken

Figure 9: *Benchmark performance comparison.*

daemon. Since the default scanning speed (100 pages each cycle) of KSM is very low, we also obtain the results of KSM when scanning 1000 and 2000 pages each cycle. As illustrated in Figure 9(a), it takes only 5 seconds for UKSM to merge all duplicated pages. While the deduplication time for KSM at 100, 1000, 2000 pages are 611, 95, 61 seconds respectively.

We then analyze the CPU usage of UKSM and KSM in the above benchmark. As shown in Figure 9(b), the CPU consumption pattern of UKSM is composed of very thin spikes and with average CPU of less than 1%. UKSM only reaches its peak CPU consumption (around 95%) at the 5th second. KSM constantly demonstrates very high CPU consumption especially at high scanning speed. This phenomenon reflects the fact that UKSM reacts rapidly to emerging duplicated pages and has a very low background CPU usage (recall that the pre-defined value for sampling level 1 is 0.2%) when all duplicated pages are already merged.

We then calculate the deduplication efficiency as *memory_saving* over *deduplication_CPU_consumption*, where *deduplication_CPU_consumption* is the sum of CPU consumption ratios of each second before all pages are deduplicated. From calculation, we find that UKSM is 8.3×, 12.6×, 11.5× more efficient than that of KSM at scan speed of 100, 1000, 2000 pages respectively.

8.1.2 COW-broken workload

We then demonstrate how UKSM improves over KSM on frequently COW-broken workloads. We emulate this case with a program that maps 2GB of memory and repeatedly *memset* one full page (with the same content) every 10 ms from the start to the end of the region. With the default setting of UKSM (COW-broken ratio threshold of 50%), it totally avoids intensively scanning this workload. However, UKSM can be configured to scan this workload if we disable the COW-broken filtering (note that KSM cannot be customized to avoid scanning this workload).

We make both KSM and UKSM consume about the same CPU power (2%, 10% and 20% of one core) and then compare the memory saving of them as shown in Figure 9(c). We can see that KSM saves only about 1/3 to 1/5 of the memory that could be saved by UKSM. Furthermore, the performance of UKSM is quite stable, in contrast, the memory saving of KSM suffers from large variations.

8.1.3 Short-lived workload

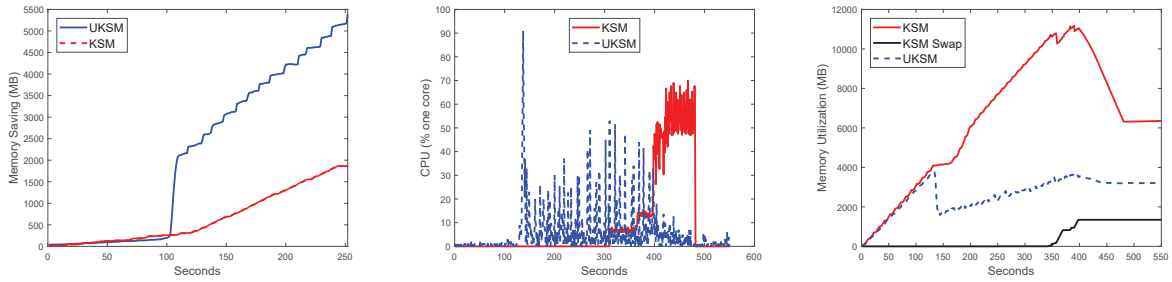
We emulate this case by a program that infinitely repeats a cycle of "mmap a region of 500MB pages of the same content, sleep for time of T, then unmap this region and sleep for another T". We observe that even with very aggressive settings of KSM (sleep time sets to 20ms, *pages_to_scan* sets to 2000, consuming about 50% CPU), it cannot merge a single page if T is less than 2 seconds.

Although it totally filters out this case with its default settings, it is possible to make UKSM sensitive to short-lived pages. After we assign its sleep time to 20ms, its max CPU consumption to 50% and its sampling round time of each level to be 50ms, 20ms, 10ms, and 5ms, respectively, UKSM can merge almost all the pages even if T is less than 200ms.

8.2 Real world benchmarks

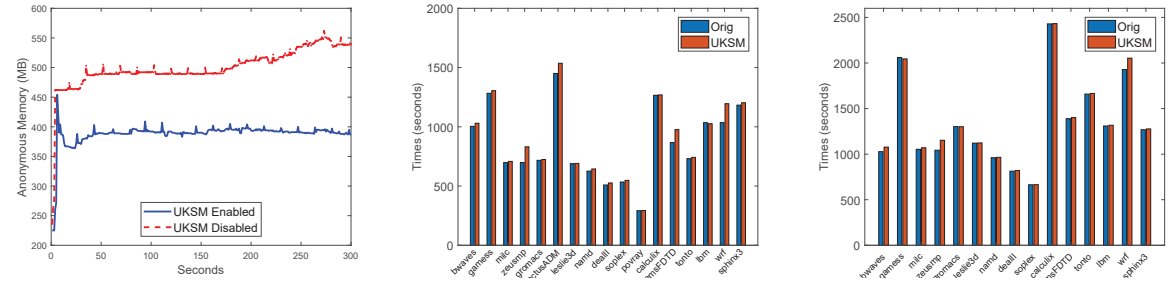
8.2.1 KVM virtual machines

Booting 25 VMs with abundant memory We uss the same benchmark used in XLH [27], As XLH is not an open-source implementation, we use an almost identical hardware/software platform settings. Thus we can compare our results with theirs. We booted 25 VMs (installed with Ubuntu server 16.04) each with a single VCPU and 512MB of memory in parallel, with starting time of 10 seconds apart. KSM is configured with the settings as that in XLH. UKSM uses the default settings. After about 260 seconds, all VMs are fully booted. Up to this point, UKSM has merged 5.3GB of memory, about 3× of what KSM has merged (Figure 10(a)). We need a warming up time to build rb-tree, offset and figure out duplications. That explains why in around 100 sec we have a jump. KSM and UKSM use about the same amount of CPU resources during this process. [27] reported that XLH can achieve only 2× the memory saving compared to KSM with same CPU resources. This implies that UKSM outperforms XLH significantly.



(a) 25 VMs Memory Saving (b) 40 VMs CPU (c) 40 VMs Memory Utilization

Figure 10: *Real workload performance comparison.*



(a) Deduplication for Docker containers. (b) CPU2006 fp, 1 CPU core (c) CPU2006 fp benchmarks in KVM

Figure 11: *(a) Deduplication for Docker containers; (b), (c) Performance impact of UKSM.*

Booting 40 VMs with memory overcommit We boot 40 VMs (with a single VCPU and 1GB of memory) in parallel, with starting time of 10 seconds apart. We record the total boot time, memory and swap usage until the system stabilized. In this benchmark, we compare UKSM (with default settings) with KSM settings adjusted by *ksmtuned*. As mentioned in related work, *ksmtuned* can dynamically increase and decrease the KSM scanning speed according to memory usage. Figure 10(b) illustrates the CPU utilization of UKSM and KSM during the deduplication process; from calculation, the aggregated CPU utilization of UKSM is about half of KSM. Figure 10(c) shows that, with KSM and *ksmtuned*, the system triggers about 1.3GB of swap and this slows down the boot process of the VMs significantly (from about 420 seconds to 550 seconds). UKSM uses only half of physical memory and requires no swap usage when the system stabilizes. The peak memory usage with UKSM is only 32% of that with KSM. With KSM, the last VM booted still waits for a long time before it can be logged in after *ksmtuned* shuts down KSM at 480 seconds. Moreover, when the swapping storm is triggered, the system suffers bad responsiveness, which would be a devastating user experience.

8.2.2 Docker containers

We start 3 Docker containers each running a WordPress website in LAMP environment (*i.e.*, Linux, MySQL, Apache and PHP). Each website contains a page with the same set of images and texts. Then we uses Firefox to emulate normal user connections by making it refreshing each website's page every second. Figure 11(a) shows the amount of the anonymously mapped memory in this process when UKSM is enabled or disabled. We find that the average memory used with UKSM enabled is only 61% of that when it is disabled. The CPU consumption during this process of *uksmd* is mostly less than 1% (one core) with few spikes to around 3%. At the time of this paper is written, no other open source implementation to our knowledge can deduplicate memory of containers, hence we do not compare with others in this benchmark.

It's worthy to note that Docker containers try hard to share the underlying files with aufs [43]. There may not be that much duplication in file cached pages. UKSM only handles anonymous pages for containers by now. File cache deduplication is left as our future work.

Table 2: UKSM introduced space saving and time consumption for mixed workload.

Application	Okular PDF	Firefox	FlashPlayer	GIMP
Space saving (MB)	415	27	63	39
Time (s)	2	5	4	1

8.2.3 Mixed workload on desktop server

In this Ubuntu desktop server, we run four applications, *i.e.*, Okular PDF reader, Firefox browser, FlashPlayer, and GIMP painter, simultaneously. The default memory is around 1,248 MB. We then perform operations with each software separately. At the same, we record the time and deduplication gain of UKSM, by monitoring the *htop* tool. With UKSM, it takes only 2/5/4/1 seconds to deduplicate 415/27/63/39 MB memory for

Okular/Firefox/FlashPlayer/GIMP in this mixed wowrkload environment. Consistent with our design principal, UKSM works well in real world systems.

8.3 Analysis of Adaptive Partial Hashing

In the first two experiments, we use two extreme scenarios, one system contains no duplication, and one full of duplicated pages. Thus, UKSM hierarchical region distilling has no effect at all, since all regions are at either the highest level, or the lowest level. In this case, the only difference between UKSM and KSM is that UKSM turns on APH.

8.3.1 Effectiveness of Adaptive Partial Hashing

Scanning speed in regions with low redundancy We first demonstrate how fast the optimized system can scan regions with low page redundancy. One extreme case is when the system is full of "quite different" pages, that is, no two pages have the identical 32-bit words at the same offset. That makes the hash strength drop to 1, or in other words, the hashing cost is about 1/1000 of the SuperFastHash hashing. The maximum scanning speed of UKSM in this case is about 5.9× higher than that of the hash-based KSM or 7.4× higher than the original KSM. On the other hand, if the pages are quite similar but not equal, the strength of hash function may rise to 1,024 words. In this worst case, the maximum scanning speed is about the same as the hash-based KSM, which is expected by the fact that the hash algorithm with the strongest strength is comparable to SuperFastHash.

Deduplication speed on highly redundant regions We then measure the maximum speed for merging identical pages when hash strength is 1. It's approximately 2.5× higher than that of the hash-based KSM or 7× higher than that of the original KSM. When increasing the strength of hash function to the maximum value, the merge speed becomes comparable to hash-based KSM or about 3× that of the original KSM.

8.3.2 Strength of hash function

The actual scan/deduplication speed on real workloads depends on the memory content pattern and our system adapts its page hash strength accordingly. A comprehensive testing on a variety of workloads has shown that the hash strength is usually within 100 words (recall that the highest value is 1,024). The scan speed of UKSM at this hash strength is about 6 to 7× higher than that of the original KSM or 2 to 5× higher than that of the hash-based KSM. We expect even smaller strength values in scientific computing or data processing environments. For example, the hash strength for all 12 SPEC-CPU2006 benchmarks ranges from 2 to 17, with the average of 7.5. These results validate the effectiveness of our hashing design.

8.4 Performance Impact

We evaluate the runtime overhead of UKSM using CPU intensive workloads of Standard Performance Evaluation Corporation (SPEC) CPU2006 benchmarks with *uksmd* under the *Full* governor. The experiments are firstly run on the host and then inside KVM virtual machines.

CPU2006 on host We evaluate UKSM in two scenarios, where 1) UKSM and CPU2006 are running within one CPU core; 2) The system has enough CPU resources so that UKSM will not compete with CPU2006. The result of CPU2006 float point benchmarks in the first scenario is shown in Figure 11(b). We can see that the average overhead is less than 3.0%. The average overhead in the second scenario is about 1.5%. We observe similar results with CPU2006 integer benchmarks (2.7%). We do not show the other figures due to space limitation.

CPU2006 in KVM The benchmarks run inside 2 VMs. Two CPU cores are enabled on the host for VMs. Each VM is assigned with 1 VCPU. As illustrated in Figure 11(c), the average overhead for CPU2006 float point is 1.8% (0.9% on CPU2006 integer group).

It is worth noting that these results are the worst case upper bound under the *Full* governor. We achieved almost the same memory saving for these benchmarks under the *Low* governor with negligible overhead.

9 Conclusion

We design a novel memory deduplication system called UKSM that (1) samples the whole memory with an enhanced hashing scheme to estimate the duplication ratio and dynamics of each memory region; and (2) performs different scanning policies for different regions to maximize computation efficiency and minimize application penalty. Experiments on both emulated workloads and real-world benchmarks show substantial improvements compared to standard Linux KSM and I/O hints based approach XLH.

Acknowledgment

The authors would like to thank our shepherd Prof. Hong Jiang and anonymous reviewers for their valuable comments. This work was supported in part by the National Science and Technology Major Project of China under Grant Number 2017ZX03001013-003, the Fundamental Research Funds for the Central Universities under Grant Number 0202-14380037, the National Natural Science Foundation of China under Grant Numbers 61772265, 61370028, 61602194, and 61321491, the National Science Foundation under Grant Numbers 1547428 and 1738965, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

[1] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.

[2] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 244–249, 2011.

[3] Shashank Rachamalla, Debahuti Mishra, and Parag Kulkarni. Share-o-meter: An empirical analysis of ksm based memory sharing in virtualized systems. In *IEEE International Conference on High Performance Computing (HiPC)*, 2013.

[4] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. Determining the use of interdomain shareable pages using kernel introspection. *Department of Computer Science, Aalborg University*, 2007.

[5] Carl A Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[6] Uksm official site. `http://kerneldedup.org/en/projects/uksm/`.

[7] Xanmod kernel 4.4.0 for ubuntu linux. `https://ubuntuforums.org/showthread.php?t=2307617`.

[8] Openmandriva lx 2014.2 kernel. `https://wiki.openmandriva.org/en/2014.2/New`.

[9] Achlinux port of linux-pf-lts 3.14.72-1. `https://aur.archlinux.org/packages/linux-pf-lts/`.

[10] Opensuse linux port of kernel-postfactum. `https://software.opensuse.org/package/kernel-postfactum`.

[11] Calculate linux 14.16. `http://distrowatch.com/?newsid=08899`.

[12] Shinto kernel secondreality v40a05. `https://www.precog.me/2015/02/27/release-shinto-kernel-secondreality-v40a05/3/`.

[13] Xda-developers, charm-kiss kernel 20140107. `http://forum.xda-developers.com/showthread.php?t=2487113`.

[14] Xda-developers, wonderchild kernel. `http://forum.xda-developers.com/showthread.php?t=2565299`.

[15] Xda-developers, decimalman's kernel playground. `http://forum.xda-developers.com/showthread.php?t=2226889`.

[16] Xda-developers, renderbroken's custom kernel. `http://forum.xda-developers.com/showthread.php?t=2724016`.

[17] Rodrigo Ceron, Rafael Folco, Breno Leitao, and Humberto Tsubamoto. Power systems memory deduplication. *IBM Redbooks*, 2012.

[18] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010.

[19] Paul Hsieh. The superfasthash function. `http://www.azillionmonkeys.com/qed/hash.html`.

[20] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Linux symposium*, pages 19–28, 2009.

[21] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Linux symposium*, volume 1, pages 225–230, 2007.

[22] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 15–26. ACM, 2012.

[23] Red hat enterprise linux virtualization administration guidechapter ksm. `https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Administration_Guide/chap-KSM.html`.

[24] Anshuj Garg, Debadatta Mishra, and Purushottam Kulkarni. Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments. In *the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 44–59. ACM, 2017.

[25] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John CS Lui. Smartmd: A high performance deduplication engine with mixed pages. In *USENIX Annual Technical Conference (ATC)*, pages 733–744, 2017.

[26] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Ksm++: Using i/o-based hints to make memory-deduplication scanners more efficient. In *ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE)*, 2012.

[27] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Xlh: More effective memory deduplication scanners through cross-layer hints. In *USENIX Annual Technical Conference (ATC)*, pages 279–290, 2013.

[28] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. In *ACM SIGPLAN Notices*, volume 48, pages 51–62, 2013.

[29] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. Cmd: classification-based memory deduplication through page access characteristics. In *ACM SIGPLAN Notices*, volume 49, pages 65–76, 2014.

[30] Yinjin Fu, Hong Jiang, Nong Xiao, Lei Tian, and Fang Liu. Aa-dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 112–120. IEEE, 2011.

[31] 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 *USENIX File and Storage Technologies (FAST)*, volume 11, pages 77–90, 2011.

[32] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–9. IEEE, 2009.

[33] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *USENIX Annual Technical Conference (ATC)*, volume 2012, pages 285–296, 2012.

[34] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *USENIX File and Storage Technologies (FAST)*, volume 9, pages 111–123, 2009.

[35] Bob Jenkins. A hash function for hash table lookup. `http://www.burtleburtle.net/bob/hash/doobs.html`.

[36] Wikipedia - avalanche effect. `https://en.wikipedia.org/wiki/Avalanche_effect`.

[37] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228:15–23, 1973.

[38] Edward Dawson, Helen Gustafson, and Anthony N Pettitt. Strict key avalanche criterion. *Australasian Journal of Combinatorics*, 6:147–153, 1992.

[39] B Jenkins. Spookyhash: a 128-bit non-cryptographic hash (2010). *http://burtleburtle.net/bob/hash/spooky. html*, 2014.

[40] Yann Collet. xxhash-extremeley fast hash algorithm, 2016.

[41] Austin Appleby. Murmurhash 2.0, 2008.

[42] Linux kernel intel p-state driver. `https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt`.

[43] aufs another unionfs. `http://aufs.sourceforge.net/aufs.html`.