Adopting Zoned Storage in Distributed Storage Systems

Abutalib Aghayev

CMU-CS-20-130

August 2020

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the Los Alamos National Laboratory award number: 394903; by a Hima and Jive Fellowship in Computer Science for International Students; and by a Carnegie Mellon University Parallel Data Lab (PDL) Entrepreneurship Fellowship. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: file systems, distributed storage systems, shingled magnetic recording, zoned namespaces, zoned storage, hard disk drives, solid-state drives

Abstract

Hard disk drives and solid-state drives are the workhorses of modern storage systems. For the past several decades, storage systems software has communicated with these drives using the *block interface*. The block interface was introduced early on with hard disk drives, and as a result, almost every storage system in use today was built for the block interface. Therefore, when flash memory based solid-state drives recently became viable, the industry chose to emulate the block interface on top of flash memory by running a *translation layer* inside the solid-state drives. This translation layer was necessary because the block interface was not a direct fit for the flash memory. More recently, hard disk drives are shifting to *shingled magnetic recording*, which increases capacity but also violates the block interface. Thus, emerging hard disk drives are also emulating the block interface by running a translation layer inside the drive. Emulating the block interface using a translation layer, however, is becoming a source of significant performance and cost problems in distributed storage systems.

In this dissertation, we argue for the elimination of the translation layer—and consequently the block interface. We propose adopting the emerging *zone interface* instead—a natural fit for both high-capacity hard disk drives and solid-state drives—and rewriting the *storage backend* component of distributed storage systems to use this new interface. Our thesis is that adopting the zone interface using a special-purpose storage backend will improve the cost-effectiveness of data storage and the predictability of performance in distributed storage systems.

We provide the following evidence to support our thesis. First, we introduce Skylight, a novel technique to reverse engineer the translation layers of modern hard disk drives and demonstrate the high *garbage collection* overhead of the translation layers. Second, based on the insights from Skylight we develop ext4-lazy, an extension of the popular ext4 file system, which is used as a storage backend in many distributed storage systems. Ext4-lazy significantly improves performance over ext4 on hard disk drives with a translation layer, but it also shows that in the presence of a translation layer it is hard to achieve the full potential of a drive with evolutionary file system changes. Third, we show that even in the absence of a translation layer, the abstractions provided by *general-purpose* file systems such as ext4 are inappropriate for a storage backend. To this end, we study a decade-long evolution of a widely used distributed storage system, Ceph, and pinpoint technical reasons that render general-purpose file systems unfit for a storage backend. Fourth, to show the advantage of a *special-purpose* storage backend in adopting the zone interface, as well as the advantages of the zone interface itself, we extend BlueStore—Ceph's special-purpose storage backend—to work on zoned devices. As a result of this work, we demonstrate how having a special-purpose backend in Ceph enables quick adoption of the zone interface, how the zone interface eliminates in-device garbage collection when running RocksDB (a key-value database used for storing metadata in BlueStore), and how the zone interface enables Ceph to reduce tail latency and increase cost-effectiveness of data storage without sacrificing performance.

Acknowledgments

My doctoral journey has been far from the ordinary: it spanned two schools, five advisors, and more years than I care to admit. In hindsight, it was a blessing in disguise—I met many great people, some of whom became life long friends. Below, I will try to acknowledge them, and I apologize if I miss anyone.

I am grateful to George Amvrosiadis and Greg Ganger for agreeing to advise me at a critical stage of my studies and for helping me cross the finish line. It was a packed two years of work, but George and Greg's laid-back and cheerful attitude made them bearable and fun. I thank Garth Gibson and Eric Xing for agreeing to advise me upon my arrival at CMU and for introducing me to the field of machine learning systems. Garth was instrumental in my coming to CMU, and I received his unwavering support throughout the years, for which I am grateful. I am grateful to Peter Desnoyers for advising me at Northeastern, for supporting my decision to reapply to graduate schools late into my studies, and for always being there for me. Remzi Arpaci-Dusseau has been my unofficial advisor since that lucky day when I first met him at SOSP '13. I am grateful for his continuous support inside and outside of academia and for his serving in my thesis committee. While at CMU, I also briefly worked with Phil Gibbons and Ehsan Totoni, and I am grateful to them for the opportunity.

I was also fortunate to have great collaborators from the industry. Sage Weil and Theodore Ts'o are two super hackers, respectively responsible for leading the development of a distributed storage system and a file system used by millions. Yet they found time to meet with me regularly, to patiently answer my questions, and to discuss research directions. Towards the end of my studies, I asked Sage to serve on my thesis committee, and he kindly agreed. I am grateful to Sage and Ted for their support. Matias Bjørling turned my request for his slides into an internship, and thus started our collaboration on zoned storage, which turned out to be crucial to my dissertation. Tim Feldman thoroughly answered my questions as I was getting started with my research on shingled magnetic recording disks. I am grateful to Matias, Tim, and many other brilliant engineers and hackers, including Samuel Just, Kefu Chai, Igor Fedotov, Mark Nelson, Hans Holmberg, Damien Le Moal, Marc Acosta, Mark Callaghan, and Siying Dong, whose answers to my countless queries greatly contributed to my technical knowledge base.

As a member of the Parallel Data Lab (PDL) I benefited immensely from the opportunities it provided, and I am therefore grateful to all who make the PDL a great lab to be a part of. Thank you Jason Boles, Chad Dougherty, Mitch Franzos, and Chuck Cranor for keeping the PDL clusters running and for fixing my never-ending technical problems. Thank you Joan Digney for always polishing my slides and posters and proofreading my papers. Thank you Karen Lindenfelser, Bill Courtright, and Greg Ganger for organizing the wonderful PDL events and for keeping it all running, and Garth Gibson, for founding the PDL. Thank you all the organizations that contribute to the PDL, and thank you Hugo Patterson for the PDL Entrepreneurship Fellowship.

I am thankful to the staff of the Computer Science Department (CSD), including Deb Cavlovich for her help with the administrative aspects of the doctoral process, and Catherine Copetas for her help with organizing CSD events. I am thankful to the anonymous donor who established the Hima and Jive Fellowship in Computer Science for International Students.

During my studies I made many friends. It was a privilege to share an office with Samuel Yeom from whom I learned so much, ranging from the finer details of the U.S. political system and why Ruth Bader Ginsburg is not retiring yet, to countless obscure facts, thanks to our daily ritual of solving The New York Times crossword after lunch. (To be fair, Sam would mostly solve it and I would try to catch up and feel elated when contributing an answer or two.) Michael Kuchnik, Jin Kyu Kim, and Aurick Qiao were the best collaborators one could wish for: smart, humble, hard working, and fun. Saurabh Kadekodi and Jinliang Wei showed me the ropes around the campus and the department when I first arrived at CMU. As I was preparing to leave CMU, Anuj Kalia shared with me his experience of navigating the academic job market and gave me useful tips. I thank all my friends, colleagues, and professors at CMU and Northeastern, including Umut Acar, Maruan Al-Shedivat, David Andersen, Joy Arulraj, Kapil Arya, Mehmet Berat Aydemir, Ahmed Badawi, Nathan Beckmann, Naama Ben-David, Daniel Berger, Brandon Bohrer, Sol Boucher, Harsh Raju Chamarthi, Dominic Chen, Andrew Chung, Gene Cooperman, Henggang Cui, Wei Dai, Omar Delen, Chris Fallin, Pratik Fegade, Matthias Felleisen, Mohammed Hossein Hajkazemi, Mor Harchol-Balter, Aaron Harlap, Kevin Hsieh, Angela Jiang, David Kahn, Rajat Kateja, Ryan Kavanagh, Jack Kosaian, Yixin Luo, Lin Ma, Pete Manolios, Sara McAllister, Prashanth Menon, Alan Mislove, Mansour Shafaei Moghaddam, Todd Mowry, Ravi Teja Mullapudi, Onur Mutlu, Guevara Noubir, Jun Woo Park, İlqar Ramazanlı, Alexey Tumanov, David Wajc, Christo Wilson, Pengtao Xie, Jason Yang, Hao Zhang, Huanchen Zhang, and Qing Zheng, for their help and support and random fun chats over the years.

My doctoral adventure would not have even started had it not been for the constant support of my undergraduate professors. Thank you from the bottom of my heart, Levent Akın, Mehmet Ufuk Çağlayan, and Can Özturan, for supporting me despite my shenanigans and bad grades and never doubting me! And I would not have become interested in computers were it not for my high school teacher, Emrullah Kaya. Thank you for gifting me that Pascal book and for teaching me how to debug code!

Finally, I am grateful to my parents and to my wife, Sevinj, for their unconditional love and support. I do not think I can repay my debt to you, but I will try for the rest of my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data has surpassed oil as the world's most valuable resource [56] [1]. But unlike oil, of which there is a limited supply, the world is drowning in data [155]. Extracting value from this new resource requires large-scale distributed storage systems that can scale seamlessly to store vast volumes of data in a *cost-effective* manner, granting the opportunity to *efficiently*—with high throughput and low latency—access and process the stored data.

The systems community is on a constant quest for designing ever more cost-effective and efficient distributed storage systems. To tame the complexity, these systems are conventionally constructed in layers. At their penultimate layer, distributed storage systems rely on *general-purpose* file systems. In a layer below, general-purpose file systems communicate with the storage devices using the *block interface*. The block interface is a poor match for modern high-capacity hard disk drives and solid-state drives, and emulating it in modern storage devices imposes a *"block interface tax"*—unpredictable performance [80, 221] and increased cost [206]—on storage devices and consequently on the distributed storage systems. Moving back up a layer, although relying on general-purpose file systems is convenient for distributed storage system designers, in the long term it is counterproductive because it imposes a *"file system tax"*—reduced performance [143][10, 149], accidental complexity and rigidity in the codebase [201].

In this dissertation we explore a clean-slate redesign of distributed storage systems that avoids both the block interface tax and the file system tax, in light of an emerging storage interface—the *zone interface*—which is a natural interface for modern storage devices. To this end, we study and quantify the block interface tax on modern hard disk drives and the file system tax on the Ceph distributed storage system. We then adapt Ceph to the zone interface and demonstrate how it improves (1) cost-effectiveness, by storing data on high-capacity hard drives and (2) performance, by achieving higher I/O throughput and lower tail latency on these drives.

1.1 10,000-feet View of Distributed Storage Systems

Distributed storage systems are complex, with many moving parts. In this section we give a high-level overview of the architecture found in most such systems and outline the scope of our work.

[1] We color code citations: peer-reviewed articles are cited in blue and non-peer-reviewed articles are cited in red.

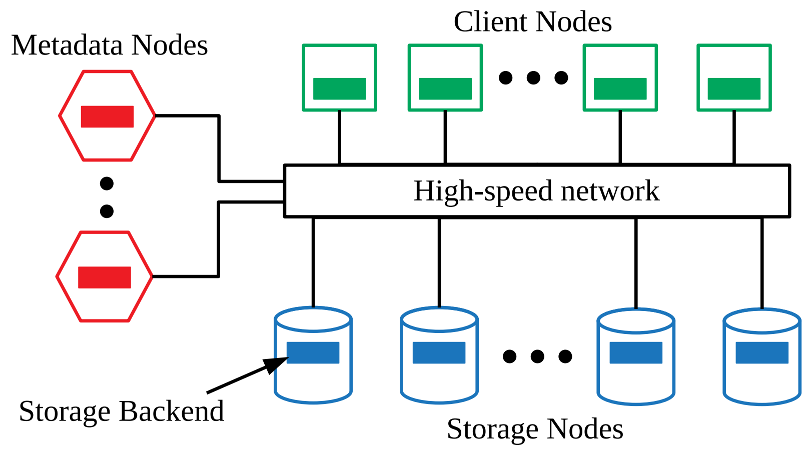


Figure 1.1: 10,000-feet view of the architecture found in most distributed storage systems. The scope of this dissertation is the storage backend that runs on every storage node.

Distributed storage systems aggregate the storage space of multiple physical nodes into a single unified data store that offers high-bandwidth and parallel I/O, horizontal scalability, fault tolerance, and other desirable features. Over the years many such systems have been developed [74, 84, 202, 205][58], but almost all of them have followed the same high-level design shown in Figure 1.1: There is the *client-side* implemented as a kernel module or a library in the client nodes that enables users to transparently access the storage system; there is the *storage backend* implemented in the storage nodes that serves user I/O requests; and there is the *metadata service* implemented on the metadata nodes that lets the clients know which storage nodes to read data from.

The scope of this dissertation is the storage backend that runs on the storage nodes. The storage nodes form the bulk of the nodes in a typical cluster running a distributed storage system, and they store all of the data in the system. The storage backend runs on every storage node, receives I/O requests over the network and serves them by accessing the storage devices that are locally attached to the storage node. *The storage backend is a crucial component that sits in the I/O path and plays a key role in the performance of the overall system.*

1.2 The State of Current Storage Backends

Distributed storage systems differ in the details of how they design the storage backend. Almost all of them, however, rely on a general-purpose file system, such as ext4 or XFS, for implementing the storage backend [74, 84, 87, 177, 202, 205][58, 154, 190, 209]. This design decision has come largely unchallenged for the past several decades. Perhaps unwittingly, it has locked the distributed storage systems into running only on the storage devices that support the *block interface*. The block interface is a method for the storage software, such as a file system, to communicate with a storage device, and storage devices that support the block interface are called *block devices*. Almost every major file system is developed to run on block devices.

The block interface and general-purpose file systems impose significant performance taxes on distributed storage systems. Although they appear to be interrelated, in the following sections we decouple them, explain how we got here, and how each independently contributes to performance problems.

1.2.1 The Block Interface Tax

There have been tremendous technological advances in hard disk drive (HDD) manufacturing process since IBM's introduction of the first HDD in 1956 [88]. The essence of how data is internally stored in HDDs, however, has largely remained the same: bit patterns that can be *independently* and *randomly* accessed and manipulated. The block interface has evolved over the years to match this essence of HDDs. It is a simple linear addressing scheme that represents the storage device as an array of 512-byte blocks. Each block is identified by an integer index and the blocks can be read or written in random order.

The earliest devices supporting the block interface appeared in 1986 [214], and it has been the most dominant interface since then. Almost every major file system developed in the past few decades was designed for the block interface. Consequently, the block interface is deeply entrenched in the storage ecosystem. So much that when the NAND flash memory paved the way for a new breed of storage devices, the industry decided to emulate the block interface on top of flash memory, even though the block interface was a bad match for how flash memory operated. Thus, solid-state drives (SSDs) were born.

Unlike HDDs that store data in blocks that can be randomly read or written, modern SSDs store data in *pages* with sizes similar to that of blocks. These pages are grouped into *erase units* spanning several megabytes, and all of the pages within an erase unit must be written sequentially. Hence, in-place overwrite of pages is impossible, and rewriting pages is only possible after first erasing all of the pages within an erase unit. To emulate the block interface on top of these constraints SSDs internally run a *translation layer* that maps blocks to pages on erase units. To handle a block overwrite, the translation layer writes data to a new page, remaps the block to the new page, and marks the old page as stale. To reclaim space from the stale pages, the translation layer performs *garbage collection* at arbitrary times: it copies non-stale pages from one erase unit to another, remaps the corresponding blocks, and erases all the pages in the original erase unit.

Emulating the block interface using all this machinery comes with a high cost and performance penalty. Enterprise SSDs overprovision up to 28% of flash memory and use gigabytes of RAM for the efficient operation of the translation layer, significantly increasing the device cost [206]. Garbage collection performs extra internal writes that increases *write amplification*—the ratio of writes issued by the host to all the writes performed internally by the device—leading to early wear-out of flash cells. More importantly, garbage collection operations interfere with user requests and lead to high tail latencies in distributed storage systems [80, 105].

Perhaps surprisingly, the block interface is becoming a burden for modern high-capacity HDDs as well. To increase capacity, all three HDD manufacturers are shifting [130, 175, 176] to Shingled Magnetic Recording (SMR) [215]. SMR HDDs internally store data in *zones* spanning hundreds of megabytes. Not unlike erase units in flash, the zones in SMR HDDs must be written sequentially and must be *reset* before being written again. To ease the adoption of SMR HDDs, drive manufacturers have introduced Drive-Managed SMR (DM-SMR) HDDs which, just like SSDs internally run a translation layer to emulate the block interface. And just like SSDs they too suffer from an unpredictable performance [2].

In summary, emulating the block interface with a translation layer on modern SSDs and HDDs increases the device cost and results in an unpredictable performance and high write amplification due to garbage collection performed by the translation layer. The unpredictable performance is undesirable in general, and in particular it is a major contributor to high tail latencies in distributed storage systems. We refer to these cost and performance overheads collectively as the *block interface tax*.

1.2.2 The File Systems Tax

The developers of distributed storage systems have conventionally designed storage backends to run on top of *general-purpose* file systems [74, 84, 87, 177, 202, 205][58, 154, 190, 209]. This convention is attractive at first glance because file systems implement most of the functionality expected from a storage backend, but in the long run it is counterproductive.

Developers hit the limitations of general-purpose file systems once they start pushing file systems to meet the scaling requirements of distributed storage systems, for example, when they store millions of entries in a single directory [10, 89, 149]. They continue, however, with their original design and try and fit general-purpose file system abstractions to their needs, incurring significant accidental complexity [25], which results in lost performance and hard-to-maintain, fragile code. This design decision is often rationalized by arguing that building a storage backend from scratch is akin to building a new general-purpose file system, which is known to take a decade on average [60, 210, 211][116]. Recent experience, however, shows that building a *special-purpose* storage backend from scratch can reach production quality in less than two years while significantly outperforming storage backends implemented on top of general-purpose file systems [5, 6].

Another major disadvantage of relying on file systems is their resistance to change. Once a file system matures, "*... file system developers tend toward a high level of conservatism when it comes to making changes; given the consequences of mistakes, this seems like a healthy survival trait.*" [45]. Effectively supporting the emerging storage hardware, on the other hand, often requires drastic changes.

In summary, relying on general-purpose file systems for distributed storage backends leads to accidental complexity and lost performance, and it hinders swift and effective adoption of emerging storage technologies. We refer to these complexity, performance, and rigidity overheads collectively as the *file system tax*.

1.3 Zoned Storage and Dilemma of Distributed Storage Systems

Although they are worlds apart in how they are built and how they work, SMR HDDs and SSDs both manage their storage medium in the same manner: as a sequence of *zones*, each of which spans megabytes and can only be written sequentially. The Zoned Storage initiative [51] leverages this similarity to introduce the *zone interface* for managing both SMR HDDs and SSDs. The SMR HDDs that support the zone interface are called Host-Managed SMR (HM-SMR) HDDs, and the SSDs that support the zone interface are caled Zoned Namespace (ZNS) SSDs.

These *zoned devices* are specifically targeted at large-scale distributed storage systems, and they come with multiple features that are attractive at scale. First, they are expected to scale to capacities of tens of terabytes in a single device. Second, zoned devices improve cost-effectiveness of data storage without incurring garbage collection overhead. Specifically, HM-SMR HDDs increase capacity by 20% over conventional HDDs; and unlike enterprise SSDs, which declare themselves dead after 28% percent of their flash memory cells wear out, ZNS SSDs support dynamically decreasing the device capacity while continuing to store data on the remaining healthy cells. Third, the zoned devices move the control over explicit data placement from device to host: now the host implements garbage collection and controls when it happens, thereby avoiding high tail latencies caused by garbage collection kicking in at any moment in regular SSDs and DM-SMR HDDs. More importantly, the host now has a chance to intelligently place data directly on zones and eliminate garbage collection for certain common workloads [4].

Thus, going forward, the distributed storage systems face a dilemma: should they adopt the zoned storage and, if so, how? This dissertation provides evidence that distributed storage systems can avoid both—the block interface tax and the file system tax—by adopting zoned storage using a special-purpose storage backend.

1.4 Thesis Statement and Contributions

Thesis statement: *Distributed storage systems can improve the cost-effectiveness of data storage and the predictability of performance if they abandon the block interface for the zone interface and general-purpose file systems for special-purpose storage backends.*

In this dissertation, we provide the following evidence to support our thesis statement:

We show that the block interface tax is becoming increasingly prohibitive. Distributed storage systems can run their storage backends on DM-SMR HDDs to be cost-effective and on regular SSDs to be performant. Both of these are block devices that emulate the block interface with a translation layer. Researchers have shown that the block interface tax—the garbage collection overhead incurred by the translation layer—leads to high tail latencies in SSDs [47, 80, 221]. We show that the garbage collection overhead is even higher in DM-SMR HDDs by performing a thorough performance characterization of popular DM-SMR HDDs (Chapter 2).

We show that it is hard to eliminate the block interface tax by evolutionary changes to general-purpose file systems. Distributed storage systems can eliminate the block interface tax by either modifying current file systems to work on zoned devices or by modifying file systems to avoid I/O patterns that cause garbage collection inside an emulated block device. Attempts to do the former have stalled due to technical difficulties [35, 140], suggesting the first option to be impractical. We show that while the second option can significantly improve the performance of ext4—the canonical file system for storage backends—on DM-SMR HDDs, it is hard to eliminate the garbage collection-inducing behavior (Chapter 3).

We study and quantify the file system tax and demonstrate that general-purpose file systems are unfit as distributed storage backends. We study the evolution of the storage backends in Ceph, a widely used distributed storage system, over ten years, and we pinpoint technical reasons as to why general-purpose file system abstractions are either too slow or inappropriate for the needs of storage backends and how they lead to performance problems. *The file system tax would still exist even in the absence block interface tax* (Chapter 4).

We demonstrate that a distributed storage system that already avoids the file system tax is also well-suited to avoid the block interface tax by quickly and effectively adopting the zoned storage. We adapt BlueStore, a special-purpose storage backend in Ceph, to zoned devices. We demonstrate that BlueStore can quickly adopt the zoned storage, and as a result, Ceph can leverage the extra capacity offered by SMR with high throughput and low tail latency, avoiding the block interface tax of DM-SMR HDDs (Chapter 5).

1.4.1 Contributions

This dissertation makes the following technical contributions:

- A novel methodology, Skylight, that combines software and hardware techniques to reverse engineer key properties of DM-SMR HDDs (Chapter 2).
- A full reverse engineering—using Skylight—of two real DM-SMR HDDs and their performance characterization (Chapter 2).
- A set of kernel modules that implement state-of-the-art translation layer algorithms and emulate DM-SMR HDDs on top of conventional HDDs (Chapter 2).
- The design and implementation of ext4-lazy, an extension of the ext4 file system, which reduces I/O patterns that cause garbage collection on emulated block devices (Chapter 3).
- A demonstration of performance improvements of ext4-lazy over ext4 on four DM-SMR HDDs from two vendors using micro- and macro-benchmarks (Chapter 3).
- The discovery of a long-standing bottleneck in ext4 for metadata-heavy workloads on HDDs and a fix for the bottleneck (Chapter 3).
- A longitudinal study of storage backend evolution in Ceph that dissects technical reasons leading to performance problems when building a storage backend on top of a general-purpose file system (Chapter 4).
- An introduction to the design of BlueStore, a clean-slate storage backend in Ceph, the challenges BlueStore solves and the opportunities it provides (Chapter 4).
- An extension to RocksDB, a popular key-value store that BlueStore uses for storing metadata, which enables RocksDB to run on zoned devices (Chapter 5).
- A demonstration of a surprisingly high write amplification in an enterprise SSD when running a concurrent sequential workload, such as RocksDB (Chapter 5).
- A demonstration of performance improvements of RocksDB running on zoned devices over RocksDB running on block devices with a translation layer (Chapter 5).
- An extension to BlueStore that enables it to run on zoned devices and an extension to Ceph that enables it to leverage the zone interface and the redundancy of data to avoid high tail latencies (Chapter 5).

- An end-to-end demonstration on a cluster of how Ceph achieves cost-effective data storage and high performance when running on HM-SMR HDDs compared to when running on DM-SMR HDDs. (Chapter 5).

1.5 Thesis Outline

The rest of this thesis is organized as follows: In Chapter 2, we study the inner workings of a DM-SMR drive—a modern high-capacity hard disk drive—and we show that the block interface tax is becoming even more prohibitive with these drives. We then put our newly acquired knowledge into use in Chapter 3, and we attempt to reduce the block interface tax in distributed storage systems by optimizing ext4—a file system used as a storage backend by many distributed storage systems—to reduce the garbage collection overhead in DM-SMR drives. While our work improves the performance of ext4 significantly on DM-SMR drives, it also demonstrates that it is hard to avoid the block interface tax by making evolutionary changes to general-purpose file systems. In Chapter 4 we demonstrate the file system tax—we show that implementing a storage backend on top of general-purpose file systems introduces significant performance overhead and hinders the adoption of novel storage hardware. In Chapter 5 we demonstrate that a distributed storage system which already avoids the file system tax is well-suited to avoiding the block interface tax and achieve cost-effective data storage, high throughput, and low tail latency. We conclude in Chapter 6.

Chapter 2

Understanding and Quantifying the Block Interface Tax in DM-SMR Drives

In this chapter we introduce SMR and describe how it increases capacity over Conventional Magnetic Recording (CMR). We then introduce Skylight, a novel methodology that combines software and hardware techniques to reverse engineer the translation layer operation in DM-SMR drives. Using Skylight we quantify the block interface tax—the performance and cost overhead stemming from emulating the block interface using a translation layer—on DM-SMR drives and introduce guidelines for their effective use.

2.1 Magnetic Recording Techniques and Overview of Skylight

In the nearly 60 years since the first hard disk drive has been introduced [88], it has become the mainstay of computer storage systems. In 2013 the hard drive industry shipped over 400 exabytes [164] of storage, or almost 60 gigabytes for every person on earth. Although facing strong competition from NAND flash-based solid-state drives (SSDs), magnetic disks hold a 10× advantage over flash in both total bits shipped [157] and per-bit cost [55], an advantage that will persist if density improvements continue at current rates.

The most recent growth in disk capacity is the result of improvements to perpendicular magnetic recording (PMR) [146], which has yielded terabyte drives by enabling bits as short as 20 nm in tracks 70 nm wide [167], but further increases will require new technologies [191]. SMR is the first such technology to reach market: 5 TB drives are available from Seagate [166] and shipments of 8 TB and 10 TB drives have been announced by Seagate [165] and HGST [83]. Other technologies (Heat-Assisted Magnetic Recording [110] and Bit-Patterned Media [53]) remain in the research stage, and may in fact use shingled recording when they are released [196].

Shingled recording spaces tracks more closely, so they overlap like rows of shingles on a roof, squeezing more tracks and bits onto each platter [215]. The increase in density comes at a cost in complexity, as modifying a disk sector will corrupt other data on the overlapped tracks, requiring copying to avoid data loss [9, 75]. Rather than push this work onto the host file system [115], DM-SMR drives shipped to date preserve compatibility with existing drives by implementing a Shingle Translation Layer (STL) [29, 78][75] that emulates the block interface on top of this complexity.

Like an SSD, an DM-SMR drive combines out-of-place writes with dynamic mapping in order to efficiently update data, resulting in a drive with performance much different from that of a CMR drive due to seek overhead for out-of-order operations. Unlike SSDs, however, which have been extensively measured and characterized [22, 32], little is known about the behavior and performance of DM-SMR drives and their translation layers, or how to optimize file systems, storage arrays, and applications to best use them.

We introduce a methodology for measuring and characterizing such drives, developing a specific series of micro-benchmarks for this characterization process, much as has been done in the past for conventional drives [77, 186, 216]. We augment these timing measurements with a novel technique that tracks actual head movements via high-speed camera and image processing and provides a source of reliable information in cases where timing results are ambiguous.

We validate this methodology on three different emulated drives that use STLs previously described in the literature [29, 78][40], implemented as a Linux *device mapper target* [49] over a conventional drive, demonstrating accurate inference of properties. We then apply this methodology to 5 TB and 8 TB DM-SMR drives provided by Seagate, inferring the STL algorithm and its properties and providing the first public characterization of such drives.

Using our approach we are able to discover important characteristics of the Seagate DM-SMR drives and their translation layer, including the following:

- *Cache type and size.* The drives use a persistent disk cache of 20 GiB and 25 GiB on the 5 TB and 8 TB drives, respectively, with high random write speed until the cache is full. The effective cache size is a function of write size and queue depth.
- *Persistent cache structure.* The persistent disk cache is written as journal entries with quantized sizes—a phenomenon absent from the academic literature on SMRs.
- *Block Mapping.* Non-cached data is statically mapped, using a fixed assignment of logical block addresses (LBAs) to physical block addresses (PBAs), similar to that used in CMR drives, with implications for performance and durability.
- *Band size.* DM-SMR drives organize data in *bands*—a set of contiguous tracks that are re-written as a unit; the examined drives have a small band size of 15–40 MiB.
- *Cleaning mechanism.* Aggressive cleaning during idle times moves data from the persistent cache to bands; cleaning duration is 0.6–1.6 s per modified band.

Our results show the details that may be discovered using Skylight, most of which impact (negatively or positively) the performance of different workloads, as described in § 2.6. These results—and the toolset allowing similar measurements on new drives—should thus be useful to users of DM-SMR drives, both in determining what workloads are best suited for these drives and in modifying applications to better use them. In addition, we hope that they will be of use to designers of DM-SMR drives and their translation layers, by illustrating the effects of low-level design decisions on system-level performance.

2.2 Background on Shingled Magnetic Recording

Shingled recording is a response to limitations on areal density with perpendicular magnetic recording due to the superparamagnetic limit [191]. In brief, for bits to become smaller, write

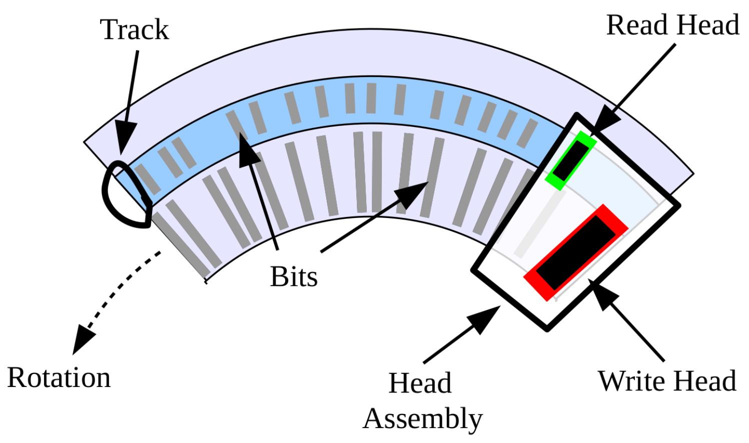


Figure 2.1: Shingled disk tracks with head width $k = 2$.

heads must become narrower, resulting in weaker magnetic fields. This requires lower coercivity (easily recordable) media, which is more vulnerable to bit flips due to thermal noise, requiring larger bits for reliability. As the head gets smaller this minimum bit size gets larger, until it reaches the width of the head and further scaling is impossible.

Several technologies have been proposed to go beyond this limit, of which SMR is the simplest [215]. To decrease the bit size further, SMR reduces the track width while keeping the head size constant, resulting in a head that writes a path several tracks wide. Tracks are then overlapped like rows of shingles on a roof, as seen in Figure 2.1. Writing these overlapping tracks requires only incremental changes in manufacturing, but much greater changes in storage software, as it becomes impossible to re-write a single sector without destroying data on the overlapped sectors.

For maximum capacity an SMR drive could be written from beginning to end, utilizing all tracks. Modifying any of this data, however, would require reading and re-writing the data that would be damaged by that write, and data to be damaged by the re-write and so on, until the end of the surface is reached. This cascade of copying may be halted by inserting *guard regions*—tracks written at the full head width—so that the tracks before the guard region may be re-written without affecting any tracks following it, as shown in Figure 2.2. These guard regions divide each disk surface into re-writable bands; since the guards hold a single track's worth of data, storage efficiency for a band size of b tracks is $\frac{b}{b+k-1}$.

Given knowledge of these bands, a host file system can ensure they are only written sequentially, for example, by implementing a log-structured file system [115, 158]. Standards have also been developed to allow a drive to identify these bands to the host [94]: HM-SMR drives expose *zones* that are an order of magnitude larger than bands, which must also be written sequentially. At the time of this work, however, these standards were still in draft form and no drives based on them were available on the open market.

Alternately the DM-SMR drives present a standard re-writable block interface that is implemented by an internal Shingle Translation Layer, much as an SSD uses a Flash Translation Layer (FTL). Although the two are logically similar, appropriate algorithms differ due to differences in the constraints placed by the underlying media: (a) high seek times for non-sequential access,

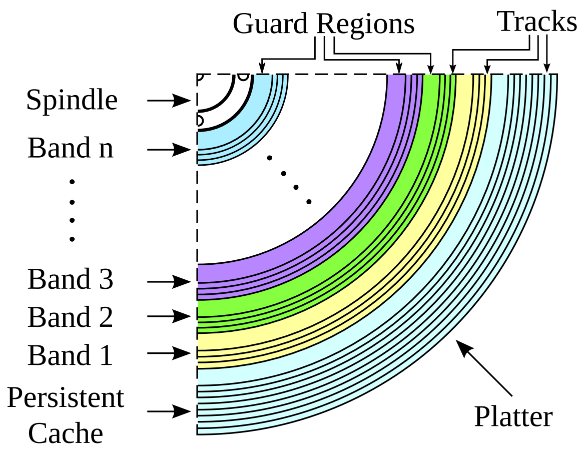


Figure 2.2: Surface of a platter in a hypothetical DM-SMR drive. A persistent cache consisting of 9 tracks is located at the outer diameter. The guard region that separates the persistent cache from the first band is simply a track that is written at a full head width of *k* tracks. Although the guard region occupies the width of *k* tracks, it contains a single track's worth of data and the remaining *k-1* tracks are wasted. The bands consist of 4 tracks, also separated with a guard region. Overwriting a sector in the last track of any band will not affect the following band. Overwriting a sector in any of the tracks will require reading and re-writing all of the tracks starting at the affected track and ending at the guard region within the band.

(b) lack of high-speed reads, (c) use of large (10s to 100s of MB) cleaning units, and (d) lack of wear-out, eliminating the need for wear leveling.

These translation layers typically store all data in bands where it is mapped at a coarse granularity, and devote a small fraction of the disk to a *persistent cache*, as shown in Figure 2.2, which contains copies of recently-written data. Data that should be retrieved from the persistent cache may be identified by checking a *persistent cache map* (or *exception map*) [29, 78]. Data is moved back from the persistent cache to bands by the process of *cleaning* (also known as *garbage collection*), which performs read-modify-write (RMW) on every band whose data was overwritten. The cleaning process may be *lazy*, running only when the free cache space is low, or *aggressive*, running during idle times.

In one translation approach, a *static mapping* algorithmically assigns a *native location* [29], a physical block address (PBA) to each logical block address (LBA) in the same way as is done in a CMR drive. An alternate approach uses coarse-grained *dynamic mapping* for non-cached LBAs [29], in combination with a small number of free bands. During cleaning, the drive writes an updated band to one of these free bands and then updates the dynamic map, potentially eliminating the need for a temporary staging area for cleaning updates and sequential writes.

In any of these cases drive operation may change based on the setting of the *volatile cache* (enabled or disabled) [160]. When the volatile cache is disabled, writes are required to be persistent before completion is reported to the host. When it is enabled, persistence is only guaranteed after a `FLUSH CACHE` command [213] or a write command with the Force Unit Access (FUA) bit set.

Drive Name	STL	Persistent Cache Type and Size	Disk Cache Multiplicity	Cleaning Type	Band Size	Mapping Type	Size
Emulated-SMR-1	Set-associative	Disk, 37.2 GiB	Single at ID	Lazy	40 MiB	Static	3.9 TB
Emulated-SMR-2	Set-associative	Flash, 9.98 GiB	N/A	Lazy	25 MiB	Static	3.9 TB
Emulated-SMR-3	Fully-associative	Disk, 37.2 GiB	Multiple	Aggressive	20 MiB	Dynamic	3.9 TB

Table 2.1: Emulated DM-SMR drive configurations. (For brevity, we drop DM from DM-SMR when naming the drives.)

2.3 Test Drives

We now describe the drives we study. First, we discuss how we emulate three DM-SMR drives using our implementation of two STLs described in the literature. Second, we describe the real DM-SMR drives we study in this paper and the real CMR drive we use for emulating DM-SMR drives.

2.3.1 Emulated Drives

We implement Cassuto et al.'s *set-associative* STL [29] and a variant of their *S-blocks* STL [29][79], which we call *fully-associative* STL, as Linux device mapper targets. These are kernel modules that export a pseudo block device to user-space that internally behaves like a DM-SMR drive—the module translates incoming requests using the translation algorithm and executes them on a CMR drive.

The *set-associative* STL manages the disk as a set of N iso-capacity (same-sized) data bands, with typical sizes of 20–40 MiB, and uses a small (1–10%) section of the disk as the persistent cache. The persistent cache is also managed as a set of n iso-capacity cache bands where $n \ll N$. When a block in data band a is to be written, a cache band is chosen through $(a \bmod n)$; the next empty block in this cache band is written and the persistent cache map is updated. Further accesses to the block are served from the cache band until cleaning moves the block to its native location, which happens when the cache band becomes full.

The *fully-associative* STL, on the other hand, divides the disk into large (we used 40 GiB) zones and manages each zone independently. The notion of zone here comes from *zone bit recording* [**?**] used in hard drives and is unrelated to SMR zones. A zone starts with 5% of its capacity provisioned to free bands for handling updates. When a block in logical band a is to be written to the corresponding physical band b, a free band c is chosen and written to and the persistent cache map is updated. When the number of free bands falls below a threshold, cleaning merges the bands b and c and writes it to a new band d and remaps the logical band a to the physical band d, freeing bands b and c in the process. This dynamic mapping of bands allows the fully-associative STL to handle streaming writes with zero overhead.

To evaluate the accuracy of our emulation strategy, we implemented a pass-through device mapper target and found negligible overhead for our tests, confirming a previous study [147]. Although in theory, this emulation approach may seem disadvantaged by the lack of access to exact sector layout, in practice this is not the case—even in real DM-SMR drives, the STL running inside the drive is implemented on top of a layer that provides linear PBAs by hiding sector layout and defect management [66]. Therefore, we believe that the device mapper target running on top of a CMR drive provides an accurate model for predicting the behavior of an STL implemented by the controller of a DM-SMR drive.

Table 2.1 shows the three emulated DM-SMR drive configurations we use in our tests. The first two drives use the set-associative STL, and they differ in the type of persistent cache and band size. The last drive uses the fully-associative STL and disk for the persistent cache. We do not have a drive configuration combining the fully-associative STL and flash for persistent cache. This is because the fully-associative STL aims to reduce long seeks during cleaning in disks, by using multiple caches evenly spread out on a disk, but flash does not suffer from long seek times.

To emulate a DM-SMR drive with a flash cache (Emulate-SMR-2) we use the Emulate-SMR-1 implementation, but use a device mapper `linear` target to redirect the underlying LBAs corresponding to the persistent cache, storing them on an SSD.

To check the correctness of the emulated DM-SMR drives we ran repeated burn-in tests using `fio` [13]. We also formatted emulated drives with the ext4 file system, compiled the Linux kernel on top, and successfully booted the system with the compiled kernel. The source code for the set-associative STL (1,200 lines of C) and a testing framework (250 lines of Go) are available at `http://sssl.ccs.neu.edu/skylight`.

2.3.2 Real Drives

Two real DM-SMR drives were tested: Seagate ST5000AS0011, a 5,900 RPM desktop drive (rotation time ≈ 10 ms) with four platters, eight heads, and 5 TB capacity (for brevity termed Seagate-SMR below), and Seagate ST8000AS0002, a similar drive with six platters, twelve heads and 8 TB capacity. Emulated drives use a Seagate ST4000NC001 (Seagate-CMR), a real CMR drive identical in drive mechanics and specification (except the 4 TB capacity) to the ST5000AS0011. Results for the 8 TB and 5 TB DM-SMR drives were similar; to save space, we only present results for the publicly-available 5 TB drive.

2.4 Characterization Tests

To motivate our drive characterization methodology we first describe the goals of our measurements. We then describe the mechanisms and methodology for the tests, and finally present results for each tested drive. For emulated DM-SMR drives, we show that the tests produce accurate answers, based on implemented parameters; for real DM-SMR drives we discover their properties. The behavior of the real DM-SMR drives under some of the tests engenders further investigation, leading to the discovery of important details about their operation.

2.4.1 Characterization Goals

The goal of our measurements is to determine key drive characteristics and parameters:

- *Drive type.* In the absence of information from the vendor, is a drive a DM-SMR or a CMR?

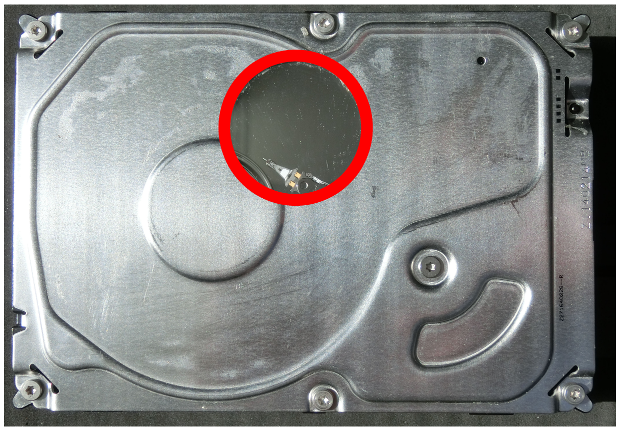


Figure 2.3: DM-SMR drive with the observation window encircled in red. Head assembly is visible parked at the inner diameter.

- *Persistent cache type.* Does the drive use flash or disk for the persistent cache? The type of the persistent cache affects the performance of random writes and reliable (volatile cache-disabled) sequential writes. If the drive uses disk for persistent cache, is it a single cache, or is it distributed across the drive [29][79]? The layout of the persistent disk cache affects the cleaning performance and the performance of the sequential read of a sparsely overwritten linear region.
- *Cleaning.* Does the drive use aggressive cleaning, improving performance for low duty-cycle applications, or lazy cleaning, which may be better for throughput-oriented ones? Can we predict the performance impact of cleaning?
- *Persistent cache size.* After some number of out-of-place writes the drive will need to begin a cleaning process, moving data from the persistent cache to bands so that it can accept new writes, negatively affecting performance. What is this limit, as a function of total blocks written, number of write operations, and other factors?
- *Band size.* Since a band is the smallest unit that may be re-written efficiently, knowledge of band size is important for optimizing DM-SMR drive workloads [29][40]. What are the band sizes for a drive, and are these sizes constant over time and space [68]?
- *Block mapping.* The mapping type affects performance of both cleaning and reliable sequential writes. For LBAs that are not in the persistent cache, is there a static mapping from LBAs to PBAs, or is this mapping dynamic?
- *Zone structure.* Determining the zone structure of a drive is a common step in understanding block mapping and band size, although the structure itself has little effect on external performance.

2.4.2 Test Mechanisms

The software part of Skylight uses `fio` to generate micro-benchmarks that elicit the drive characteristics. The hardware part of Skylight tracks the head movement during these tests. It resolves

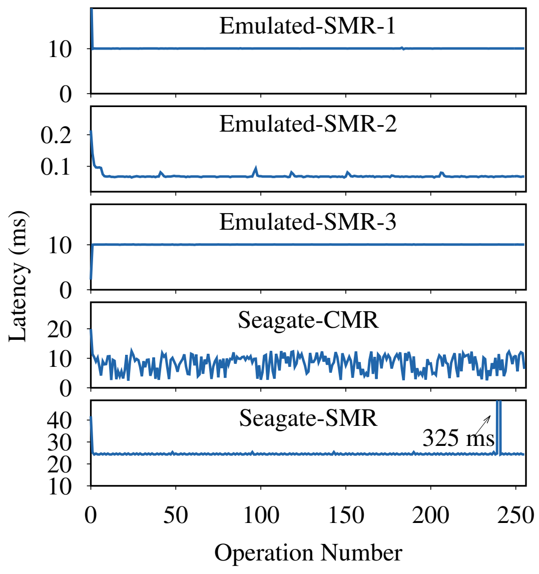


Figure 2.4: Discovering drive type using latency of random writes. Y-axis varies in each graph.

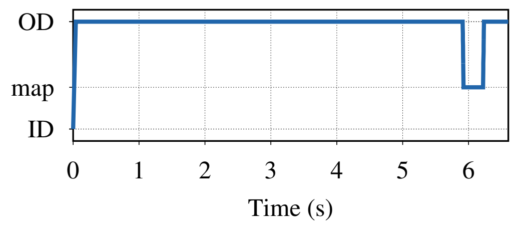


Figure 2.5: Seagate-SMR head position during random writes.

ambiguities when interpreting the latency of the data obtained from the micro-benchmarks and leads to discoveries that are not possible with micro-benchmarks alone. To track the head movements, we installed (under clean-room conditions) a transparent window in the drive casing over the region traversed by the head. Figure 2.3 shows the head assembly parked at the inner diameter (ID). We recorded the head movements using Casio EX-ZR500 camera at 1,000 frames per second and processed the recordings with `ffmpeg` to generate head location value for each video frame.

We ran the tests on a 64-bit Intel Core-i3 Haswell system with 16 GiB RAM and 64-bit Linux kernel version 3.14. Unless otherwise stated, we disabled kernel read-ahead, drive look-ahead and drive volatile cache using `hdparm`. Extensions to `fio` developed for these tests have been upstreamed. Slow-motion clips for the head position graphs shown in the paper, as well as the tests themselves, are available at `http://sssl.ccs.neu.edu/skylight`.

2.4.3 Drive Type and Persistent Cache Type

Test 1 exploits the unusual random write behavior of the DM-SMR drives to differentiate them from CMR drives. While random writes to a CMR drive incur varying latency due to random seek time and rotational delay, random writes to a DM-SMR drive are sequentially logged to the persistent cache with a fixed latency. If random writes are not local, DM-SMR drives that use separate persistent caches by the LBA range [29] may still incur varying write latency. Therefore, random writes are done within a small region to ensure that a single persistent cache is used.

Figure 2.4 shows the results for this test. Emulated-SMR-1 sequentially writes incoming random writes to the persistent cache. It fills one empty block after another and due to synchronicity of the writes it misses the next empty block by the time the next write arrives. Therefore, it waits for a complete rotation resulting in a 10 ms write latency, which is the rotation time of the underly-

Test 1: Discovering Drive Type

```
1 Write blocks in the first 1 GiB in random order to the drive.
2 if latency is fixed then the drive is DM-SMR else the drive is CMR.
```

ing CMR drive. The sub-millisecond latency of Emulated-SMR-2 shows that this drive uses flash for the persistent cache. The latency of Emulated-SMR-3 is identical to that of Emulated-SMR-1, suggesting a similar setup. The varying latency of Seagate-CMR identifies it as a conventional drive. Seagate-SMR shows a fixed ≈ 25 ms latency with a ≈ 325 ms bump at the 240th write. While the fixed latency indicates that it is a DM-SMR drive, we resort to the head position graph to understand why it takes 25 ms to write a single block and what causes the 325 ms latency.

Figure 2.5 shows that the head, initially parked at the ID, seeks to the outer diameter (OD) for the first write. It stays there during the first 239 writes (incidentally, showing that the persistent cache is at the OD), and on the 240th write it seeks to the center, staying there for ≈ 285 ms before seeking back and continuing to write.

Is all of 25 ms latency associated with every block write spent writing or is some of it spent in rotational delay? When we repeat the test multiple times, the completion time of the first write ranges between 41 and 52 ms, while the remaining writes complete in 25 ms. The latency of the first write always consists of a seek from the ID to the OD (≈ 16 ms). We hypothesize that the remaining time is spent in rotational delay—likely waiting for the beginning of a delimited location—and writing (25 ms). Depending on where the head lands after the seek, the latency of the first write changes between 41 ms and 52 ms. The remaining writes are written as they arrive, without seek time and rotational delay, each taking 25 ms. Hence, we hypothesize that a single block *host write* results in a 2.5 track *internal write*. We realize that 25 ms latency is artificially high and expect it to drop in future drives, nevertheless, we base our further explanations on this assumption. In the following section we explore this phenomenon further.

Journal Entries with Quantized Sizes

If after Test 1 we immediately read blocks in the written order, read latency is fixed at ≈ 5 ms, indicating 0.5 track distance (covering a complete track takes a full rotation, which is 10 ms for the drive; therefore 5 ms translates to 0.5 track distance) between blocks. On the other hand, if we write blocks asynchronously at the maximum queue depth of 31 [117] and immediately read them, latency is fixed at ≈ 10 ms, indicating a missed rotation due to contiguous placement. Furthermore, although the drive still reports 25 ms completion time for every write, asynchronous writes complete faster—for the 256 write operations, asynchronous writes complete in 216 ms whereas synchronous writes complete in 6,539 ms, as seen in Figure 2.5. Gathering these facts, we arrive at Figure 2.6. Writing asynchronously with high queue depth allows the drive to pack multiple blocks into a single internal write, placing them contiguously (shown on the right). The drive reports the completion of individual host writes packed into the same internal write once the internal write completes. Thus, although each of the host writes in the same internal write is reported to take 25 ms, it is the same 25 ms that went into writing the internal write. As a result, in the asynchronous case, the drive does fewer internal writes, which accounts for the fast completion time. The contiguous placement also explains the 10 ms latency when reading blocks in the

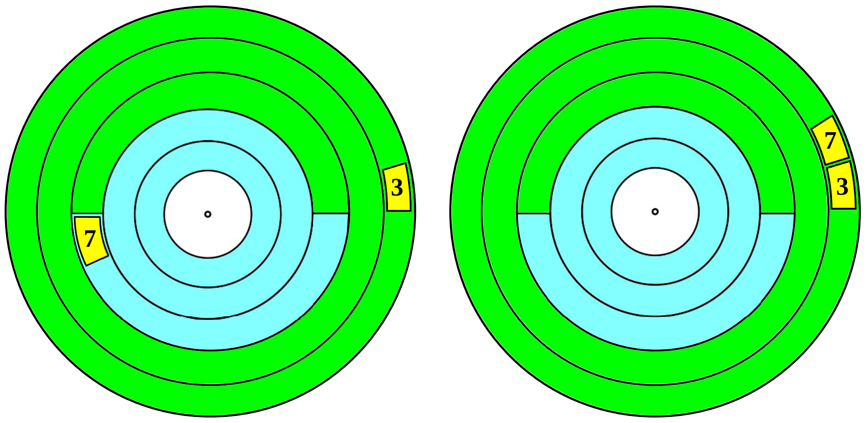


Figure 2.6: Surface of a disk platter in a hypothetical DM-SMR drive divided into two 2.5 track imaginary regions. The left figure shows the placement of random blocks 3 and 7 when writing synchronously. Each internal write contains a single block and takes 25 ms (50 ms in total) to complete. The drive reports 25 ms write latency for each block; reading the blocks in the written order results in a 5 ms latency. The right figure shows the placement of blocks when writing asynchronously with high queue depth. A single internal write contains both of the blocks, taking 25 ms to complete. The drive still reports 25 ms write latency for each block; reading the blocks back in the written order results in a 10 ms latency due to missed rotation.

written order. Writing synchronously, however, results in doing a separate internal write for every block (shown on the left), taking longer to complete. Placing blocks starting at the beginning of 2.5 track internal writes explains the 5 ms latency when reading blocks in the written order.

To understand how the internal write size changes with the increasing host write size, we keep writing at the maximum queue depth, gradually increasing the write size. Figure 2.7 shows that the writes in the range of 4 KiB–26 KiB result in 25 ms latency, suggesting that 31 host writes in this size range fit in a single internal write. As we jump to the 28 KiB writes, the latency increases by ≈ 5 ms (or 0.5 track) and remains approximately constant for the writes of sizes up to 54 KiB. We observe a similar jump in latency as we cross from 54 KiB to 56 KiB and also from 82 KiB to 84 KiB. This shows that the internal write size increases in 0.5 track increments. Given that the persistent cache is written using a "log-structured journaling mechanism" [67], we infer that the 0.5 track of 2.5 track minimum internal write is the journal entry that grows in 0.5 track increments, and the remaining 2 tracks contain out-of-band data, like parts of the persistent cache map affected by the host writes. The purpose of this quantization of journal entries is not known, but may be in order to reduce rotational delay or simplify delimiting and locating them. We further hypothesize that the 325 ms delay in Figure 2.4, observed every 240th write, is a map merge operation that stores the updated map at the middle tracks.

As the write size increases to 256 KiB we see varying delays, and inspection of completion times shows less than 31 writes completing in each burst, implying a bound on the journal entry size. Different completion times for large writes suggest that for these, the journal entry size is determined dynamically, likely based on the available drive resources at the time when the journal entry is formed.

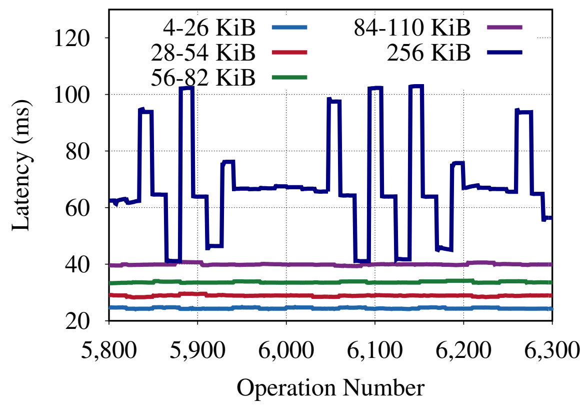


Figure 2.7: Random write latency of different write sizes on Seagate-SMR, when writing at the queue depth of 31. Each latency graph corresponds to the latency of a group of writes. For example, the graph at 25 ms corresponds to the latency of writes with sizes in the range of 4–26 KiB. Since writes with different sizes in a range produced similar latency, we plotted a single latency as a representative.

2.4.4 Disk Cache Location and Layout

We next determine the location and layout of the disk cache, exploiting a phenomenon called *fragmented reads* [29]. When sequentially reading a region in a DM-SMR drive, if the cache contains newer version of some of the blocks in the region, the head has to seek to the persistent cache and back, physically fragmenting a logically sequential read. In Test 2, we use these variations in seek time to discover the location and layout of the disk cache.

Test 2: Discovering Disk Cache Location and Layout

```
1 Starting at a given offset, write a block and skip a block, and so on, writing 512 blocks in total.
2 Starting at the same offset, read 1,024 blocks; call average latency $\text{lat}_{offset}$.
3 Repeat steps 1 and 2 at the offsets high, low, mid.
4 if $\text{lat}_{high} < \text{lat}_{mid} < \text{lat}_{low}$ then
      There is a single disk cache at the ID.
  else if $\text{lat}_{high} > \text{lat}_{mid} > \text{lat}_{low}$ then
      There is a single disk cache at the OD.
  else if $\text{lat}_{high} = \text{lat}_{mid} = \text{lat}_{low}$ then
      There are multiple disk caches.
  else
      assert($\text{lat}_{high} = \text{lat}_{low}$ and $\text{lat}_{high} > \text{lat}_{mid}$)
      There is a single disk cache in the middle.
```

The test works by choosing a small region and writing every other block in it and then reading the region sequentially from the beginning, forcing a fragmented read. LBA numbering conventionally starts at the OD and grows towards the ID. Therefore, a fragmented read at low LBAs on a drive with the disk cache located at the OD would incur negligible seek time, whereas a fragmented read at high LBAs on the same drive would incur high seek time. Conversely, on a drive

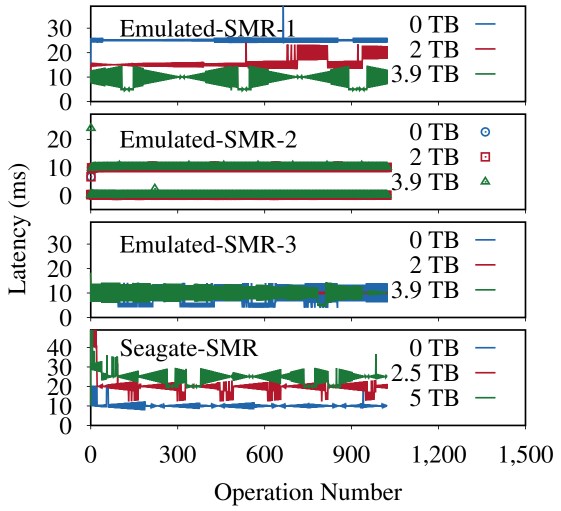


Figure 2.8: Discovering disk cache structure and location using fragmented reads.

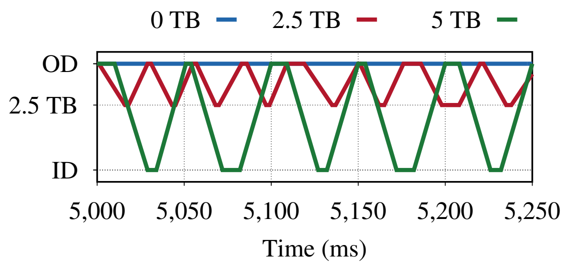


Figure 2.9: Seagate-SMR head position during fragmented reads.

with the disk cache located at the ID, a fragmented read would incur high seek time at low LBAs and negligible seek time at high LBAs. On a drive with the disk cache located at the middle diameter (MD), fragmented reads at low and high LBAs would incur similar high seek times and they would incur negligible seek times at middle LBAs. Finally, on a drive with multiple disk caches evenly distributed across the drive, the fragmented read latency would be mostly due to rotational delay and vary little across the LBA space. Guided by these assumptions, to identify the location of the disk cache, the test chooses a small region at low, middle, and high LBAs and forces fragmented reads at these regions.

Figure 2.8 shows the latency of fragmented reads at three offsets on all DM-SMR drives. The test correctly identifies the Emulated-SMR-1 as having a single cache at the ID. For Emulated-SMR-2 with flash cache, latency is seen to be negligible for flash reads, and a full missed rotation for each disk read. Emulated-SMR-3 is also correctly identified as having multiple disk caches—the latency graph of all fragmented reads overlap, all having the same 10 ms average latency. For Seagate-SMR (test performed with volatile cache enabled with `hdparm -W1`) we confirm that it has a single disk cache at the OD.

Figure 2.9 shows the Seagate-SMR head position during fragmented reads at offsets of 0 TB, 2.5 TB and 5 TB. For the offsets of 2.5 TB and 5 TB, we see that the head seeks back and forth between the OD and near-center and between the OD and the ID, respectively, occasionally missing a rotation. The cache-to-data distance for the LBAs near 0 TB was too small for the resolution of our camera.

2.4.5 Cleaning Algorithm

The fragmented read effect is also used in Test 3 to determine whether the drive uses aggressive or lazy cleaning, by creating a fragmented region and then pausing to allow an aggressive cleaning to run before reading the region.

Test 3: Discovering Cleaning Type

```
1 Starting at a given offset, write a block and skip a block and so on, writing 512 blocks in total.
2 Pause for 3–5 seconds.
3 Starting at the same offset, read 1024 blocks.
4 if latency is fixed then cleaning is aggressive else cleaning is lazy.
```

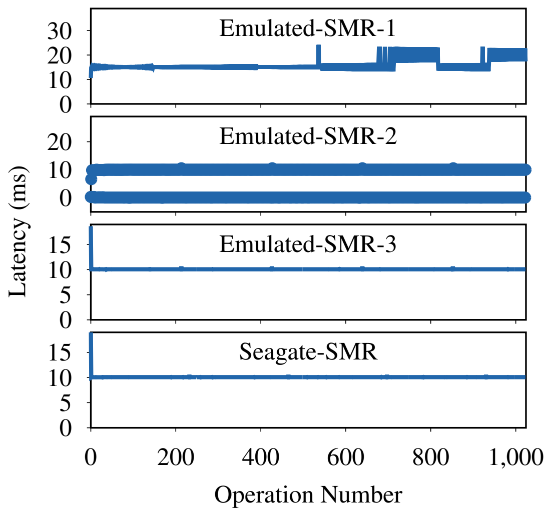


Figure 2.10: Discovering the type of cleaning using Test 3.

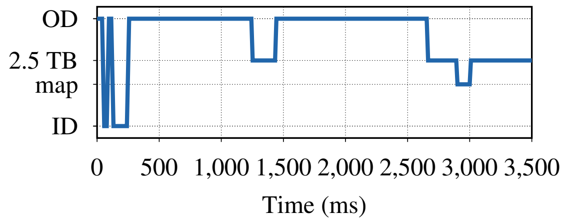


Figure 2.11: Seagate-SMR head position during pause in step 2 of Test Test 3.

Figure 2.10 shows the read latency graph of step 3 from Test 3 at the offset of 2.5 TB, with a three second pause in step 2. For all drives, offsets were chosen to land within a single band (§ 2.4.8). After a pause the top two emulated drives continue to show fragmented read behavior, indicating lazy cleaning, while in Emulated-SMR-3 and Seagate-SMR reads are no longer fragmented, indicating aggressive cleaning.

Figure 2.11 shows the Seagate-SMR head position during the 3.5 second period starting at the beginning of step 2. Two short seeks from the OD to the ID and back are seen in the first 200 ms; their purpose is not known. The RMW operation for cleaning a band starts at 1,242 ms after the last write, when the head seeks to the band at 2.5 TB offset, reads for 180 ms and seeks back to the cache at the OD where it spends 1,210 ms. We believe this time is spent forming an updated band and persisting it to the disk cache, to protect against power failure during band overwrite. Next, the head seeks to the band, taking 227 ms to overwrite it and then seeks to the center to update the map. Hence, cleaning a band in this case took $\approx$ 1.6 s. We believe the center to contain the map because the head always moves to this position after performing a RMW, and stays there for a short period before eventually parking at the ID. After 3 seconds, reads begin and the head seeks back to the band location, where it stays until reads complete (only the first 500 ms is seen in Figure 2.11).

We confirmed that the operation starting at 1,242 ms is indeed an RMW: when step 3 is begun before the entire cleaning sequence has completed, read behavior is unchanged from Test 2. We

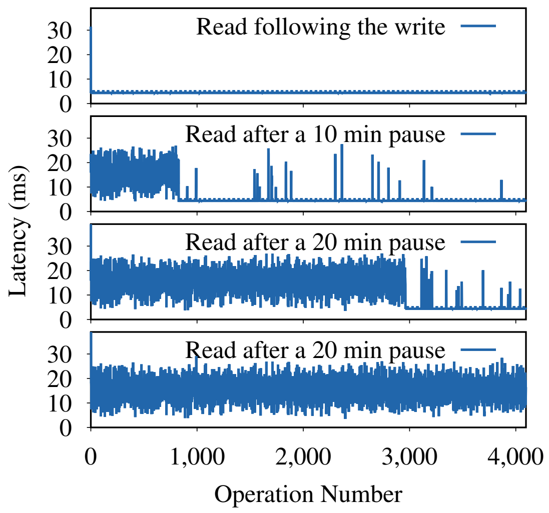


Figure 2.12: Latency of reads of random writes immediately after the writes and after pauses.

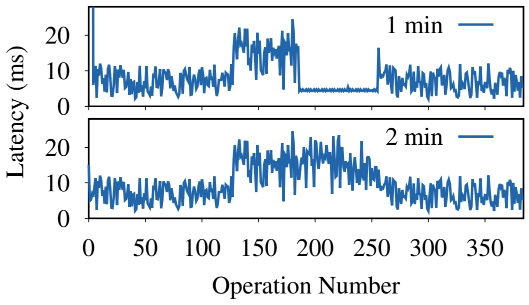


Figure 2.13: Verifying hypothesized cleaning algorithm on Seagate-SMR.

did not explore the details of the RMW; alternatives like *partial read-modify-write* [151] may also have been used.

Seagate-SMR Cleaning Algorithm

We next start exploring performance-relevant details that are specific to the Seagate-SMR cleaning algorithm, by running Test 4. In step 1, as the drive receives random writes, it sequentially logs them to the persistent cache as they arrive. Therefore, immediately reading the blocks back in the written order should result in a fixed rotational delay with no seek time. During the pause in step 3, cleaning process moves the blocks from the persistent cache to their native locations. As a result, reading after the pause should incur varying seek time and rotational delay for the blocks moved by the cleaning process, whereas unmoved blocks should still incur a fixed latency.

Test 4: Exploring Cleaning Algorithm

```
1  Write 4,096 random blocks.
2  Read back the blocks in the written order.
3  Pause for 10–20 minutes.
4  Repeat steps 2 and 3.
```

In Figure 2.12 read latency is shown immediately after step 2, and then after 10, 30, and 50 minutes. We observe that the latency is fixed when we read the blocks immediately after the writes. If we re-read the blocks after a 10-minute pause, we observe random latencies for the first ≈ 800 blocks, indicating that the cleaning process has moved these blocks to their native locations. Since every block is expected to be on a different band, the number of operations with random read latencies after each pause shows the progress of the cleaning process, that is, the number of bands it has cleaned. Given that it takes ≈ 30 minutes to clean ≈ 3,000 bands, it takes ≈ 600 ms to clean a band whose single block has been overwritten. We also observe a growing number of cleaned blocks in the unprocessed region (for example, operations 3,000–4,000 in the 30 minute graph); based on this behavior, we hypothesize that cleaning follows Algorithm 1.

Algorithm 1: Hypothesized Cleaning Algorithm of Seagate-SMR

```
1 Read the next block from the persistent cache, find the block's band.
2 Scan the persistent cache identifying blocks belonging to the band.
3 Read-modify-write the band, update the map.
```

To test this hypothesis we run Test 5. In Figure 2.13 we see that after one minute, all of the blocks written in step 1, some of those written in step 2, and all of those written in step 3 have been cleaned, as indicated by the non-uniform latency, while the remainder of step 2 blocks remain in the cache, confirming our hypothesis. After two minutes all blocks have been cleaned. (The higher latency for step 2 blocks is due to their higher mean seek distance.)

Test 5: Verifying the Hypothesized Cleaning Algorithm

```
1 Write 128 blocks from a 256 MiB linear region in random order.
2 Write 128 random blocks across the LBA space.
3 Repeat step 1, using different blocks.
4 Pause for one minute; read all blocks in the written order.
```

2.4.6 Persistent Cache Size

We discover the size of the persistent cache by ensuring that the cache is empty and then measuring how much data may be written before cleaning begins. We use random writes across the LBA space to fill the cache, because sequential writes may fill the drive bypassing the cache [29] and cleaning may never start. Also, with sequential writes, a drive with multiple caches may fill only one of the caches and start cleaning before all of the caches are full [29]. With random writes, bypassing the cache is not possible; also, they will fill multiple caches at the same rate and cleaning will start when all of the caches are almost full.

The simple task of filling the cache is complicated in drives using extent mapping: a cache is considered full when the extent map is full or when the disk cache is full, whichever happens first. The latter is further complicated by journal entries with quantized sizes—as seen previously (§ 2.4.3), a single 4 KB write may consume as much cache space as dozens of 8 KB writes. Due to this overhead, actual size of the disk cache is larger than what is available to host writes—we differentiate the two by calling them *persistent cache **raw** size* and *persistent cache size*, respectively.

Figure 2.14 shows three possible scenarios on a hypothetical drive with a persistent cache raw size of 36 blocks and a 12 entry extent map. The minimum journal entry size is 2 blocks, and it grows in units of 2 blocks to the maximum of 16 blocks; out-of-band data of 2 blocks is written with every journal entry; the persistent cache size is 32 blocks.

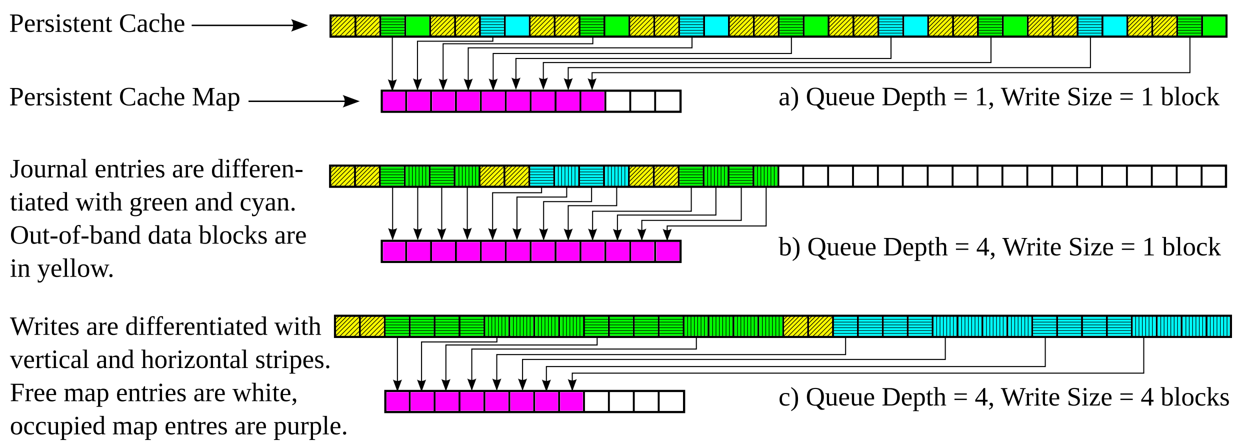

Persistent Cache
Persistent Cache Map
a) Queue Depth = 1, Write Size = 1 block
Journal entries are differentiated with green and cyan. Out-of-band data blocks are in yellow.
b) Queue Depth = 4, Write Size = 1 block
Writes are differentiated with vertical and horizontal stripes. Free map entries are white, occupied map entres are purple.
c) Queue Depth = 4, Write Size = 4 blocks

Figure 2.14: Three different scenarios triggering cleaning on drives using journal entries with quantized sizes and extent mapping. The text on the left in the figure explains the meaning of the colors.

Figure 2.14 (a) shows the case of queue depth 1 and 1-block writes. After the host issues 9 writes, the drive puts every write to a separate 2-block journal entry, fills the cache with 9 journal entries and starts cleaning. Every write consumes a slot in the map, shown by the arrows. Due to low queue depth, the drive leaves one empty block in each journal entry, wasting 9 blocks. Exploiting this behavior, Test 6 discovers the persistent cache raw size. (In this and the following tests, we detect the start of cleaning when the IOPS drops to near zero.)

Test 6: Discovering Persistent Cache Raw Size

```
1 Write with a small size and low queue depth until cleaning starts.
2 Persistent cache raw size = number of writes × (min. journal entry size + out-of-band data size).
```

Figure 2.14 (b) shows the case of queue depth 4 and 1-block writes. After the host issues 12 writes, the drive forms three 4-block journal entries. Writing these journal entries to the cache fills the map and the drive starts cleaning despite a half-empty cache. We use Test 7 to discover the *persistent cache map size*.

Test 7: Discovering Persistent Cache Map Size

```
1 Write with a small size and high queue depth until cleaning starts.
2 Persistent cache map size = number of writes.
```

Finally, Figure 2.14 (c) shows the case of queue depth 4 and 4-block writes. After the host issues 8 writes, the drive forms two 16-block journal entries, filling the cache. Due to high queue depth and large write size, the drive is able to fill the cache (without wasting any blocks) before the map fills. We use Test 8 to discover the *persistent cache size*.

Test 8: Discovering Persistent Cache Size

```
1 Write with a large size and high queue depth until cleaning starts.
2 Persistent cache size = total host write size.
```

Drive	Write Size	QD	Operation Count	Host Writes	Internal Writes
Seagate-SMR	4 KiB	1	22,800	89 MiB	**100 GiB**[a]
	4 KiB	31	**182,270**	0.7 GiB	N/A
	64 KiB	31	**182,231**	11.12 GiB	N/A
	128 KiB	31	137,496	**16.78 GiB**	N/A
	256 KiB	31	67,830	**16.56 GiB**	N/A
Emulated-SMR-1	4 KiB	1	9,175,056	35 GiB	35 GiB
Emulated-SMR-2	4 KiB	1	2,464,153	9.4 GiB	9.4 GiB
Emulated-SMR-3	4 KiB	1	9,175,056	35 GiB	35 GiB

Table 2.2: Discovering persistent cache parameters. [a]This estimate is based on the hypothesis that all of 25 ms during a single block write is spent writing to disk. While the results of the experiments indicate this to be the case, we think 25 ms latency is artificially high and expect it to drop in future drives, which would require recalculation of this estimate.

Table 2.2 shows the result of the tests on Seagate-SMR and Figure 2.15 shows the corresponding graph. In the first row of the table, we discover persistent cache raw size using Test 6. Writing with 4 KiB size and queue depth of 1 produces a fixed 25 ms latency (§ 2.4.3), that is 2.5 rotations. Hypothesizing that all of the 25 ms is spent writing and a track size is ≈ 2 MiB at the OD, 22,800 operations correspond to ≈ 100 GiB.

In rows 2 and 3 we discover the persistent cache map size using Test 7. For write sizes of 4 KiB and 64 KiB cleaning starts after ≈ 182,200 writes, which corresponds to 0.7 GiB and 11.12 GiB of host writes, respectively. This confirms that in both cases the drive hits the map size limit, corresponding to scenario (b) in Figure 14. Assuming that the drive uses a low watermark to trigger cleaning, we estimate that the map size is 200,000 entries.

In rows 4 and 5 we discover the persistent cache size using Test 8. With 128 KiB writes we write ≈ 17 GiB in fewer operations than in row 3, indicating that we are hitting the size limit. To confirm this, we increase write size to 256 KiB in row 5; as expected, the number of operations drops by half while the total write size stays the same. Again, assuming that the drive has hit the low watermark, we estimate that the persistent cache size is 20 GiB.

Journal entries with quantized sizes and extent mapping are absent topics in academic literature on SMR, so emulated drives implement neither feature. Running Test 6 on emulated drives produces all three answers, since in these drives, the cache is block-mapped, and the cache size and cache raw size are the same. Furthermore, set-associative STL divides the persistent cache into cache bands and assigns data bands to them using modulo arithmetic. Therefore, despite having a single cache, under random writes it behaves similarly to a fully-associative cache. The bottom rows of Table 2.2 show that in emulated drives, Test 8 discovers the cache size (see Table 2.1) with 95% accuracy.

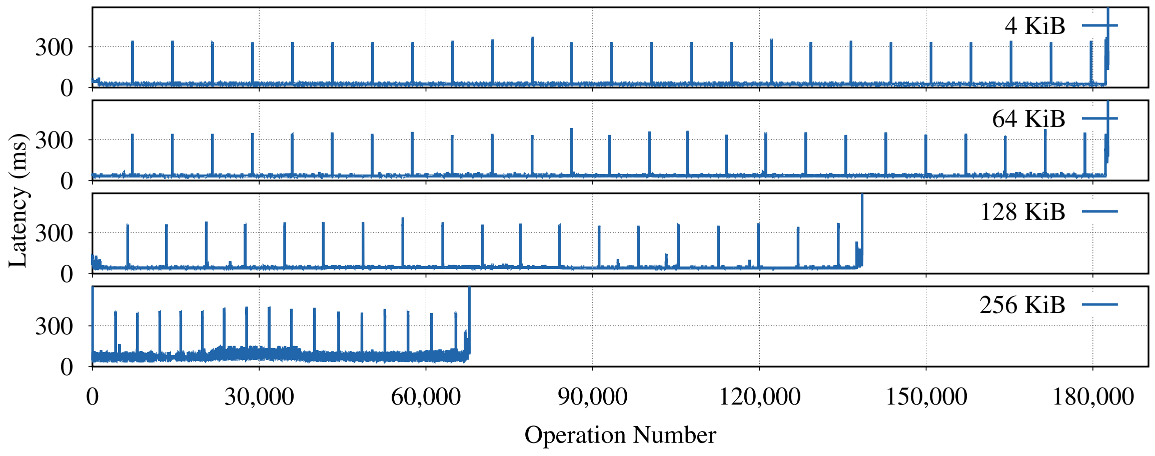


Figure 2.15: Write latency of asynchronous writes of varying sizes with queue depth of 31 until cleaning starts. Starting from the top, the graphs correspond to the lines 2-5 in Table 2.2. When writing asynchronously, more writes are packed into the same journal entry. Therefore, although the map merge operations still occur at every 240th journal write, the interval seems greater than in Figure 2.16. For 4 KiB and 64 KiB write sizes, we hit the map size limit first, hence cleaning starts after the same number of operations. For 128 KiB write size we hit the space limit before hitting the map size limit; therefore, cleaning starts after fewer number of operations than in 64 KiB writes. Doubling the write size to 256 KiB confirms that we are hitting the space limit, since cleaning starts after half the number of operations of 128 KiB writes.

2.4.7 Is Persistent Cache Shingled?

We next determine whether the STL manages the persistent cache as a circular log [1]. While this would not guarantee that the persistent cache is shingled (an STL could also manage a randomly writable region as a circular log), it would strongly indicate that the persistent cache is shingled. We start with Test 9, which chooses a sequence of 10,000 random LBAs across the drive space and writes the sequence straight through, three times. Given that the persistent cache has space for ≈ 23,000 synchronous block writes (Table 2.2), a trivial STL would fill the cache and start cleaning before the writes complete.

Test 9: Discovering if the Persistent Cache is a Circular Log—Part I

```
1 Choose 10,000 random blocks across the LBA space.
2 for $i \leftarrow 0$ to $i < 3$ do
3     Write the 10,000 blocks from step 1 in the chosen order.
```

Figure 2.17 shows that unlike Figure 2.16, cleaning does not start after 23,000 writes. Two simple hypotheses that explain this phenomenon are:

1. The STL manages the persistent cache as a circular log. When the head of the log wraps around, STL detects stale blocks and overwrites without cleaning.

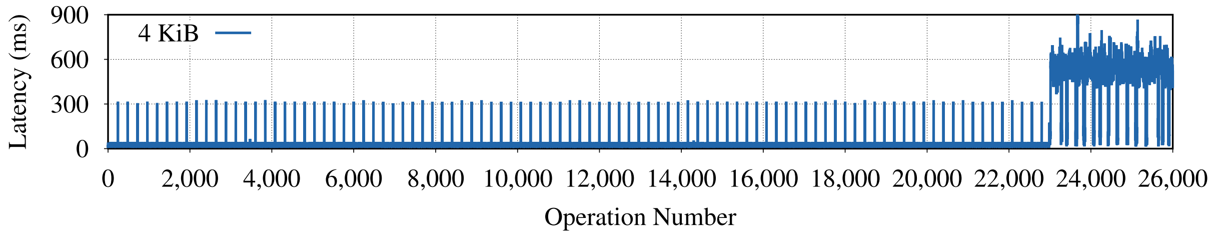


Figure 2.16: Write latency of 4 KiB synchronous random writes, corresponding to the first line in Table 2.2. As explained in § 2.4.3, when writing synchronously the drive writes a journal entry for every write operation. Every 240th journal entry write results in a ≈ 325 ms latency, which as was hypothesized in § 2.4.3 includes a map merge operation. After ≈ 23,000 writes, cleaning starts and the IOPS drops precipitously to 0–3. To emphasize the high latency of writes during cleaning we perform 3,000 more operations. As the graph shows, these writes (23,000–26,000) have ≈ 500 ms latency.

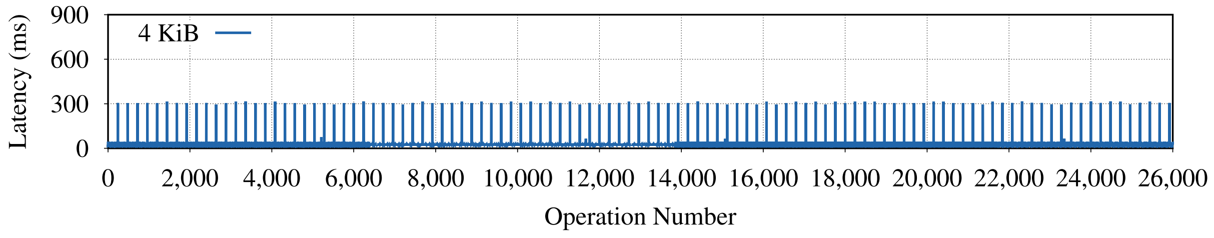


Figure 2.17: Write latency of 30,000 random block writes with a repeating pattern. We choose 10,000 random blocks across the LBA space and write them in the chosen order. We then write the same 10,000 blocks in the same order two more times. Unlike Figure 2.16, the cleaning does not start after ≈ 23,000 writes, because due to the repeating pattern, as the head of the log wraps around, the STL only finds stale blocks that it can overwrite without cleaning.

2. The STL overwrites blocks in-place. Since there are 10,000 unique blocks, we never fill the persistent cache and cleaning never starts.

To find out which one of these is true, we run Test 10. Since there are still 10,000 unique blocks, if the hypothesis (2) is true, that is if the STL overwrites the blocks in-place, we should never consume more than 10,000 writes' worth of space and cleaning should not start before the writes complete. Figure 2.18 shows that cleaning starts after ≈ 23,000 writes, invalidating hypothesis (2). Furthermore, if we compare Figure 2.18 to Figure 2.16, we see that the latency of writes after cleaning starts is ≈ 100 ms and ≈ 500 ms, respectively. This corroborates hypothesis (1)—latency is lower in the former, because after the head of the log wraps around, the STL finds some stale blocks (since these blocks were chosen from a small pool of 10,000 unique blocks), that it can overwrite without cleaning. When the blocks are chosen across the LBA space, as in Figure 2.16, once the head wraps around, the STL ends up effectively cleaning before every write since it almost never finds a stale block.

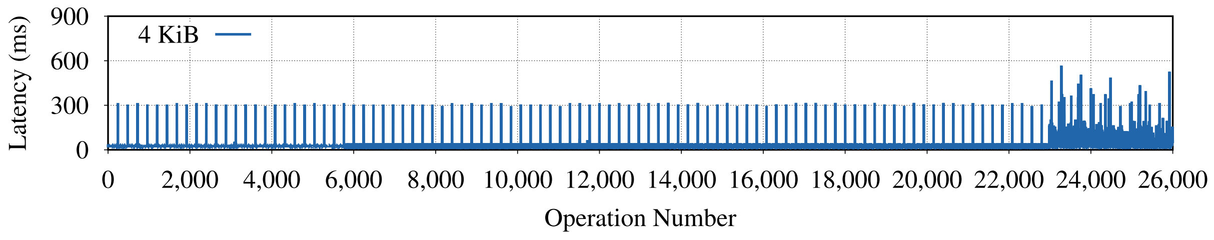


Figure 2.18: Write latency of 30,000 random block writes chosen from a pool of 10,000 unique blocks. Unlike Figure 2.17, cleaning starts after ≈ 23,000, because as the head of the log wraps around, the STL does not immediately find stale blocks. However, since the blocks are chosen from a small pool, the STL still does find a large number of stale blocks and can often overwrite without cleaning. Therefore, compared to Figure 2.16 the write latency during cleaning (operations 23,000-26,000) is not as high, since in Figure 2.16 the blocks are chosen across the LBA space and the STL almost never finds a stale block when the head of the log wraps around.

Test 10: Discovering if the Persistent Cache is a Circular Log—Part II

```
1 Choose 10,000 random blocks across the LBA space.
2 for $i \leftarrow 0$ to $i < 30{,}000$ do
3     Randomly choose a block from the blocks in step 1 and write.
```

2.4.8 Band Size

STLs proposed to date [9, 29][79] clean a single band at a time, by reading unmodified data from a band and updates from the cache, merging them, and writing the merge result back to a band. Test 11 determines the band size, by measuring the granularity at which this cleaning process occurs.

Test 11: Discovering the Band Size

```
1 Select an accuracy granularity $a$, and a band size estimate $b$.
2 Choose a linear region of size $100 \times b$ and divide it into $a$-sized blocks.
3 Write 4 KiB to the beginning of every $a$-sized block, in random order.
4 Force cleaning to run for a few seconds and read 4 KiB from the beginning of every $a$-sized block
    in sequential order.
5 Consecutive reads with identical high latency identify a cleaned band.
```

Assuming that the linear region chosen in Test 11 lies within a region of equal track length, for data that is not in the persistent cache, 4 KB reads at a fixed stride a should see identical latencies—that is, a rotational delay equivalent to $(a \bmod T)$ bytes where T is the track length. Conversely reads of data from cache will see varying delays in the case of a disk cache due to the different (and random) order in which they were written or sub-millisecond delays in the case of a flash cache.

With aggressive cleaning, after pausing to allow the disk to clean a few bands, a linear read of the written blocks will identify the bands that have been cleaned. For a drive with lazy cleaning the

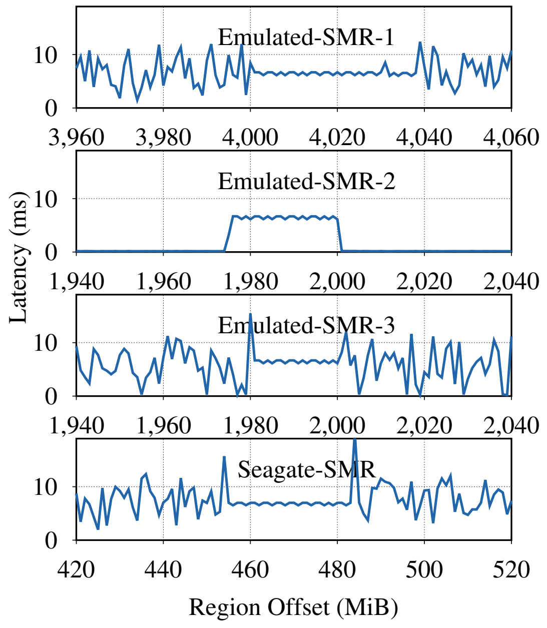


Figure 2.19: Discovering the band size. The flat latency regions correspond to sequentially reading a complete band from its native location.

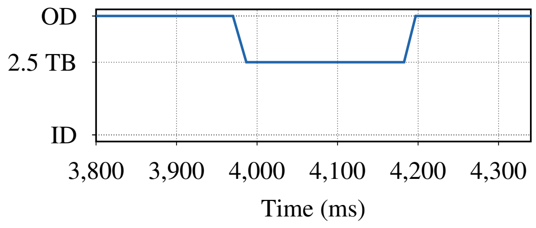


Figure 2.20: Head position during the sequential read for Seagate-SMR, corresponding to the time period in Figure 2.19.

linear region is chosen so that writes fill the persistent cache and force a few bands to be cleaned, which again may be detected by a linear read of the written data.

In Figure 2.19 we see the results of Test 11 for $a = 1$ MiB and $b = 50$ MiB, respectively, with the region located at the 2.5 TB offset; for each drive we zoom in to show an individual band that has been cleaned. We correctly identify the band size for the emulated drives (see Table 2.1). The band size of Seagate-SMR at this location is seen to be 30 MiB; running tests at different offsets shows that bands are iso-capacity within a zone (§ 2.4.12) but vary from 36 MiB at the OD to 17 MiB at the ID.

Figure 2.20 shows the head position of Seagate-SMR corresponding to the time period in Figure 2.19. It shows that the head remains at the OD during the reads from the persistent cache up to 454 MiB, then seeks to 2.5 TB offset and stays there for 30 MiB, and then seeks back to the cache at OD, confirming that the blocks in the band are read from their native locations.

2.4.9 Cleaning Time of a Single Band

We observed that cleaning a band whose single block was overwritten can take ≈ 600 ms whereas if we overwrite 2 MiB of the band by skipping every other block, cleaning time increases to ≈ 1.6 s (§ 2.4.5). While ≈ 600 ms cleaning time due to a single block overwrite gives us a lower bound on the cleaning time, we do not know the upper bound. Now that we understand the persistent cache structure and band size, in addition to the cleaning algorithm, we create an adversarial workload that will give us an upper bound for the cleaning time of a single band.

Table 2.2 shows that with a queue depth of 31, we can write 182,270 blocks, that is 5,880 journal entries, resulting in 700 MiB host writes. Assuming the band size is 35 MiB at the OD, 700 MiB corresponds to 20 bands. Therefore, if we distribute (through random writes) the blocks of 20 bands among 5,880 journal entries, the drive will need to read every packet to clean a single band. Assuming 5–10 ms read time for a packet, reading all of the packets to assemble the band will take 29–60 s. To confirm this hypothesis, we shuffled the first 700 MiB worth of blocks and wrote them with a queue depth of 31. The cleaning took ≈ 15 minutes, which is ≈ 45 s per band.

2.4.10 Block Mapping

Once we discover the band size (§ 2.4.8), we can use Test 12 to determine the mapping type. This test exploits varying inter-track switching latency between different track pairs to detect if a band was remapped. After overwriting the first two tracks of band *b*, cleaning will move the band to its new location—a different physical location only if dynamic mapping is used. Plotting latency graphs of step 2 and step 4 will produce the same pattern for the static mapping and a different pattern for the dynamic mapping.

Test 12: Discovering mapping type.

```
1 Choose two adjacent iso-capacity bands $a$ and $b$; set $n$ to the number of blocks in a track.
2 for $i \leftarrow 0$ to $i < 2$ do
      for $j \leftarrow 0$ to $j < n$ do
          Read block $j$ of track 0 of band $a$
          Read block $j$ of track $i$ of band $b$
3 Overwrite the first two tracks of band $b$; force cleaning to run.
4 Repeat step 2.
```

Adapting this test to a drive with lazy cleaning involves some extra work. First, we should start the test on a drive after a secure erase, so that the persistent cache is empty. Due to lazy cleaning, the graph of step 4 will be the graph of switching between a track and the persistent cache. Therefore, we will fill the cache until cleaning starts, and repeat step 2 once in a while, comparing its graph to the previous two: if it is similar to the last, then data is still in the cache, if it is similar to the first, then the drive uses static mapping, otherwise, the drive uses dynamic mapping.

We used track and block terms to concisely describe the test above, but the size chosen for these parameters of the test need not match track size and block size of the underlying drive. Figure 2.21, for example, shows the plots for the test on all of the drives using 2 MiB for the track size and 16 KiB for the block size. The latency pattern before and after cleaning is different only for Emulated-SMR-3 (seen on the top right), correctly indicating that it uses dynamic mapping. For all of the remaining drives, including Seagate-SMR, the latency pattern is the same before and after cleaning, indicating a static mapping.

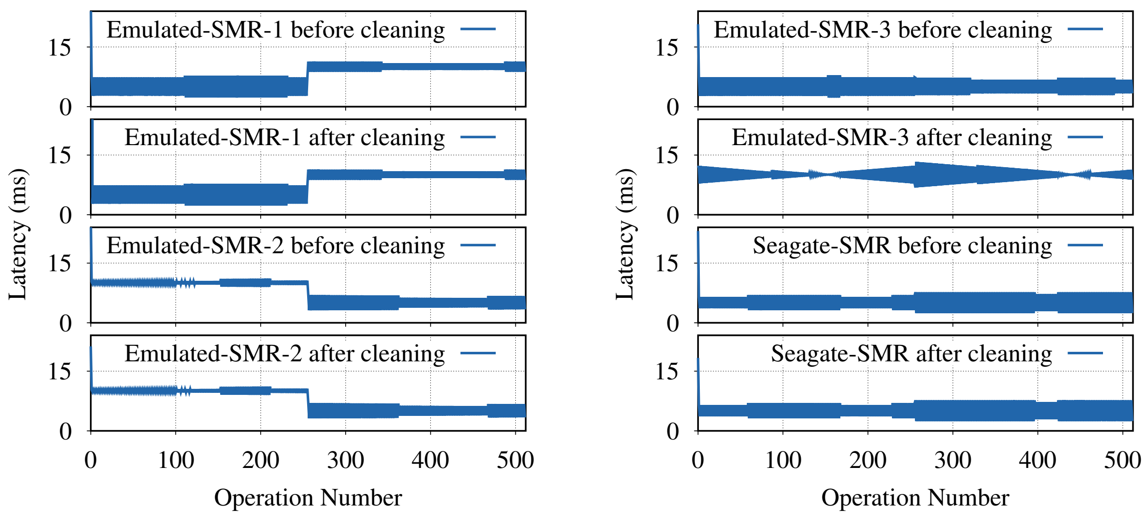


Figure 2.21: Detecting mapping type.

2.4.11 Effect of Mapping Type on Drive Reliability

The type of band mapping used in a DM-SMR drive affects the drive reliability for the reasons explained next. Figure 2.22 shows sequential read throughput on Seagate-SMR. We get a similar graph for sequential writes when we enable the volatile cache, which suggests that the drive sustains full throughput for sequential writes. Seagate-SMR does not contain flash and it uses static mapping, therefore it can achieve full throughput only if it buffers the data in the volatile cache and writes directly to the band, bypassing the persistent cache.

This performance improvement, however, comes with a risk of data loss. Since there is no backup of the overwritten data, if power is lost midway through the band overwrite, blocks in the following tracks are left in a corrupt state, resulting in data loss. We also lose the new data since it was buffered in the volatile cache.

A similar error, known as *torn write* [15, 109], occurs in CMR drives as well, wherein only a portion of a sector gets written before power is lost. In CMR drives, the time required to atomically overwrite a sector is small enough that reports of such errors are rare [109]. A DM-SMR drive with static mapping, on the other hand, is similar to a CMR drive with large (in case of Seagate-SMR, 17–36 MiB) sectors. Therefore, there is a high probability that a random power loss during streaming sequential writes will disrupt a band overwrite.

Figure 2.23 describes two error scenarios in a hypothetical DM-SMR drive that uses static mapping. These errors are the consequence of the used mapping scheme, since the only way of sustaining full throughput in such a scheme is to write to the band directly. Introducing small amount of flash to a DM-SMR drive for persistent buffering has its own challenges—exploiting parallelism for fast flash writes and managing wear-leveling is possible only if large amounts of flash is used, which is not feasible inside a DM-SMR drive. On the other hand, when using a dynamic band mapping scheme, similar to fully-associative STL, a drive can write the new contents of a band directly to a free band without jeopardizing the existing band data. This, followed by an atomic switch in the mapping table would result in full-throughput sequential writes without

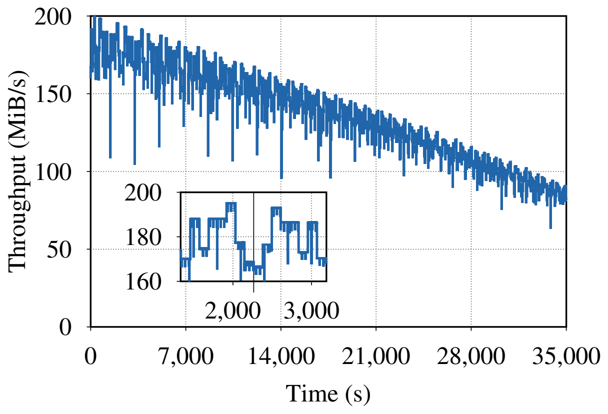


Figure 2.22: Sequential read throughput of Seagate-SMR.

sacrificing reliability. The idea is similar to log-block FTLs [106, 142] that have been successful in overcoming slow block overwrites in NAND flash. For the reasons described, we expect that the next generation of DM-SMR drives will use dynamic band mapping.

We successfully reproduced torn writes on Seagate-SMR by using an Arduino UNO R3 board with a 2-channel relay shield to control the power to the drive. After Running Test 13 at arbitrary offsets, we could reproduce hard read errors as shown in Figure 2.24 on all of our sample drives. The offset where errors occurred differed between drives. These errors disappeared after overwriting the affected regions.

Test 13: Reproducing torn writes.

```
1 Choose an *offset*.
2 for $i \leftarrow 0$ to $i < 50$ do
3     Power on the drive and start 1 MiB sequential writes at *offset*.
4     After 10 seconds power off the drive; wait for 5 seconds and power on the drive.
5     Starting at the crash point, go back 5,000 blocks and read 6,000 blocks.
```

2.4.12 Zone Structure

We use sequential reads (Test 14) to discover the zone structure of Seagate-SMR. (As mentioned before, the notion of zone here comes from *zone bit recording* [**?**] and is unrelated to SMR zones.) While there are no such drives yet, on drives with dynamic mapping a secure erase that would restore the mapping to the default state may be necessary for this test to work. Figure 2.22 shows the zone profile of Seagate-SMR, with a zoom to the beginning.

Test 14: Discovering Zone Structure

```
1 Enable kernel read-ahead and drive look-ahead.
2 Sequentially read the whole drive in 1 MiB blocks.
```

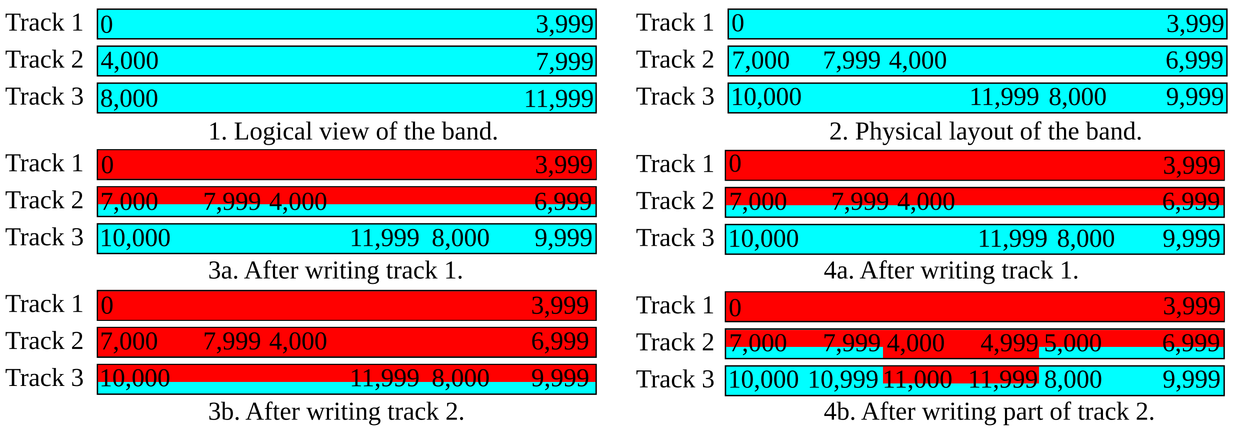

Track 1 0 3,999
Track 2 4,000 7,999
Track 3 8,000 11,999
1. Logical view of the band.
Track 1 0 3,999
Track 2 7,000 7,999 4,000 6,999
Track 3 10,000 11,999 8,000 9,999
2. Physical layout of the band.
Track 1 0 3,999
Track 2 7,000 7,999 4,000 6,999
Track 3 10,000 11,999 8,000 9,999
3a. After writing track 1.
Track 1 0 3,999
Track 2 7,000 7,999 4,000 6,999
Track 3 10,000 11,999 8,000 9,999
4a. After writing track 1.
Track 1 0 3,999
Track 2 7,000 7,999 4,000 6,999
Track 3 10,000 11,999 8,000 9,999
3b. After writing track 2.
Track 1 0 3,999
Track 2 7,000 7,999 4,000 4,999 5,000 6,999
Track 3 10,000 10,999 11,000 11,999 8,000 9,999
4b. After writing part of track 2.

Figure 2.23: Torn write scenarios in a hypothetical DM-SMR drive with bands consisting of 3 tracks and the write head with of 1.5 tracks. The tracks are shown horizontally instead of circularly to make the illustration clear. (1) shows the logical view of the band consisting of 3 tracks, each track having 4,000 sectors. (2) shows the physical layout of the tracks on a platter that accounts for the track skew. (3a) and (3b) show the corruption scenario when the power is lost during the track switch. The red region in (3a) shows the situation after track 1 of the band has been overwritten—track 1 contains new data whereas track 2 is corrupted; track 3 contains the old data. (3b) shows the situation after track 2 has been overwritten—track 1 and track 2 contain new data whereas track 3 is corrupted. If power is lost while the head switches from track 2 to track 3, block ranges 10,000–11,999 and 8,000–9,999, or the single range of 8,000–11,999 is left in a corrupt state. (4a) and (4b) show the corruption scenario when the power is lost during the track overwrite. (4a) is identical to (3a) and shows the situation after track 1 of the band has been overwritten. (4b) shows the situation where power is lost after blocks 4,000–4,999 have been overwritten. In this case, block ranges 7,000–7,999, 5,000–6,999, and 11,000–11,999, or two ranges of 5,000–7,999 and 11,000–11,999 are left in a corrupt state.

Similar to CMR drives, the throughput falls as we reach higher LBAs; unlike CMR drives, there is a pattern that repeats throughout the graph, shown by the zoomed part. This pattern has an axis of symmetry indicated by the dotted vertical line at 2,264th second. There are eight distinct plateaus to the left and to the right of the axis with similar throughputs. The fixed throughput in a single plateau and a sharp change in throughput between plateaus suggest a wide radial stroke and a head switch. Plateaus corresponds to large zones of size 18–20 GiB, gradually decreasing to 4 GiB as we approach higher LBAs. The slight decrease in throughput in symmetric plateaus on the right is due to moving from a larger to smaller radii, where sector per track count decreases; therefore, throughput decreases as well.

We confirmed these hypotheses using the head position graph shown in Figure 2.25 (a), which corresponds to the time interval of the zoomed graph of Figure 2.22. Unlike with CMR drives, where we could not observe head switches due to narrow radial strokes, with this DM-SMR drive head switches are visible to an unaided eye. Figure 2.25 (a) shows that the head starts at the OD and slowly moves towards the MD completing this inwards move at 1,457th second, indicated by the vertical dotted line. At this point, the head has just completed a wide radial stroke reading gigabytes from the top surface of the first platter, and it performs a jump back to the OD and starts

```
ata2.00: exception Emask 0x0 SAct 0x1000 SErr 0x0 action 0x0
ata2.00: irq_stat 0x40000008
ata2.00: failed command: READ FPDMA QUEUED
ata2.00: cmd 60/c0:60:40:1d:19/02:00:80:00:00/40 tag 12 ncq 360448 in
         res 41/00:c0:f8:1e:19/00:02:80:00:00/00 Emask 0x401 (device error) <F>
ata2.00: status: { DRDY ERR }
```

Figure 2.24: Hard read error under Linux kernel 3.16 when reading a region affected by a torn write.

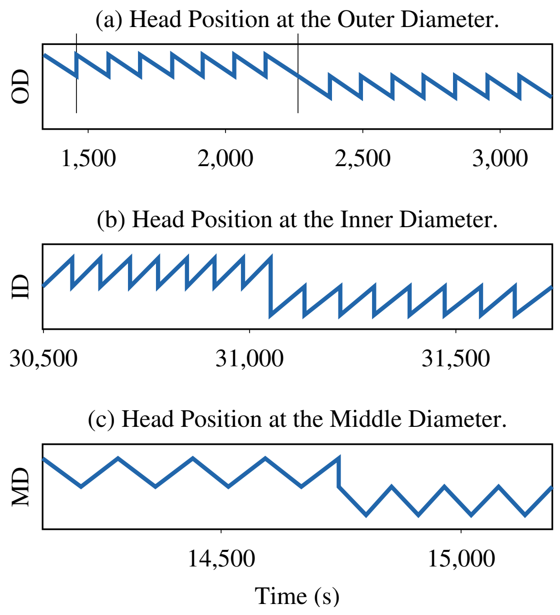


Figure 2.25: Seagate-SMR head position during sequential reads at different offsets.

a similar stroke on the bottom surface of the first platter. The direction of the head movement indicates that the shingling direction is towards the ID at the OD. The head completes the descent through the platters at 2,264th second—indicated by the vertical solid line—and starts its ascent reading surfaces in the reverse order. These wide radial strokes create "horizontal zones" that consist of thousands of tracks on the same surface, as opposed to "vertical zones" spanning multiple platters in CMR drives. We expect these horizontal zones to be the norm in DM-SMR drives, since they facilitate SMR mechanisms like allocation of iso-capacity bands, static mapping, and dynamic band size adjustment [68]. Figure 2.25 (b) corresponds to the end of Figure 2.22, shows that the direction of the head movement is reversed at the ID, indicating that both at the OD and at the ID, shingling direction is towards the middle diameter. To our surprise, Figure 2.25 (c) shows that a conventional serpentine layout with wide serpents is used at the MD. We speculate that although the whole surface is managed as if it is shingled, there is a large region in the middle that is not shingled.

It is hard to confirm the shingling direction without observing the head movement. The existence of "horizontal zones", on the other hand, can also be confirmed by contrasting the sequential latency graphs of Seagate-SMR and Seagate-CMR. Figure 2.26 (a) shows the latency graph for the zoomed region in Figure 2.22. As expected, the shape of latency graph matches the shape of the

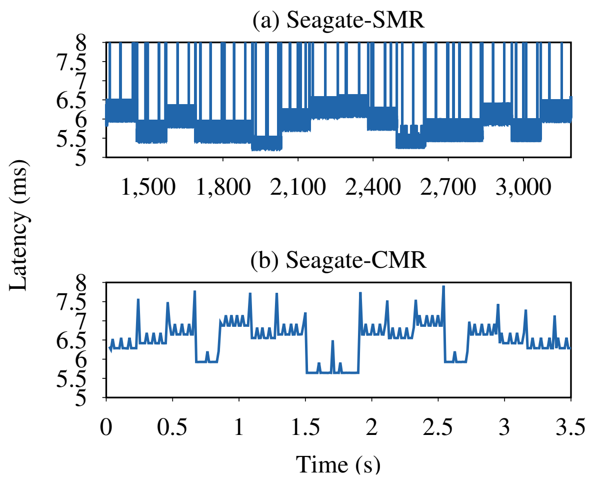


Figure 2.26: Sequential read latency of Seagate-CMR and Seagate-SMR corresponding to a complete cycle of ascent and descent through platter surfaces. While Seagate-CMR completes the cycle in 3.5 seconds, Seagate-SMR completes it in 1,800 seconds, since the latter reads thousands of tracks from a single surface before switching to the next surface.

throughput graph mirrored around the *x* axis. Figure 2.26 (b), shows an excerpt from the latency graph of Seagate-CMR that is also repeated throughout the latency graph. This graph too has a pattern that is mirrored at the center, also indicating a completed ascent and descent through the surfaces. However, Seagate-CMR completes the cycle in 3.5 s since it reads only a few tracks from each surface, whereas Seagate-SMR completes the cycle in 1,800 s, indicating that it reads thousands of tracks from a single surface.

Smaller spikes at the graph of Seagate-CMR correspond to track switches, and higher spikes correspond to head switches. While the extra 1 ms head switch latency every few megabytes does not affect the accuracy of emulation, it shows up in some of the tests, for example as the bump around 4,030th MiB in Figure 2.19. Figure 2.26 also shows that the number of platters can be inferred from the latency graph of sequential reads.

2.5 Related Work

Little has been published on the subject of system-level behavior of DM-SMR drives. Although several works have discussed requirements and possibilities for use of shingled drives in systems [9, 114], only three papers to date present example translation layers and simulation results [29, 78, 118]. A range of STL approaches is found in the patent literature [40, 65, 68, 79], but evaluation and analysis is lacking. Several SMR-specific file systems have been proposed, such as SMRfs [76], SFS [115], and HiSMRfs [98]. He and Du [81] propose a static mapping to minimize re-writes for in-place updates, which requires high guard overhead (20%) and assumes file system free space is contiguous in the upper LBA region. Pitchumani et al.[147] present an emulator implemented as a Linux device mapper target that mimics shingled writing on top of a CMR drive. Tan et al.[187] describe a simulation of S-blocks algorithm, with a more accurate simulator cali-

Property	Drive Model	
	ST5000AS0011	ST8000AS0011
Drive Type	SMR	SMR
Persistent Cache Type	Disk	Disk
Cache Layout and Location	Single, at the OD	Single, at the OD
Cache Size	20 GiB	25 GiB
Cache Map Size	200,000	250,000
Band Size	17–36 MiB	15–40 MiB
Block Mapping	Static	Static
Cleaning Type	Aggressive	Aggressive
Cleaning Algorithm	FIFO	FIFO
Cleaning Time	0.6–1.6 s/band	0.6–1.6 s/band
Zone Structure	4–20 GiB	5–40 GiB
Shingling Direction	Towards MD	N/A

Table 2.3: Properties of the 5 TB and the 8 TB Seagate drives discovered using Skylight methodology. The benchmarks worked out of the box on the 8 TB drive. Since the 8 TB drive was on loan, we did not drill a hole on it; therefore, shingling direction for it is not available.

brated with data from a real CMR drive. Shafaei et al.[172, 173] propose DM-SMR models based on the Skylight work. Wu et al.[218, 219] evaluate the performance of host-aware SMR drives, which are hybrid SMR drives that can present both block interface and zone interface.

This work draws heavily on earlier disk characterization studies that have used micro-benchmarks to elicit details of internal performance, such as [161], [77], [108], [186], [216]. Due to the presence of a translation layer, however, the specific parameters examined in this work (and the micro-benchmarks for measuring them) are different.

2.6 Summary and Recommendations

As Table 2.3 shows, the Skylight methodology enables us to discover key properties of two drive-managed SMR disks automatically. With manual intervention, it allows us to completely reverse engineer a drive. The purpose of doing so is not just to satisfy our curiosity, however, but to guide both their use and evolution. In particular, we draw the following conclusions from our measurements of the 5 TB Seagate drive:

- Write latency with the volatile cache disabled is high (Test 1). This appears to be an artifact of specific design choices rather than fundamental requirements, and we hope for it to drop in later firmware revisions.
- Sequential throughput (with the volatile cache disabled) is much lower (by 3× or more, depending on write size) than for conventional drives. (We omitted these test results, as performance is identical to the random writes in Test 1.) Due to the use of static mapping (Test 12), achieving full sequential throughput requires enabling volatile cache.
- Random I/O throughput (with the volatile cache enabled or with high queue depth) is high (Test 7)—15× that of the equivalent CMR drive. This is a general property of any DM-SMR

drive using a persistent cache.

- Throughput may degrade precipitously when the cache fills after many writes (Table 2.2). The point at which this occurs depends on write size and queue depth. (Although results with the volatile cache enabled are not presented in § 2.4.6, they are similar to those for a queue depth of 31.)
- Background cleaning begins after ≈1 second of idle time, and proceeds in steps requiring 0.6–45 seconds of idle time to clean a single band (§ 2.4.9).
- Sequential reads of randomly-written data will result in random-like read performance until cleaning completes (§ 2.4.4).

DM-SMR drives like the ones we studied should offer good performance if the following conditions are met: (a) the volatile cache is enabled or a high queue depth is used, (b) writes display strong spatial locality, modifying only a few bands at any particular time, (c) non-sequential writes (or all writes, if the volatile cache is disabled) occur in bursts of less than 16 GB or 180,000 operations (Table 2.2), and (d) long powered-on idle periods are available for background cleaning.

The extra capacity and low energy consumption of DM-SMR drives make them ideal for increasing the cost-effectiveness of data storage in distributed storage systems. This is only possible, however, by a running general-purpose file system, such as ext4 or XFS, on top of DM-SMR drives. Yet in this chapter, we have shown that DM-SMR drives have high block interface tax—even a moderate amount of random writes leads to prohibitively high garbage collection overhead. We expect the high block interface tax of DM-SMR drives to cause performance problems with current file systems, which have been optimized for CMR drives for decades.

In the next chapter, we first confirm our hunch and then use our newly acquired knowledge of DM-SMR drive internals to modify the ext4 file system to avoid I/O patterns that amplifies the block interface tax.

Chapter 3

Reducing the Block Interface Tax in DM-SMR Drives by Evolving Ext4

Distributed storage systems can avoid the block interface tax by either modifying file systems to work directly on zoned devices or by modifying file systems to avoid I/O patterns that amplify the block interface tax. In this chapter we take the latter approach and introduce a simple technique that almost eliminates random writes in journaling file systems. We demonstrate our technique on the ext4 file system and achieve significant performance improvement on DM-SMR drives.

3.1 SMR Adoption and Ext4-Lazy Summary

The industry has tried to address SMR adoption by introducing two kinds of SMR drive: drive-managed (DM-SMR) and host-managed (HM-SMR). DM-SMR drives are a drop-in replacement for conventional drives that offer higher capacity with the traditional block interface, but can suffer performance degradation when subjected to non-sequential write traffic. Unlike CMR drives that have a low but consistent throughput under random writes, DM-SMR drives offer high throughput for a short period followed by a precipitous drop, as shown in Figure 3.1. HM-SMR drives, on the other hand, offer the backward-incompatible *zone interface* that requires major changes to the I/O stacks to allow SMR-aware software to optimize their access pattern.

A new HM-SMR drive interface presents an interesting problem to storage researchers who have already proposed new file system designs based on it [98, 115][33]. It also presents a challenge to the developers of existing file systems [36, 57, 59] who have been optimizing their code for CMR drives for years. There have been attempts to revamp mature Linux file systems like ext4 and XFS [35, 140, 141] to use the new zone interface, but these attempts have stalled due to the large amount of redesign required. The Log-Structured File System (LFS) [158], on the other hand, has an architecture that can be most easily adapted to an HM-SMR drive. However, although LFS has been influential, hard disk file systems based on it [169][107] have not reached production quality in practice [124, 159][135] .

In this work, we take an alternative approach to SMR adoption. Instead of redesigning for the zone interface used by HM-SMR drives, we make an incremental change to a mature, high performance file system, to optimize its performance on a DM-SMR drive. The systems community

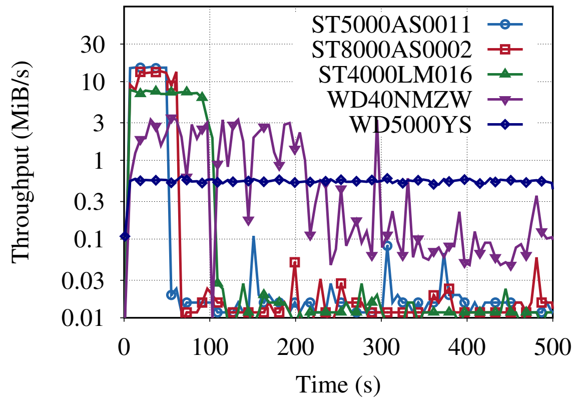


Figure 3.1: Throughput of CMR and DM-SMR drives from Table 3.1 under 4 KiB random write traffic. CMR drive has a stable but low throughput under random writes. DM-SMR drive, on the other hand, have a short period of high throughput followed by a continuous period of ultra-low throughput.

Type	Vendor	Model	Capacity	Form Factor
DM-SMR	Seagate	ST8000AS0002	8 TB	3.5 inch
DM-SMR	Seagate	ST5000AS0011	5 TB	3.5 inch
DM-SMR	Seagate	ST4000LM016	4 TB	2.5 inch
DM-SMR	Western Digital	WD40NMZW	4 TB	2.5 inch
CMR	Western Digital	WD5000YS	500 GB	3.5 inch

Table 3.1: CMR and DM-SMR drives from two vendors used for evaluation.

is no stranger to taking a revolutionary approach when faced with a new technology [23], only to discover that the existing system can be evolved to take the advantage of the new technology with a little effort [24]. Following a similar evolutionary approach, we take the first step to optimize the ext4 file system for DM-SMR drives, observing that random writes are even more expensive on these drives, and that metadata writeback is a key generator of it.

We introduce ext4-lazy, a small change to ext4 that eliminates most metadata writeback. Like other journaling file systems [152], ext4 writes metadata twice: As Figure 3.2 (a) shows, it first writes the metadata block to a temporary location J in the journal and then marks the block as *dirty* in memory. Once it has been in memory for long enough (controlled by `/proc/sys/vm/dirty_expire_centisecs` in Linux), the *writeback* (or *flusher*) thread writes the block to its *static location* S, resulting in a random write. Although metadata writeback is typically a small portion of a workload, it results in many random writes, as Figure 3.3 shows. Ext4-lazy, on the other hand, marks the block as *clean* after writing it to the journal, to prevent the writeback, and inserts a mapping (S, J) to an in-memory map allowing the file system to access the block in the journal, as seen in Figure 3.2 (b). Ext4-lazy uses a large journal so that it can continue writing updated blocks while reclaiming the space from the stale blocks. During mount, it reconstructs the in-memory map from the journal resulting in a modest increase in mount time. Our results show that ext4-lazy significantly improves performance on DM-SMR drives, as well as on CMR drives

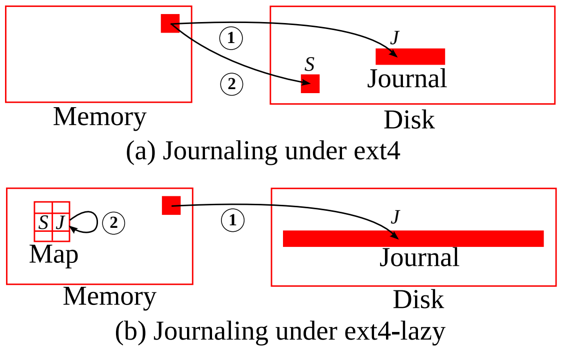


Figure 3.2: (a) Ext4 writes a metadata block to disk twice. It first writes the metadata block to the journal at some location J and marks it dirty in memory. Later, the writeback thread writes the same metadata block to its static location S on disk, resulting in a random write. (b) Ext4-lazy, writes the metadata block approximately once to the journal and inserts a mapping (S, J) to an in-memory map so that the file system can find the metadata block in the journal.

for metadata-heavy workloads.

Our key contribution in this work is the design, implementation, and evaluation of ext4-lazy on DM-SMR and CMR drives. Our change is minimally invasive—we modify 80 lines of existing code and introduce the new functionality in additional files totaling 600 lines of C code. On a metadata-light ($\leq$ 1% of total writes) file server benchmark, ext4-lazy increases DM-SMR drive throughput by 1.7-5.4×. For directory traversal and metadata-heavy workloads it achieves 2-13× improvement on both DM-SMR and CMR drives.

In addition, we make two contributions that are applicable beyond our proposed approach:

- For purely sequential write workloads, DM-SMR drives perform at full throughput and do not suffer performance degradation. We identify the minimal sequential I/O size to trigger this behavior for a popular DM-SMR drive.
- We show that for physical journaling [152], a small journal is a bottleneck for metadata-heavy workloads. Based on our result, ext4 developers have increased the default journal size from 128 MiB to 1 GiB for file systems over 128 GiB [193].

3.2 Background on the Ext4 File System

The ext4 file system evolved [111, 127] from ext2 [28], which was influenced by Fast File System (FFS) [128]. Similar to FFS, ext2 divides the disk into *cylinder groups*—or as ext2 calls them, *block groups*—and tries to put all blocks of a file in the same block group. To further increase locality, the metadata blocks (*inode bitmap*, *block bitmap*, and *inode table*) representing the files in a block group are also placed within the same block group, as Figure 3.4 (a) shows. *Group descriptor blocks*, whose location is fixed within the block group, identify the location of these metadata blocks that are typically located in the first megabyte of the block group.

In ext2 the size of a block group was limited to 128 MiB—the maximum number of 4 KiB data blocks that a 4 KiB block bitmap can represent. Ext4 introduced *flexible block groups* or *flex_*

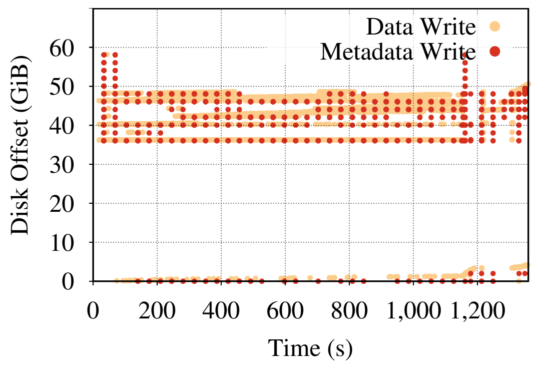


Figure 3.3: Offsets of data and metadata writes obtained with `blktrace`, when compiling Linux kernel 4.6 with all of its modules on a fresh ext4 file system. The workload writes 12 GiB of data, 185 MiB of journal (omitted from the graph), and only 98 MiB of metadata, making it 0.77% of total writes.

bgs [111], a set of contiguous block groups (we assume the default size of 16 block groups per flex_bg) whose metadata is consolidated in the first 16 MiB of the first block group within the set, as shown in Figure 3.4 (b).

Ext4 ensures metadata consistency via journaling, however, it does not implement journaling itself; rather, it uses a generic kernel layer called the *Journaling Block Device* [194] that runs in a separate kernel thread called *jbd2*. In response to file system operations, ext4 reads metadata blocks from disk, updates them in memory, and exposes them to jbd2 for journaling. For increased performance, jbd2 batches metadata updates from multiple file system operations (by default, for 5 seconds) into a transaction buffer and atomically commits the transaction to the journal—a circular log of transactions with a head and tail pointer. A transaction may commit early if the buffer reaches maximum size, or if a synchronous write is requested. In addition to metadata blocks, a committed transaction contains *descriptor blocks* that record the static locations of the metadata blocks within the transaction. After a commit, jbd2 marks the in-memory copies of metadata blocks as dirty so that the writeback threads would write them to their static locations. If a file system operation updates an in-memory metadata block before its dirty timer expires, jbd2 writes the block to the journal as part of a new transaction and delays the writeback of the block by resetting its timer.

On DM-SMR drives, when the metadata blocks are eventually written back, they dirty the bands that are mapped to the metadata regions in a flex_bg, as seen in Figure 3.4 (c). The bottom part of Figure 3.4 (c) shows the logical view of Seagate ST8000AS0002—an 8 TB DM-SMR drive we studied in Chapter 2. With an average band size of 30MiB, the drive has over 260,000 bands with sectors *statically* mapped to the bands, and a ≈ 25 GiB persistent cache that is not visible to the host (and not shown in figure). The STL in this drive detects sequential writes and starts streaming them directly to the bands, bypassing the persistent cache. Random writes, however, end up in the persistent cache, dirtying bands. Since a metadata region is not aligned with a band, metadata writes to it may dirty zero, one, or two extra bands, depending on whether the metadata region spans one or two bands and whether the data around the metadata region has been written.

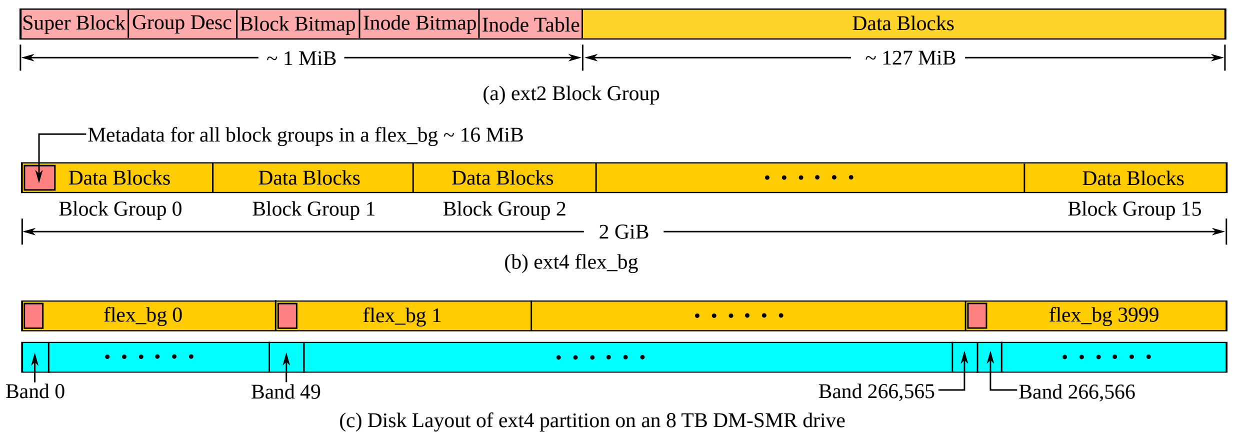


Figure 3.4: (a) In ext2, the first megabyte of a 128 MiB block group contains the metadata blocks describing the block group, and the rest is data blocks. (b) In ext4, a single flex_bg concatenates multiple (16 in this example) block groups into one giant block group and puts all of the metadata in the first block group. (c) Modifying data in a flex_bg will result in a metadata write that may dirty one or two bands, seen at the boundary of bands 266,565 and 266,566.

3.3 Design and Implementation of ext4-lazy

We start by motivating ext4-lazy, follow with a high-level view of our design, and finish with the implementation details.

3.3.1 Motivation

The motivation for ext4-lazy comes from two observations: (1) metadata writeback in ext4 results in random writes that cause a significant cleaning load on a DM-SMR drive, and (2) file system metadata comprises a small set of blocks, and *hot* (frequently updated) metadata is an even smaller set. The corollary of the latter observation is that managing hot metadata in a circular log several times the size of hot metadata turns random writes into purely sequential writes, reducing the cleaning load on a DM-SMR drive. We first give calculated evidence supporting the first observation and follow with empirical evidence for the second observation.

On an 8 TB partition, there are about 4,000 flex_bgs, the first 16 MiB of each containing the metadata region, as shown in Figure 3.4 (c). With a 30 MiB band size, updating every flex_bg would dirty 4,000 bands on average, requiring cleaning of 120 GiB worth of bands, generating 360 GiB of disk traffic. A workload touching 1/16 of the whole disk, that is 500 GiB of files, would dirty at least 250 bands requiring 22.5 GiB of cleaning work. The cleaning load increases further if we consider floating metadata like extent tree blocks and directory blocks.

To measure the hot metadata ratio, we emulated the I/O workload of a build server on ext4, by running 128 parallel Compilebench [126] instances, and categorized all of the writes completed by disk. Out of 433 GiB total writes, 388 GiB were data writes, 34 GiB were journal writes, and 11 GiB were metadata writes. The total size of unique metadata blocks was 3.5 GiB, showing that it was only 0.8% of total writes, and that 90% of journal writes were overwrites.

3.3.2 Design

At a high level, ext4-lazy adds the following components to ext4 and jbd2:

Map: Ext4-lazy tracks the location of metadata blocks in the journal with *jmap*—an in-memory map that associates the static location S of a metadata block with its location J in the journal. The mapping is updated whenever a metadata block is written to the journal, as shown in Figure 3.2 (b).

Indirection: In ext4-lazy all accesses to metadata blocks go through jmap. If the most recent version of a block is in the journal, there will be an entry in jmap pointing to it; if no entry is found, then the copy at the static location is up-to-date.

Cleaner: The cleaner in ext4-lazy reclaims space from locations in the journal which have become *stale*, that is, invalidated by the writes of new copies of the same metadata block.

Map reconstruction on mount: On every mount, ext4-lazy reads the descriptor blocks from the transactions between the tail and the head pointer of the journal and populates jmap.

3.3.3 Implementation

We now detail our implementation of the above components and the trade-offs we make during the implementation. We implement jmap as a standard Linux red-black tree [113] in jbd2. After jbd2 commits a transaction, it updates jmap with each metadata block in the transaction and marks the in-memory copies of those blocks as clean so they will not be written back. We add indirect lookup of metadata blocks to ext4 by changing the call sites that read metadata blocks to use a function which looks up the metadata block location in jmap, as shown in Listing 3.1, modifying 40 lines of ext4 code in total.

```
- submit_bh(READ | REQ_META | REQ_PRIO, bh);
+ jbd2_submit_bh(journal, READ | REQ_META | REQ_PRIO, bh);
```

Listing 3.1: Adding indirection to a call site reading a metadata block.

The indirection allows ext4-lazy to be backward-compatible and gradually move metadata blocks to the journal. However, the primary reason for indirection is to be able to migrate *cold* (not recently updated) metadata back to its static location during cleaning, leaving only hot metadata in the journal.

We implement the cleaner in jbd2 in just 400 lines of C, leveraging the existing functionality. In particular, the cleaner merely reads live metadata blocks from the tail of the journal and adds them to the transaction buffer using the same interface used by ext4. For each transaction it keeps a doubly-linked list that links jmap entries containing live blocks of the transaction. Updating a jmap entry invalidates a block and removes it from the corresponding list. To clean a transaction, the cleaner identifies the live blocks of a transaction in constant time using the transaction's list, reads them, and adds them to the transaction buffer. The beauty of this cleaner is that it does not "stop-the-world", but transparently mixes cleaning with regular file system operations causing no interruptions to them, as if cleaning was just another operation. We use a simple cleaning policy—after committing a fixed number of transactions, clean a fixed number of transactions—and leave sophisticated policy development, such as hot and cold separation, for future work.

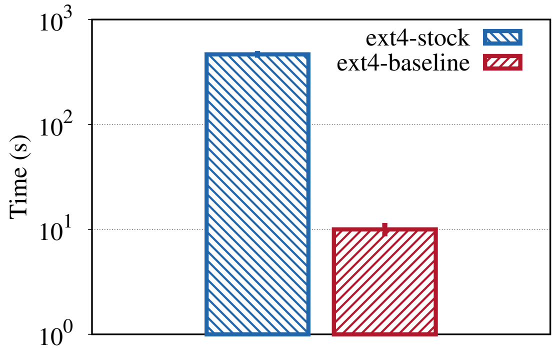


Figure 3.5: Completion time for a benchmark creating 100,000 files on ext4-stock (ext4 with 128 MiB journal) and on ext4-baseline (ext4 with 10 GiB journal).

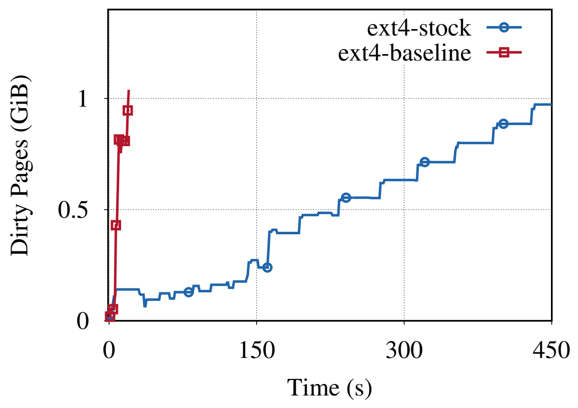


Figure 3.6: The volume of dirty pages during benchmark runs obtained by sampling `/proc/meminfo` every second on ext4-stock and ext4-baseline.

Map reconstruction is a small change to the recovery code in jbd2. Stock ext4 resets the journal on a normal shutdown; finding a non-empty journal on mount is a sign of crash and triggers the recovery process. With ext4-lazy, the state of the journal represents the persistent image of jmap, therefore, ext4-lazy never resets the journal and always "recovers". In our prototype, ext4-lazy reconstructs the jmap by reading descriptor blocks from the transactions between the tail and head pointer of the journal, which takes ≈ 5 seconds when the space between the head and tail pointer is ≈ 1 GiB.

3.4 Evaluation

We run all experiments on a system with a quad-core Intel i7-3820 (Sandy Bridge) 3.6 GHz CPU, 16 GB of RAM running Linux kernel 4.6 on the Ubuntu 14.04 distribution, using the drives listed in Table 3.1. To reduce the variance between runs, we unmount the file system between runs, always start with the same file system state, disable lazy initialization (`mkfs.ext4 -E lazy_itable_init=0,lazy_journal_init=0 /dev/<dev>`) when formatting ext4 partitions, and fix the writeback cache ratio [227] for our disks to 50% of the total—by default, this ratio is computed dynamically from the writeback throughput [192]. We repeat every experiment at least five times and report the average and standard deviation of the runtime.

3.4.1 Journal Bottleneck

Since it affects our choice of baseline, we start by showing that for metadata-heavy workloads, the default 128 MiB journal of ext4 is a bottleneck. We demonstrate the bottleneck on the CMR drive WD5000YS from Table 3.1 by creating 100,000 small files in over 60,000 directories, using *Create-Files* microbenchmark from Filebench [188]. The workload size is ≈ 1 GiB and fits in memory.

Although ext4-lazy uses a large journal by definition, since enabling a large journal on ext4 is a command-line option to `mkfs`, we choose ext4 with a 10 GiB journal (created by passing `‘‘-J size=10240’’` to `mkfs.ext4`) as our baseline. In the rest of this paper, we refer to ext4 with the default journal size of 128 MiB as *ext4-stock*, and we refer to ext4 with 10 GiB journal as *ext4-baseline*.

We measure how fast ext4 can create the files in memory and do not consider the writeback time. Figure 3.5 shows that on ext4-stock the benchmark completes in $\approx$ 460 seconds, whereas on ext4-baseline it completes 46× faster, in $\approx$ 10 seconds. Next we show how a small journal becomes a bottleneck.

The ext4 journal is a circular log of transactions with a head and tail pointer (§ 3.2). As the file system performs operations, jbd2 commits transactions to the journal, moving the head forward. A committed transaction becomes *checkpointed* when every metadata block in it is either written back to its static location due to a dirty timer expiration, or it is written to the journal as part of a newer transaction. To recover space, at the end of every commit jbd2 checks for transactions at the tail that have been checkpointed, and when possible moves the tail forward. On a metadata-light workload with a small journal and default dirty timer, jbd2 always finds checkpointed transactions at the tail and recovers the space without doing work. However, on a metadata-heavy workload, incoming transactions fill the journal before the transactions at the tail have been checkpointed. This results in a *forced checkpoint*, where jbd2 synchronously writes metadata blocks at the tail transaction to their static locations and then moves the tail forward, so that a new transaction can start [194].

We observe the file system behavior while running the benchmark by enabling tracepoints in the jbd2 code (`/sys/kernel/debug/tracing/events/jbd2/`). On ext4-stock, the journal fills in 3 seconds, and from then on until the end of the run, jbd2 moves the tail by performing forced checkpoints. On ext4-baseline the journal never becomes full and no forced checkpoints happen during the run.

Figure 3.6 shows the volume of dirtied pages during the benchmark runs. On ext4-baseline, the benchmark creates over 60,000 directories and 100,000 files, dirtying about 1 GiB worth of pages in 10 seconds. On ext4-stock, directories are created in the first 140 seconds. Forced checkpoints still happen during this period, but they complete fast, as the small steps in the first 140 seconds show. Once the benchmark starts filling directories with files, the block groups fill and writes spread out to a larger number of block groups across the disk. Therefore, forced checkpoints start taking as long as 30 seconds, as indicated by the large steps, during which the file system stalls, no writes to files happen, and the volume of dirtied pages stays fixed.

This result shows that for disks, a small journal is a bottleneck for metadata-heavy buffered I/O workloads, as the journal wraps before metadata blocks are written to disk, and file system operations are stalled until the journal advances via synchronous writeback of metadata blocks. With a sufficiently large journal, all transactions will be written back before the journal wraps. For example, for a 190 MiB/s disk and a 30 second dirty timer, a journal size of 30s × 190 MiB/s = 5,700 MiB will guarantee that when the journal wraps, the transactions at the tail will be checkpointed. Having established our baseline, we move on to evaluation of ext4-lazy.

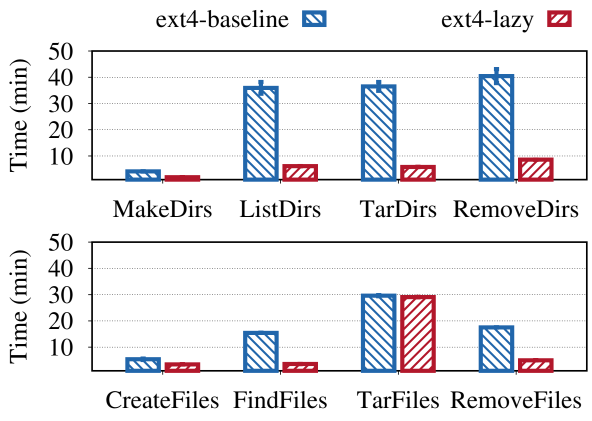


Figure 3.7: Microbenchmark runtimes on ext4-baseline and ext4-lazy.

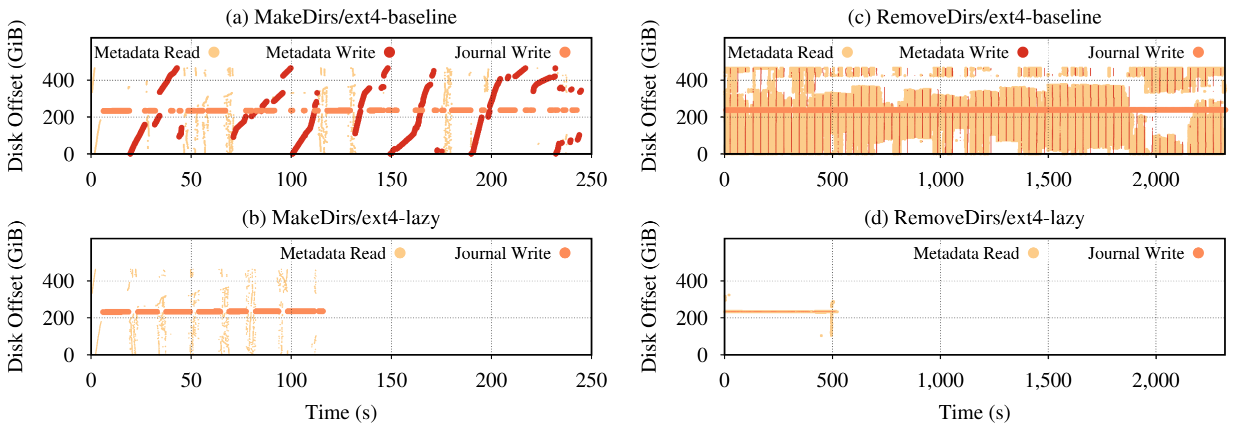


Figure 3.8: Disk offsets of I/O operations during MakeDirs and RemoveDirs microbenchmarks on ext4-baseline and ext4-lazy. Metadata reads and writes are spread out while journal writes are at the center. The dots have been scaled based on the I/O size. In part (d), journal writes are not visible due to low resolution. These are pure metadata workloads with no data writes.

3.4.2 Ext4-lazy on a CMR Drive

We first evaluate ext4-lazy on the CMR drive WD5000YS from Table 3.1 via a series of microbenchmarks and a file server macrobenchmark. We show that on a CMR drive, ext4-lazy provides a significant speedup for metadata-heavy workloads, and specifically for massive directory traversal workloads. On metadata-light workloads, however, ext4-lazy does not have much impact.

Microbenchmarks

We evaluate directory traversal and file/directory create operations using the following benchmarks. *MakeDirs* creates 800,000 directories in a directory tree of depth 10. *ListDirs* runs `ls -lR` on the directory tree. *TarDirs* creates a tarball of the directory tree, and *RemoveDirs* removes the directory tree. *CreateFiles* creates 600,000 4 KiB files in a directory tree of depth 20. *FindFiles*

	Metadata Reads	Metadata Writes	Journal Writes
MakeDirs/ext4-baseline	143.7±2.8 MiB	4,631±33.8 MiB	4,735±0.1 MiB
MakeDirs/ext4-lazy	144±4 MiB	0 MiB	4,707±1.8 MiB
RemoveDirs/ext4-baseline	4,066.4±0.1 MiB	322.4±11.9 MiB	1,119±88.6 MiB
RemoveDirs/ext4-lazy	4,066.4±0.1 MiB	0 MiB	472±3.9 MiB

Table 3.2: Distribution of the I/O types with MakeDirs and RemoveDirs benchmarks running on ext4-baseline and ext4-lazy.

runs `find` on the directory tree. *TarFiles* creates a tarball of the directory tree, and *RemoveFiles* removes the directory tree. MakeDirs and CreateFiles—microbenchmarks from Filebench—run with 8 threads and execute sync at the end. All benchmarks start with a cold cache obtained by executing `echo 3 > /proc/sys/vm/drop_caches` as root.

Benchmarks that are in the file/directory create category (MakeDirs, CreateFiles) complete 1.5-2× faster on ext4-lazy than on ext4-baseline, while the remaining benchmarks that are in the directory traversal category, except TarFiles, complete 3-5× faster, as seen in Figure 3.7. We choose MakeDirs and RemoveDirs as a representative of each category and analyze their performance in detail.

MakeDirs on ext4-baseline results in ≈ 4,735 MiB of journal writes that are transaction commits containing metadata blocks, as seen in the first row of Table 3.2 and at the center in Figure 3.8 (a); as the dirty timer on the metadata blocks expires, they are written to their static locations, resulting in a similar amount of metadata writeback. The block allocator is able to allocate large contiguous blocks for the directories, because the file system is fresh. Therefore, in addition to journal writes, metadata writeback is sequential as well. The write time dominates the runtime in this workload, hence, by avoiding metadata writeback and writing only to the journal, ext4-lazy halves the writes as well as the runtime, as seen in the second row of Table 3.2 and Figure 3.8 (b). On an aged file system, the metadata writeback is more likely to be random, resulting in even higher improvement on ext4-lazy.

An interesting observation about Figure 3.8 (b) is that although the total volume of metadata reads—shown as periodic vertical spreads—is ≈ 140 MiB (3% of total I/O in the second row of Table 3.2), they consume over 30% of runtime due to long seeks across the disk. In this benchmark, the metadata blocks are read from their static locations because we run the benchmark on a fresh file system, and the metadata blocks are still at their static locations. As we show next, once the metadata blocks migrate to the journal, reading them is much faster since no long seeks are involved.

In RemoveDirs benchmark, on both ext4-baseline and ext4-lazy, the disk reads ≈ 4,066 MiB of metadata, as seen in the last two rows of Table 3.2. However, on ext4-baseline the metadata blocks are scattered all over the disk, resulting in long seeks as indicated by the vertical spread in Figure 3.8 (c), while on ext4-lazy they are within the 10 GiB region in the journal, resulting in only short seeks, as Figure 3.8 (d) shows. Ext4-lazy also benefits from skipping metadata writeback, but most of the improvement comes from eliminating long seeks for metadata reads. The significant difference in the volume of journal writes between ext4-baseline and ext4-lazy seen in Table 3.2 is caused by metadata write coalescing: since ext4-lazy completes faster, there are more operations

	Data Writes	Metadata Writes	Journal Writes
ext4-baseline	34,185±10.3 MiB	480±0.2 MiB	1,890±18.6 MiB
ext4-lazy	33,878±9.8 MiB	0 MiB	1,855±15.4 MiB

Table 3.3: Distribution of write types completed by the disk during Postmark run on ext4-baseline and ext4-lazy. Metadata writes make 1.3% of total writes in ext4-baseline, only ⅓ of which is unique.

in each transaction, with many modifying the same metadata blocks, each of which is only written once to the journal.

The improvement in the remaining benchmarks, are also due to reducing seeks to a small region and avoiding metadata writeback. We do not observe a dramatic improvement in TarFiles, because unlike the rest of the benchmarks that read only metadata from the journal, TarFiles also reads data blocks of files that are scattered across the disk.

Massive directory traversal workloads are a constant source of frustration for users of most file systems [11, 72, 120, 144, 171]. One of the biggest benefits of consolidating metadata in a small region is an order of magnitude improvement in such workloads, which to our surprise was not noticed by previous work [145, 156, 224]. On the other hand, the above results are obtainable in the ideal case that all of the directory blocks are hot and therefore kept in the journal. If, for example, some part of the directory is cold and the policy decides to move those blocks to their static locations, removing such a directory will incur an expensive traversal.

File Server Macrobenchmark

We first show that ext4-lazy slightly improves the throughput of a metadata-light file server workload. Next we try to reproduce a result from previous work without success.

To emulate a file server workload, we first started with the *Fileserver* macrobenchmark from Filebench, but we encountered bugs for large configurations. The development on Filebench has been recently restarted and the recommended version is still in alpha stage. Therefore, we decided to use Postmark [102], with some modifications.

Like the Fileserver macrobenchmark from Filebench, Postmark first creates a *working set* of files and directories and then executes *transactions* like reading, writing, appending, deleting, and creating files on the working set. We modify Postmark to execute sync after creating the working set, so that the writeback of the working set does not interfere with transactions. We also modify Postmark not to delete the working set at the end, but to run sync, to avoid high variance in runtime due to the race between deletion and writeback of data.

Our Postmark configuration creates a working set of 10,000 files spread sparsely across 25,000 directories with file sizes ranging from 512 bytes to 1 MiB, and then executes 100,000 transactions with the I/O size of 1 MiB. During the run, Postmark writes 37.89 GiB of data and reads 31.54 GiB of data from user space. Because ext4-lazy reduces the amount of writes, to measure its effect, we focus on writes.

Table 3.3 shows the distribution of data writes completed by the disk while the benchmark is running on ext4-baseline and on ext4-lazy. On ext4-baseline, metadata writes comprise 1.3% of

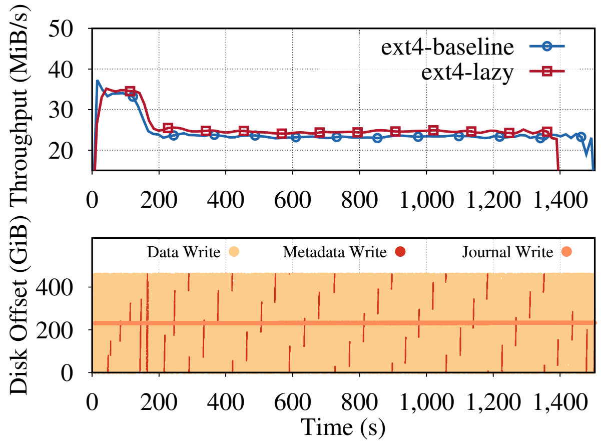


Figure 3.9: The top graph shows the throughput of the disk during a Postmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during ext4-baseline run. The graph does not reflect sizes of the writes, but only their offsets.

total writes, all of which ext4-lazy avoids. As a result, the disk sees 5% increase in throughput on ext4-lazy from 24.24 MiB/s to 25.47 MiB/s and the benchmark completes 100 seconds faster on ext4-lazy, as the throughput graph in Figure 3.9 shows. The increase in throughput is modest because the workload spreads out the files across the disk resulting in traffic that is highly non-sequential, as data writes in the bottom graph of Figure 3.9 show. Therefore, it is not surprising that reducing random writes of a non-sequential write traffic by 1.3% results in a 5% throughput improvement. However, the same random writes result in extra cleaning work for DM-SMR drives (§ 3.3.1).

Previous work [145] that writes metadata only once reports performance improvements even in a metadata-light workloads, like kernel compile. This has not been our experience. We compiled Linux kernel 4.6 with all its modules on ext4-baseline and observed that it generated 12 GiB of data writes and 185 MiB of journal writes. At 98 MiB, metadata writes comprised only 0.77% of total writes completed by the disk. This is expected, since metadata blocks are cached in memory, and because they are journaled, unlike data pages their dirty timer is reset whenever they are modified (§ 3.3), delaying their writeback. Furthermore, even on a system with 8 hardware threads running 16 parallel jobs, we found kernel compile to be CPU-bound rather than disk-bound, as Figure 3.10 shows. Given that reducing writes by 1.3% on a workload that utilized the disk 100% resulted in only 5% increase in throughput (Figure 3.9), it is not surprising that reducing writes by 0.77% on such a low-utilized disk does not cause improvement.

3.4.3 Ext4-lazy on DM-SMR Drives

We show that unlike CMR drives, where ext4-lazy had a big impact on just metadata-heavy workloads, on DM-SMR drives it provides significant improvement on both, metadata-heavy and metadata-light workloads. We also identify the minimal sequential I/O size to trigger streaming writes on a popular DM-SMR drive.

An additional critical factor for file systems when running on DM-SMR drives is the cleaning

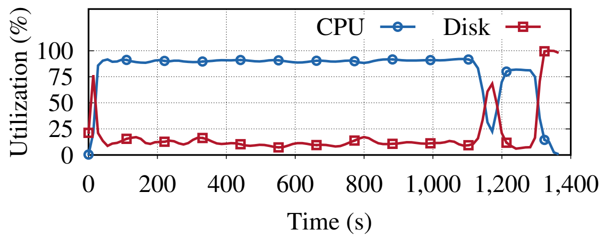


Figure 3.10: Disk and CPU utilization sampled from `iostat` output every second, while compiling Linux kernel 4.6 including all its modules, with 16 parallel jobs (`make -j16`) on a quad-core Intel i7-3820 (Sandy Bridge) CPU with 8 hardware threads.

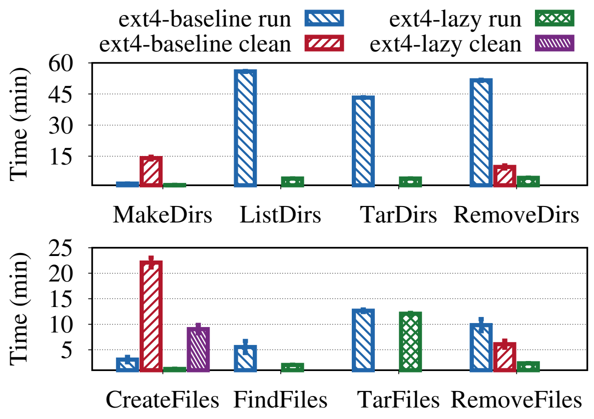


Figure 3.11: Microbenchmark runtimes and cleaning times on ext4-baseline and ext4-lazy running on a DM-SMR drive. Cleaning time is the additional time after the benchmark run that the DM-SMR drive was busy cleaning.

time after a workload. A file system resulting in a short cleaning time gives the drive a better chance of emptying the persistent cache during idle times of a bursty I/O workload, and has a higher chance of continuously performing at the persistent cache speed, whereas a file system resulting in a long cleaning time is more likely to force the drive to interleave cleaning with file system user work.

In the next section we show microbenchmark results on just one of the DM-SMR drives—ST8000AS0002 from Table 3.1. At the end of every benchmark, we run a vendor provided script that polls the disk until it has completed background cleaning and reports the total cleaning time, which we report in addition to the benchmark runtime. We achieve similar normalized results for the remaining drives.

Microbenchmarks

Figure 3.11 shows results of the microbenchmarks (§ 3.4.2) repeated on ST8000AS0002 with a 2 TB partition, on ext4-baseline and ext4-lazy. MakeDirs and CreateFiles do not fill the persistent cache, therefore, they typically complete 2-3× faster than on CMR drive. Similar to CMR drive, MakeDirs and CreateFiles are 1.5-2.5× faster on ext4-lazy. On the other hand, the remaining directory traversal benchmarks, ListDir for example, completes 13× faster on ext4-lazy, compared to being 5× faster on CMR drive.

The cleaning times for ListDirs, FindFiles, TarDirs, and TarFiles are zero because they do not write to disk. (TarDirs and TarFiles write their output to a different disk.) However, cleaning time for MakeDirs on ext4-lazy is zero as well, compared to ext4-baseline's 846 seconds, despite having written over 4 GB of metadata, as Table 3.2 shows. Being a pure metadata workload, MakeDirs on ext4-lazy consists of journal writes only, as Figure 3.8 (b) shows, all of which are streamed, bypassing the persistent cache and resulting in zero cleaning time. Similarly, cleaning time for RemoveDirs and RemoveFiles are 10-20 seconds on ext4-lazy compared to 590-366 seconds on ext4-baseline, because these too are pure metadata workloads resulting in only journal writes for ext4-lazy. During deletion, however, some journal writes are small and end up in persistent cache, resulting in short cleaning times.

We confirmed that the drive was streaming journal writes in previous benchmarks by repeating the MakeDirs benchmark on the DM-SMR drive with an observation window shown in Figure 2.3 and watching the head movement. We first identified the physical location of the journal on the platter by observing the head while reading the journal blocks. We then observed that shortly after starting the MakeDirs benchmark, the head moved to the physical location of the journal on the platter and remained there until the end of the benchmark. This observation lead to Test 15 for identifying the minimal sequential write size that triggers streaming. Using this test, we found that sequential writes of at least 8 MiB in size are streamed. We also observed that a single 4 KiB random write in the middle of a sequential write disrupted streaming and moved the head to the persistent cache; soon the head moved back and continued streaming.

Test 15: Identify the minimal sequential write size for streaming

```
1 Choose identifiable location $L$ on the platter
2 Start with a large sequential write size $S$
3 do
      Write $S$ bytes sequentially at $L$
      $S = S - 1$ MiB
  while Head moves to $L$ and stays there until the end of the write
4 $S = S + 1$ MiB
5 Minimal sequential write size for streaming is $S$
```

File Server Macrobenchmark

We show that on DM-SMR drives the benefit of ext4-lazy increases with the partition size and ext4-lazy achieves a significant speedup on a variety of DM-SMR drives with different STLs and persistent cache sizes.

Table 3.4 shows the distribution of write types completed by a ST8000AS0002 DM-SMR drive with a 400 GB partition during the file server macrobenchmark (§ 3.4.2). On ext4-baseline, metadata writes make up 1.6% of total writes. Although the unique amount of metadata is only ≈ 120 MiB, as the storage slows down, metadata writeback increases slightly, because each operation takes a long time to complete and the writeback of a metadata block occurs before the dirty timer is reset.

	Data Writes	Metadata Writes	Journal Writes
ext4-baseline	32,917±9.7 MiB	563±0.9 MiB	1,212±12.6 MiB
ext4-lazy	32,847±9.3 MiB	0 MiB	1,069±11.4 MiB

Table 3.4: Distribution of write types completed by a ST8000AS0002 DM-SMR drive during a Postmark run on ext4-baseline and ext4-lazy. Metadata writes make up 1.6% of total writes in ext4-baseline, only 1/5 of which is unique.

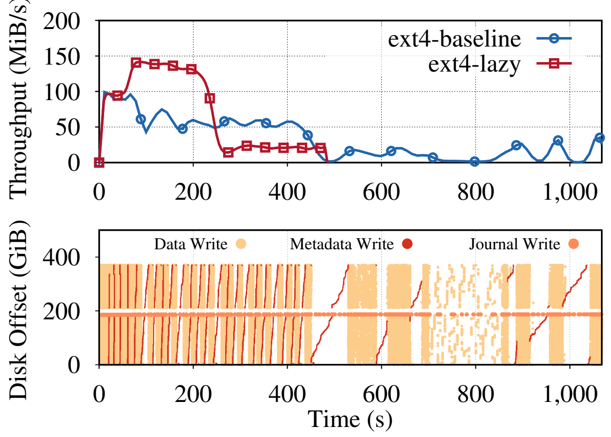


Figure 3.12: The top graph shows the throughput of a ST8000AS0002 DM-SMR drive with a 400 GB partition during a Postmark run on ext4-baseline and ext4-lazy. The bottom graph shows the offsets of write types during the run on ext4-baseline. The graph does not reflect sizes of the writes, but only their offsets.

Unlike on the CMR drive, the effect is profound on the ST8000AS0002 DM-SMR drive. The benchmark completes more than 2× faster on ext4-lazy, in 461 seconds, as seen in Figure 3.12. On ext4-lazy, the drive sustains 140 MiB/s throughput and fills the persistent cache in 250 seconds, and then drops to a steady 20 MiB/s until the end of the run. On ext4-baseline, however, the large number of small metadata writes reduce throughput to 50 MiB/s taking the drive 450 seconds to fill the persistent cache. Once the persistent cache fills, the drive interleaves cleaning and file system user work, and small metadata writes become prohibitively expensive, as seen, for example, between seconds 450-530. During this period we do not see any data writes, because the writeback thread alternates between page cache and buffer cache when writing dirty blocks, and it is the buffer cache's turn. We do, however, see journal writes because jbd2 runs as a separate thread and continues to commit transactions.

The benchmark completes even slower on a full 8 TB partition, as seen in Figure 3.13 (a), because ext4 spreads the same workload over more bands. With a small partition, updates to different files are likely to update the same metadata region. Therefore, cleaning a single band frees more space in the persistent cache, allowing it to accept more random writes. With a full partition, however, updates to different files are likely to update different metadata regions; now the cleaner has to clean a whole band to free a space for a single block in the persistent cache. Hence, after an hour of ultra-low throughput due to cleaning, it recovers slightly towards the end, and the

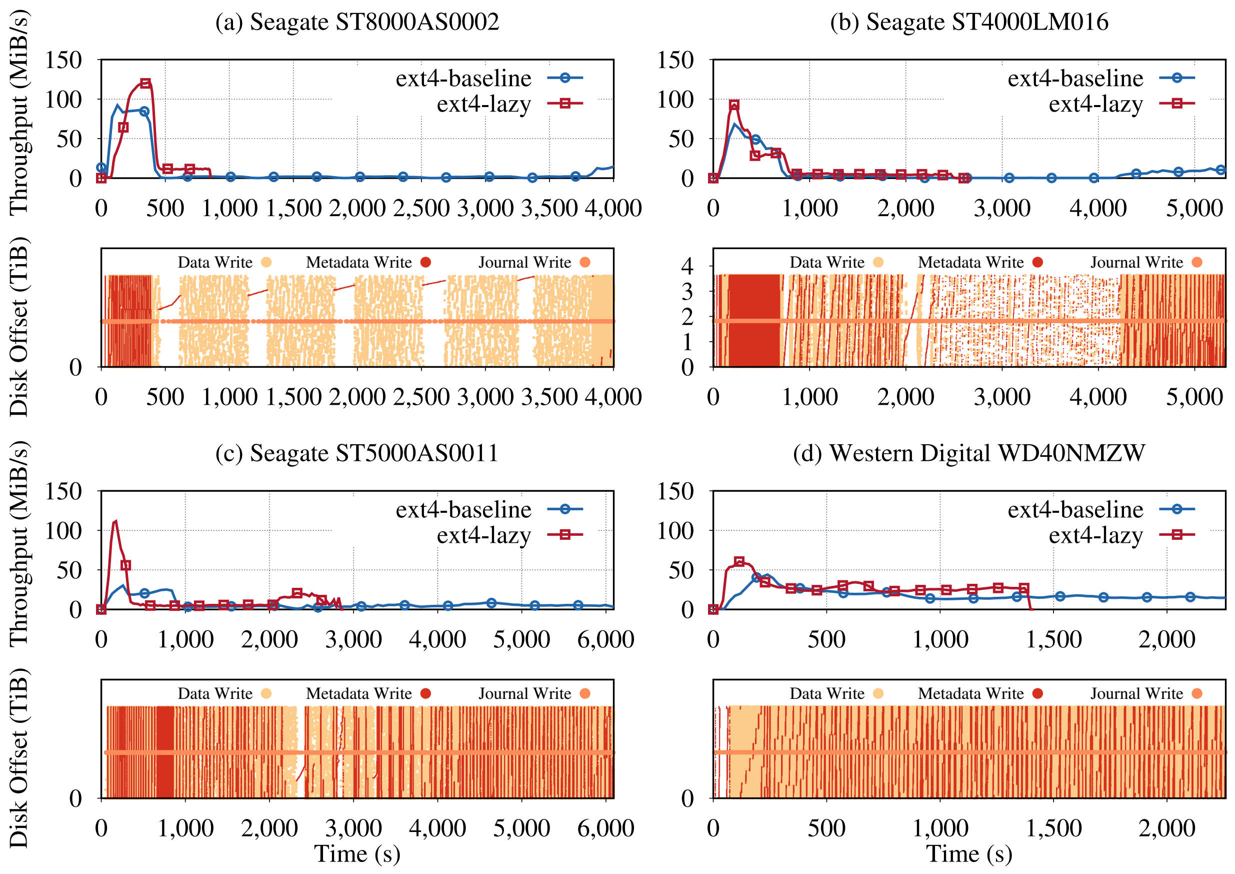


Figure 3.13: The top graphs show the throughput of four DM-SMR drives on a full disk partition during a Postmark run on ext4-baseline and ext4-lazy. Ext4-lazy provides a speedup of 5.4× 2×, 2×, 1.7× in parts (a), (b), (c), and (d), respectively. The bottom graphs show the offsets of write types during ext4-baseline run. The graphs do not reflect sizes of the writes, but only their offsets.

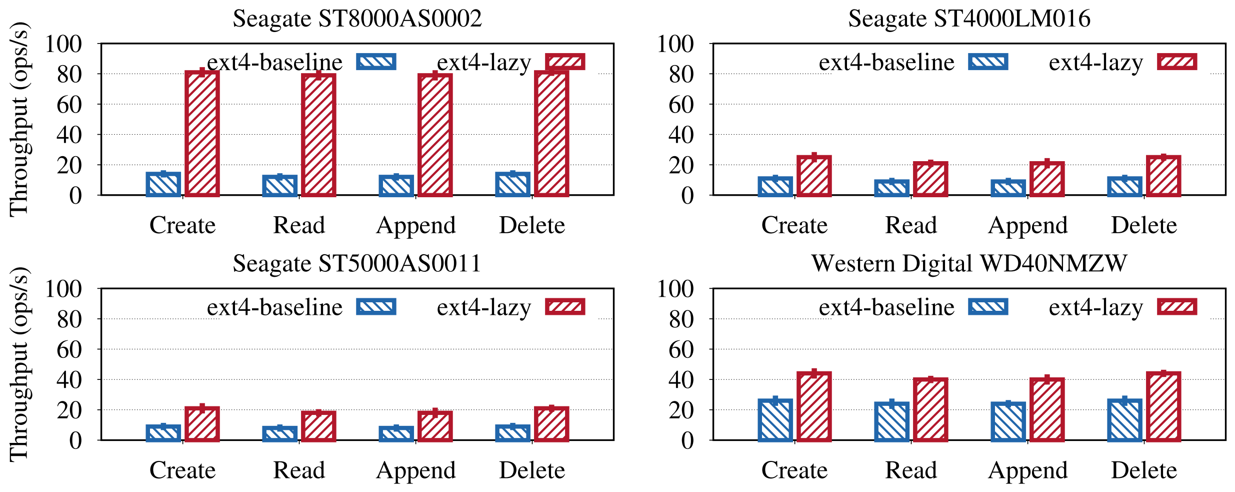


Figure 3.14: Postmark reported transaction throughput numbers for ext4-baseline and ext4-lazy running on four DM-SMR drives with a full disk partition. Only includes numbers from the transaction phase of the benchmark.

benchmark completes 5.4× slower on ext4-baseline.

On the ST4000LM016 DM-SMR drive, the benchmark completes 2× faster on ext4-lazy, as seen in Figure 3.13 (b), because the drive throughput is almost always higher than on ext4-baseline. With ext4-baseline, the drive enters a long period of cleaning with ultra-low throughput at 2,000th second and recovers around 4,200th second completing the benchmark with higher throughput.

We observe a similar phenomenon on the ST5000AS0011 DM-SMR drive, as shown in Figure 3.13 (c). Unlike with ext4-baseline that continues with a low throughput until the end of the run, with ext4-lazy the cleaning cycle eventually completes and the workload finishes 2× faster.

The last DM-SMR drive in our list, WD40NMZW model found in My Passport Ultra from Western Digital [197], shows a different behavior from previous disks, suggesting a different STL design. We think it is using an S-blocks-like architecture [29] with dynamic mapping that enables cheaper cleaning (§ 2.2). Unlike previous drives that clean only when idle or when the persistent cache is full, WD40NMZW seems to regularly mix cleaning with file system user work. Therefore, its throughput is not as high as the Seagate drives initially, but after the persistent cache becomes full, it does not suffer as sharp of a drop and its steady-state throughput is higher. Nevertheless, with ext4-lazy the disk achieves 1.4-2.5× increase in throughput over ext4-baseline, depending on the state of the persistent cache, and the benchmark completes 1.7× faster.

Figure 3.14 shows Postmark transaction throughput numbers for the runs. All of the drives show a significant improvement with ext4-lazy. An interesting observation is that, while with ext4-baseline WD40NMZW is 2× faster than ST8000AS0002, with ext4-lazy the situation is reversed and ST8000AS0002 is 2× faster than WD40NMZW, and fastest overall.

3.4.4 Performance Overhead of Ext4-Lazy

Indirection Overhead: To determine the overhead of in-memory jmap lookup, we populated jmap with 10,000 mappings pointing to random blocks in the journal, and measured the total time to read all of the blocks in a fixed random order. We then measured the time to read the same random blocks directly, skipping the jmap lookup, in the same order. We repeated each experiment five times, starting with a cold cache every time, and found no difference in total time read time—reading from disk dominated the total time of the operation.

Memory Overhead: A single jmap entry consists of a red-black tree node (3×8 bytes), a doubly-linked list node (2×8 bytes), a mapping (12 bytes), and a transaction id (4 bytes), occupying 56 bytes in memory. Hence, for example, a million-entry jmap that can map 3.8 GiB of hot metadata, requires 53 MiB of memory. Although this is already a modest overhead for today's systems, it can further be reduced with memory-efficient data structures.

Seek Overhead: The rationale for introducing cylinder groups in FFS, which manifest themselves as block groups in ext4, was to create clusters of inodes that are spread over the disk close to the blocks that they reference, to avoid long seeks between an inode and its associated data [129]. Ext4-lazy, however, puts hot metadata in the journal located at the center of the disk, requiring a half-seek to read a file in the worst case. The TarFiles benchmark (§ 3.4.2) shows that when reading files from a large and deep directory tree, where directory traversal time dominates, putting the metadata at the center wins slightly over spreading it out. To measure the seek overhead on a shallow directory, we created a directory with 10,000 small files located at the outer diameter of the disk on ext4-lazy, and starting with a cold cache creating the tarball of the directory. We observed that since files were created at the same time, their metadata was written sequentially to the journal. The code for reading metadata blocks in ext4 uses readahead since the introduction of flex_bgs. As a result, the metadata of all files was brought into the buffer cache in just 3 seeks. After five repetitions of the experiment on ext4-baseline an ext4-lazy, the average times were 103 seconds and 101 seconds, respectively.

Cleaning Overhead: In our benchmarks, the 10 GiB journal always contained less than 10% live metadata. Therefore, most of the time the cleaner reclaimed space simply by advancing the tail. We kept reducing the journal size and the first noticeable slowdown occurred with a journal size of 1.4 GiB, that is, when the live metadata was $\approx$ 70% of the journal.

3.5 Related Work

Researchers have tinkered with the idea of separating metadata from data and managing it differently in local file systems before. Like many other good ideas, it may have been ahead of its time because the technology that would benefit most from it did not exist yet, preventing adoption.

The Multi-Structured File System[133] (MFS) is the first file system proposing the separation of data and metadata. It was motivated by the observation that the file system I/O is becoming a bottleneck because data and metadata exert different access patterns on storage, and a single storage system cannot respond to these demands efficiently. Therefore, MFS puts data and metadata on isolated disk arrays, and for each data type it introduces on-disk structures optimized for the respective access pattern. Ext4-lazy differs from MFS in two ways: (1) it writes metadata as a log, whereas MFS overwrites metadata in-place; (2) facilitated by (1), ext4-lazy does not require a separate device for storing metadata in order to achieve performance improvements.

DualFS [145] is a file system influenced by MFS—it also separates data and metadata. Unlike MFS, however, DualFS uses well known data structures for managing each data type. Specifically, it combines an FFS-like [128] file system for managing data, and LFS-like [158] file system for managing metadata. hFS [224] improves on DualFS by also storing small files in a log along with metadata, thus exploiting disk bandwidth for small files. Similar to these file systems ext4-lazy separates metadata and data, but unlike them it does not confine metadata to a log—it uses a hybrid design where metadata can migrate back and forth between file system and log as needed. However, what really sets ext4-lazy apart is that it is not a new prototype file system; it is an evolution of a production file system, showing that a journaling file system can benefit from the metadata separation idea with a small set of changes that does not require on-disk format changes.

ESB [101] separates data and metadata on ext2, and puts them on a CMR drive and an SSD, respectively, to explore the effect of speeding up metadata operations on I/O performance. It is a virtual block device that sits below ext2 and leverages the fixed location of static metadata to forward metadata block requests to an SSD. The downside of this approach is that unlike ext4-lazy, it cannot handle floating metadata, like directory blocks. ESB authors conclude that for metadata-light workloads speeding up metadata operations will not improve I/O performance on a CMR drive, which aligns with our findings (§ 3.4.2).

A separate metadata server is the norm in distributed storage systems like Ceph, Lustre [209], and Panasas [205]. TableFS [156] extends the idea to a local file system: it is a FUSE-based [185] file system that stores metadata in LevelDB [73] and uses ext4 as an object store for large files. Unlike ext4-lazy, TableFS is disadvantaged by FUSE overhead, but still it achieves substantial speedup against production file systems on metadata-heavy workloads.

3.6 Summary

In this chapter we take an evolutionary approach to adapting general-purpose file systems to DM-SMR drives. Our work has three takeaways. First, it shows how effective a well-chosen small change can be. Second, it suggests that while three decades ago it was wise for file systems to scatter the metadata across the disk, today, with large memory sizes that cache metadata and with changing recording technology, putting metadata at the center of the disk and managing it as a log looks like a better choice. Third, it shows that we can reduce the block interface tax only so much using evolutionary changes to a mature file system: Although we improved throughput for write-heavy workloads on DM-SMR drives, the overhead of garbage collection was still significant.

It appears that the only option for avoiding the block interface tax is to take the revolutionary approach and design a new general-purpose file system for the zone interface. At this point, it is worth taking a step back and asking the following question: How appropriate are the abstractions provided by a general-purpose file system for a storage backend? Can a storage backend perform better if it bypasses the file system and runs on a raw device, and is it practical to do so? We answer these questions in the next chapter.

Chapter 4

Understanding and Quantifying the File System Tax in Ceph

In this chapter we study and quantify the file system tax—the overhead in code complexity, performance, and flexibility stemming from building a storage backend on top of a general-purpose file system—in Ceph, a widely used distributed storage system. To this end, we perform a longitudinal study of storage backend evolution in Ceph. We dissect reasons that impede building high-performance storage backends on top of general-purpose file systems and describe the design of BlueStore, a special-purpose storage backend.

4.1 The State of Current Storage Backends

Traditionally distributed storage systems have built their storage backends on top of general-purpose file systems, such as ext4 or XFS [74, 84, 87, 177, 202, 205][58, 154, 190, 209]. This approach has delivered a reasonable performance, precluding questions on the suitability of file systems as a distributed storage backend. Several reasons have contributed to the success of file systems as the storage backend. First, they allow delegating the hard problems of data persistence and block allocation to a well-tested and highly performant code. Second, they offer a familiar interface (POSIX) and abstractions (files, directories). Third, they enable the use of standard tools (`ls`, `find`) to explore disk contents.

The team behind the Ceph distributed storage system [202] also followed this convention for almost a decade. Hard-won lessons that the Ceph team learned using several popular file systems led them to question the fitness of file systems as storage backends. This is not surprising in hindsight. Stonebraker, after building the INGRES database for a decade, noted that "operating systems offer all things to all people at much higher overhead" [183]. Similarly, exokernels demonstrated that customizing abstractions to applications results in a significantly better performance [61, 100].

In 2015 the Ceph project started designing and implementing BlueStore, a user space storage backend that stores data directly on raw storage devices, and metadata in a key-value store. By taking full control of the I/O path, BlueStore has been able to efficiently implement full data checksums, inline compression, and fast overwrites of erasure-coded data, while also improving performance on common customer workloads. In 2017, after just two years of development, Blue-

Store became the default production storage backend in Ceph. A 2018 survey among Ceph users shows that 70% use BlueStore in production with hundreds of petabytes in deployed capacity [125].

Our first contribution in this work is outlining and explaining in detail the technical reasons behind Ceph's decision to develop BlueStore. The first reason is that it is hard to implement efficient transactions on top of existing file systems. Transaction support in the storage backend simplifies implementing strong consistency that many distributed storage systems provide [87, 202][154, 209]. A storage backend can seamlessly implement transactions if the backing file system already supports them [112, 162]. Yet, most file systems implement the POSIX standard, which lacks a transaction concept.

A significant body of work aims to introduce transactions into file systems [85, 131, 137, 150, 162, 170, 180, 217], but none of these approaches have been adopted due to their high performance overhead, limited functionality, interface complexity, or implementation complexity. Therefore, distributed storage system developers typically resort to using inefficient or complex mechanisms, such as implementing a Write-Ahead Log (WAL) on top of a file system [154], or leveraging a file system's internal transaction mechanism [209]. The experience of the Ceph team shows that these options deliver subpar performance or result in a fragile system.

The second reason is that the file system's metadata does not scale, yet its performance can significantly affect the performance of the distributed storage system as a whole. Inability to efficiently enumerate large directory contents or handle small files at scale in file systems can cripple performance for both centralized [205][209] and distributed [202][154] metadata management designs. To address this problem, distributed storage system developers use metadata caching [154], deep directory hierarchies arranged by data hashes [202], custom databases [182], or patches to file systems [20, 21, 226]. The specific challenge that the Ceph team faced was enumerating directories with millions of entries fast, and the lack of ordering in the returned result. Both Btrfs and XFS-based backends suffered from this problem, and directory splitting operations meant to distribute the metadata load were found to clash with file system policies, crippling overall system performance.

Our second contribution, is to introduce the design of BlueStore, the challenges its design overcomes, and opportunities for future improvements. Novelties of BlueStore include (1) storing low-level file system metadata, such as extent bitmaps, in a key-value store, thereby avoiding on-disk format changes and reducing implementation complexity; (2) optimizing clone operations and minimizing the overhead of the resulting extent reference-counting through careful interface design; (3) BlueFS—a user space file system that enables RocksDB to run faster on a raw storage device; and (4) a space allocator with a fixed 35 MiB memory usage per terabyte of disk space.

Finally, perform several experiments that evaluate the improvement of design changes from Ceph's previous production backend, FileStore, to BlueStore. We experimentally measure the performance effect of issues like the overhead of journaling file systems, double writes to the journal, inefficient directory splitting, and update-in-place mechanisms (as opposed to copy-on-write).

4.2 Background on the Ceph Distributed Storage System

Figure 4.1 shows the high-level architecture of Ceph. At the core of Ceph is the Reliable Autonomic Distributed Object Store (RADOS) service [204]. RADOS scales to thousands of Object Storage

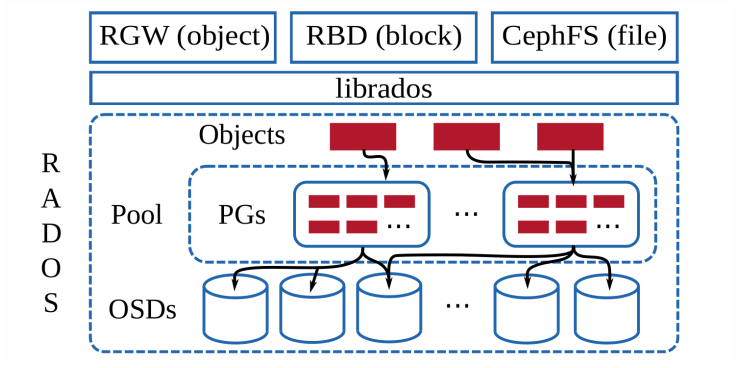


Figure 4.1: High-level depiction of Ceph's architecture. A single pool with 3× replication is shown. Therefore, each placement group (PG) is replicated on three OSDs.

Devices (OSDs), providing self-healing, self-managing, replicated object storage with strong consistency. Ceph's `librados` library provides a transactional interface for manipulating objects and object collections in RADOS. Out of the box, Ceph provides three services implemented using `librados`: the RADOS Gateway (RGW), an object storage similar to Amazon S3 [8]; the RADOS Block Device (RBD), a virtual block device similar to Amazon EBS [7]; and CephFS, a distributed file system with POSIX semantics.

Objects in RADOS are stored in logical partitions called *pools*. Pools can be configured to provide redundancy for the contained objects either through replication or erasure coding. Within a pool, the objects are sharded among aggregation units called *placement groups* (PGs). Depending on the replication factor, PGs are mapped to multiple OSDs using CRUSH, a pseudo-random data distribution algorithm [203]. Clients also use CRUSH to determine the OSD that should contain a given object, obviating the need for a centralized metadata service. PGs and CRUSH form an indirection layer between clients and OSDs that allows the migration of objects between OSDs to adapt to cluster or workload changes.

In every node of a RADOS cluster, there is a separate *Ceph OSD* daemon per local storage device. Each OSD processes client I/O requests from `librados` clients and cooperates with peer OSDs to replicate or erasure-code updates, migrate data, or recover from failures. Data is persisted to the local device via the internal *ObjectStore* interface, which provides abstractions for objects, object collections, a set of primitives to inspect data, and transactions to update data. A transaction combines an arbitrary number of primitives operating on objects and object collections into an atomic operation. In principle each OSD may run a different backend implementing the ObjectStore interface, although in practice clusters tend to run the same backend implementation.

4.2.1 Evolution of Storage Backends in Ceph

The first implementation of the ObjectStore interface was in fact a user space file system called Extent and B-Tree-based Object File System (EBOFS). In 2008, Btrfs was emerging with attractive features such as transactions, deduplication, checksums, and transparent compression, which were lacking in EBOFS. Therefore, as shown in Figure 4.2, EBOFS was replaced by FileStore—an ObjectStore implementation on top of Btrfs.

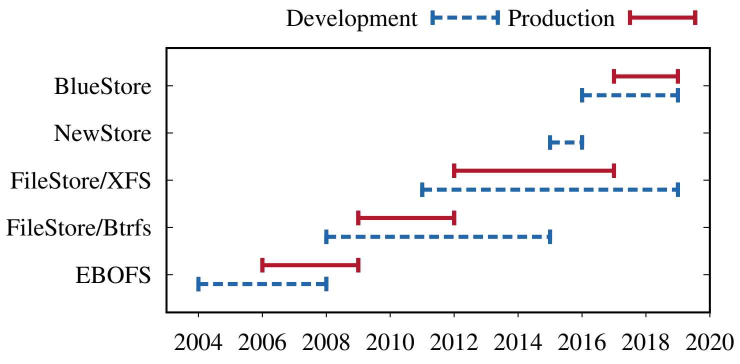


Figure 4.2: Timeline of storage backend evolution in Ceph. For each backend, the period of development, and the period of being the default production backend is shown.

In FileStore, an object collection is mapped to a directory and object data is stored in a file. Initially, object attributes were stored in POSIX extended file attributes (`xattrs`), but were later moved to LevelDB when object attributes exceeded size or count limitations of `xattrs`. FileStore on Btrfs was the production backend for several years, throughout which Btrfs remained unstable and suffered from severe data and metadata fragmentation. In the meantime, the ObjectStore interface evolved significantly, making it impractical to switch back to EBOFS. Instead, FileStore was ported to run on top of XFS, ext4, and later ZFS. Of these, FileStore on XFS became the de facto backend because it scaled better and had faster metadata performance [82].

While FileStore on XFS was stable, it still suffered from metadata fragmentation and did not exploit the full potential of the hardware. Lack of native transactions led to a user space WAL implementation that performed full data journaling and capped the speed of read-modify-write workloads—a typical Ceph workload—to the WAL's write speed. In addition, since XFS was not a copy-on-write file system, clone operations used heavily by snapshots were significantly slower.

NewStore was the first attempt at solving the metadata problems of file-system-based backends. Instead of using directories to represent object collections, NewStore stored object metadata in RocksDB, an ordered key-value store, while object data was kept in files. RocksDB was also used to implement the WAL, making read-modify-write workloads efficient due to a combined data and metadata log. Storing object data as files and running RocksDB on top of a journaling file system, however, introduced high consistency overhead. This led to the implementation of BlueStore, which used raw disks. The following section describes the challenges BlueStore aimed to resolve. A complete description of BlueStore is given in § 4.4.

4.3 Building Storage Backends on Local File Systems is Hard

This section describes the challenges faced by the Ceph team while trying to build a distributed storage backend on top of local file systems.

4.3.1 Challenge 1: Efficient Transactions

Transactions simplify application development by encapsulating a sequence of operations into a single atomic unit of work. Thus, a significant body of work aims to introduce transactions into file systems [85, 131, 137, 150, 162, 170, 180, 217]. None of these works have been adopted by production file systems, however, due to their high performance overhead, limited functionality, interface complexity, or implementation complexity.

Hence, there are three tangible options for providing transactions in a storage backend running on top of a file system: (1) hooking into a file system's internal (but limited) transaction mechanism; (2) implementing a WAL in user space; and (3) using key-value database with transactions as a WAL. Next, we describe why each of these options results in significant performance or complexity overhead.

Leveraging File System Internal Transactions

Many file systems implement an in-kernel transaction framework that enables performing compound internal operations atomically [34, 43, 179, 194]. Since the purpose of this framework is to ensure internal file system consistency, its functionality is generally limited, and thus, unavailable to users. For example, a rollback mechanism is not available in file system transaction frameworks because it is unnecessary for ensuring internal consistency of a file system.

Until recently, Btrfs was making its internal transaction mechanism available to users through a pair of system calls that atomically applied operations between them to the file system [43]. The first version of FileStore that ran on Btrfs relied on these system calls, and suffered from the lack of a rollback mechanism. More specifically, if a Ceph OSD ecountered a fatal event in the middle of a transaction, such as a software crash or a `KILL` signal, Btrfs would commit a partial transaction and leave the storage backend in an inconsistent state.

Solutions attempted by the Ceph and Btrfs teams included introducing a single system call for specifying the entire transaction [198], and implementing rollback through snapshots [199], both of which proved costly. Btrfs authors recently deprecated transaction system calls [26]. This outcome is similar to Microsoft's attempt to leverage NTFS's in-kernel transaction framework for providing an atomic file transaction API, which was deprecated due to its high barrier to entry [104].

These experiences strongly suggest that it is hard to leverage the internal transaction mechanism of a file system in a storage backend implemented in user space.

Implementing the WAL in User Space

An alternative to utilizing the file system's in-kernel transaction framework was to implement a logical WAL in user space. While this approach worked, it suffered from three major problems.

Slow Read-Modify-Write. Typical Ceph workloads perform many read-modify-write operations on objects, where preparing the next transaction requires reading the effect of the previous transaction. A user space WAL implementation, on the other hand, performs three steps for every transaction. First, the transaction is serialized and written to the log. Second, `fsync` is called to commit the transaction to disk. Third, the operations specified in the transaction are applied to the file system. The effect of a transaction cannot be read by upcoming transactions until the third step completes, which is dependent on the second step. As a result, every read-modify-write operation incurred the full latency of the WAL commit, preventing efficient pipelining.

Non-idempotent Operations. In FileStore, objects are represented by files, and collections are mapped to directories. With this data model, replaying a logical WAL after a crash is challenging due to non-idempotent operations. While the WAL is trimmed periodically, there is always a window of time when a committed transaction that is still in the WAL has already been applied to the file system. For example, consider a transaction consisting of three operations: ① `clone a→b`; ② `update a`; ③ `update c`. If a crash happens after the second operation, replaying the WAL corrupts object `b`. As another example, consider a transaction: ① `update b`; ② `rename b→c`; ③ `rename a→b`; ④ `update d`. If a crash happens after the third operation, replaying the WAL corrupts object `a`, which is now named `b`, and then fails because object `a` does not exist anymore.

FileStore on Btrfs solved this problem by periodically taking persistent snapshots of the file system and marking the WAL position at the time of snapshot. Then on recovery the latest snapshot was restored, and the WAL was replayed from the position marked at the time of the snapshot.

When FileStore abandoned Btrfs in favor of XFS (§ 4.2.1), the lack of efficient snapshots caused two problems. First, on XFS the `sync` system call is the only option for synchronizing file system state to storage. However, in typical deployments with multiple drives per node, `sync` is too expensive because it synchronizes all file systems on all drives. This problem was resolved by adding `syncfs` system call [200] to the Linux kernel, which synchronizes only a given file system.

The second problem was that with XFS, there is no option to restore a file system to a specific state after which the WAL can be replayed without worrying about non-idempotent operations. To address this problem, guards (sequence numbers) were added to avoid replaying non-idempotent operations. The guards were hard to reason about, hard to test, and slowed operations down. Verifying correctness of guards for complex operations was hard due to the large problem space. Tooling was written to generate random permutations of complex operation sequences, which was combined with failure injection to semi-comprehensively verify that all failure cases were correctly handled, but the code ended up fragile and hard-to-maintain.

Double Writes. The final problem with the WAL in FileStore is that it writes data twice: first to the WAL, and then to the file system, effectively halving the disk bandwidth on write-intensive workloads. This is a well-known problem that leads most file systems to only log metadata changes, allowing data to be lost after a crash. It is possible to avoid the penalty of double writes for new data, by first writing it to disk and then logging only the respective metadata. However, FileStore's approach of using the state of the file system to infer the namespace of objects and their states makes this method hard to use due to corner cases, such as partially written or temporary files. While FileStore's approach turned out to be problematic, it was originally chosen for a technically useful reason: the alternative required implementing an in-memory cache for data and metadata to any updates waiting on the FileStore journal, despite the kernel having a page and inode cache of its own.

Using a Key-Value Store as the WAL

With NewStore, the metadata was stored in RocksDB, an ordered key-value store, while the object data were still represented as files in a file system. Hence, metadata operations could be performed atomically; data overwrites, however, were logged into RocksDB and executed later. We first describe how this design addresses the three problems of a logical WAL, and then show that it introduces high consistency overhead that stems from running atop a journaling file system.

First, slow read-modify-write operations are avoided because the key-value interface allows reading the new state of an object without waiting for the transaction to commit.

Second, the problem of non-idempotent operation replay is avoided because the read side of such operations is resolved at the time when the transaction is prepared. For example, for `clone a→b`, if object `a` is small, it is copied and inserted into the transaction; if object `a` is large, a copy-on-write mechanism is used, which changes both `a` and `b` to point to the same data and marks the data read-only.

Finally, the problem of double writes is avoided for new objects because the object namespace is now decoupled from the file system state. Therefore, data for a new object is first written to the file system and then a reference to it is atomically added to the database.

Despite these favorable properties, the combination of RocksDB and a journaling file system introduces high consistency overhead, similar to the *journaling of journal* problem [97, 174]. Creating an object in NewStore entails two steps: (1) writing to a file and calling `fsync`, and (2) writing the object metadata to RocksDB synchronously [92], which also calls `fsync`. Ideally, the `fsync` in each step should issue one expensive `FLUSH CACHE` command [213] to disk. With a journaling file system, however, each `fsync` issues two flush commands: after writing the data, and after committing the corresponding metadata changes to the file system journal. Hence, creating an object in NewStore results in four expensive flush commands to disk.

We demonstrate the overhead of journaling using a benchmark that emulates a storage backend creating many objects. The benchmark has a loop in which each iteration first writes 0.5 MiB of data and then inserts a 500 byte metadata to RocksDB. We run the benchmark on two setups. The first setup emulates NewStore, issuing four flush operations for every object creation: data is written as a file to XFS, and the metadata is inserted to stock RocksDB running on XFS. The second setup emulates object creation on raw disk, which issues two flush operations for every object creation: data is written to the raw disk and the metadata is inserted to a modified RocksDB that runs on a raw disk with a preallocated pool of WAL files.

Figure 4.3 shows that the object creation throughput is 80% higher on raw disk than on XFS when running on a HDD and 70% when running on an NVMe SSD.

4.3.2 Challenge 2: Fast Metadata Operations

Inefficiency of metadata operations in file systems is a source of constant struggle for distributed storage systems [143][149, 226]. One of the key metadata challenges in Ceph with the FileStore backend stems from the slow directory enumeration operations (`readdir`) on large directories, and the lack of ordering in the returned result [178].

Objects in RADOS are mapped to a PG based on a hash of their name and enumerated by hash order. Enumeration is necessary for operations like scrubbing [163], recovery, or for serving

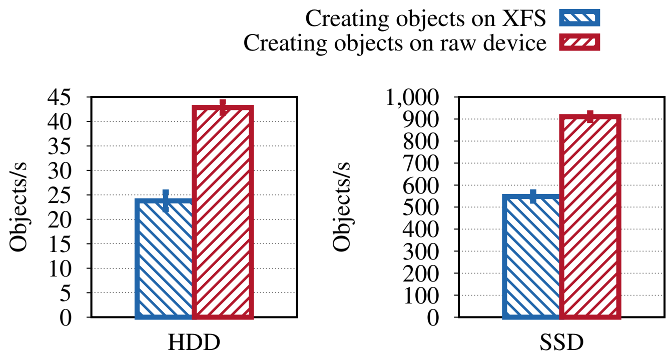


Figure 4.3: The overhead of running an object store workload on a journaling file system. Object creation throughput is 80% higher on a raw HDD (4 TB Seagate ST4000NM0023) and 70% higher on a raw NVMe SSD (400 GB Intel P3600).

`librados` calls that list objects. For objects with long names—as is often the case with RGW—FileStore works around the file name length limitation in local file systems using extended attributes, which may require a `stat` call to determine the object name. FileStore follows a commonly-adopted solution to the slow enumeration problem: a directory hierarchy with large fan-out is created, objects are distributed among directories, and then selected directories' contents are sorted after being read.

To sort them quickly and to limit the overhead of potential `stat` calls, directories are kept small (a few hundred entries) by splitting them when the number of entries within them grows. The process can be costly. While XFS tries to allocate the directory and its contents in the same *allocation group* [93], subdirectories are typically placed in different allocation groups to ensure there is space for future directory entries to be located close together [128]. Therefore, as the number of objects grows, directory contents spread out, and split operations take longer to complete. This can cripple performance if all Ceph OSDs start splitting in unison, and it has been affecting many Ceph users over the years [27, 52, 181].

To demonstrate this effect, we configure a 16-node Ceph cluster (§ 4.6) with roughly half the recommended number of PGs to increase load per PG and accelerate splitting, and we insert millions of 4 KiB objects with queue depth of 128 at the RADOS layer (§ 4.2). Figure 4.4 shows the effect of the splitting on FileStore for an all-SSD cluster. While the first split is not noticeable in the graph, the second split causes a precipitous drop that kills the throughput for 7 minutes on an all-SSD cluster and for 120 minutes on an all-HDD cluster (not shown), during which a large and deep directory hierarchy with millions of entries is scanned and even a deeper hierarchy is created. Splitting can dramatically affect workload performance on both HDDs and NVMe SSDs since it happens inline with workload operations, and it can spawn enough I/O to even throttle SSDs that can handle significant operation parallelism.

4.3.3 Other Challenges

Many public and private clouds rely on distributed storage systems like Ceph for providing storage services [139]. Without complete control of the I/O stack, it is hard for distributed storage systems

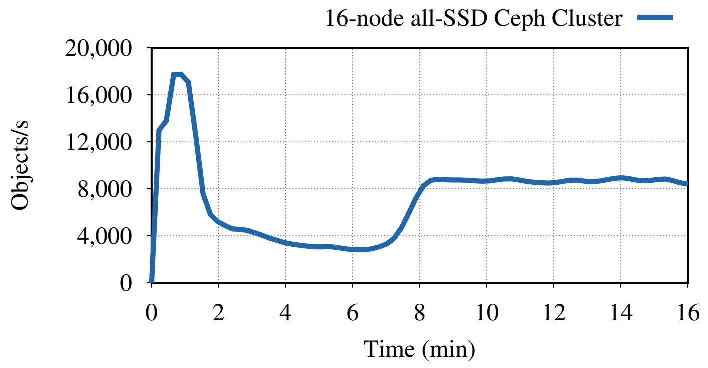


Figure 4.4: The effect of directory splitting on throughput with FileStore backend. The workload inserts 4 KiB objects using 128 parallel threads at the RADOS layer to a 16-node Ceph cluster (setup explained in § 4.6). Directory splitting brings down the throughput for 7 minutes on an all-SSD cluster. Once the splitting is complete, the throughput recovers but does not return to peak, due to combination of deeper nesting of directories, increased size of the underlying file system, and an imperfect implementation of the directory hashing code in FileStore.

to enforce storage latency SLOs in these clouds. And yet, running the storage backend on top of a file system cedes the control of the I/O stack to the operating system policies and mechanisms.

One cause of high-variance request latencies in file-system-based storage backends is the operating system page cache. To improve user experience, most OSs implement the page cache using write-back policy, in which a write operation completes once the data is buffered in memory and the corresponding pages are marked as *dirty*. On a system with little I/O activity, the dirty pages are written back to disk at regular intervals, synchronizing the on-disk and in-memory copies of data. On a busy system, on the other hand, the write-back behavior is governed by a complex set of policies that can trigger writes at arbitrary times [14, 44, 220].

Hence, while the write-back policy results in a responsive system for users with lightly loaded machines, it complicates achieving predictable latency on busy storage backends. Even with a periodic use of `fsync`, FileStore has been unable to bound the amount of deferred inode metadata write-back, leading to inconsistent performance.

Another challenge for file-system-based backends is implementing operations that work better with copy-on-write support, such as snapshots. If the backing file system is copy-on-write, these operations can be implemented efficiently. However, even if the copy-on-write is supported, a file system may have other drawbacks, like fragmentation in FileStore on Btrfs (§ 4.2.1). If the backing file system is not copy-on-write, then these operations require performing expensive full copies of objects, which makes snapshots and overwriting of erasure-coded data prohibitively expensive in FileStore (§ 4.5.2).

4.4 BlueStore: A Clean-Slate Approach

BlueStore is a storage backend designed from scratch to replace backends that relied on file systems and faced the challenges outlined in the previous section. Some of the main goals of BlueStore

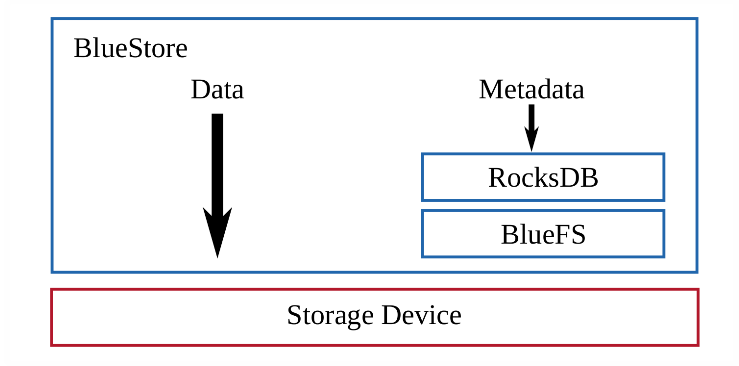


Figure 4.5: The high-level architecture of BlueStore. Data is written to the raw storage device using direct I/O. Metadata is written to RocksDB running on top of BlueFS. BlueFS is a user space library file system designed for RocksDB, and it also runs on top of the raw storage device.

were:

1. Fast metadata operations (§ 4.4.1)
2. No consistency overhead for object writes (§ 4.4.1)
3. Copy-on-write clone operation (§ 4.4.2)
4. No journaling double-writes (§ 4.4.2)
5. Optimized I/O patterns for HDD and SSD (§ 4.4.2)

BlueStore achieved all of these goals within just two years and became the default storage backend in Ceph. Two factors played a key role in why BlueStore matured so quickly compared to general-purpose POSIX file systems that take a decade to mature [210, 211][60, 116]. First, BlueStore implements a small, special-purpose interface, and not a complete POSIX I/O specification. Second, BlueStore is implemented in user space, which allows it to leverage well-tested and high-performance third-party libraries. Finally, BlueStore's control of the I/O stack enables additional features whose discussion we defer to § 4.5.

The high-level architecture of BlueStore is shown in Figure 4.5. BlueStore runs directly on raw disks. A space allocator within BlueStore determines the location of new data, which is asynchronously written to disk using direct I/O. Internal metadata and user object metadata is stored in RocksDB, which runs on BlueFS, a minimal user space file system tailored to RocksDB. The BlueStore space allocator and BlueFS share the disk and periodically communicate to balance free space. The remainder of the section describes metadata and data management in BlueStore.

4.4.1 BlueFS and RocksDB

BlueStore achieves its first goal, **fast metadata operations**, by storing metadata in RocksDB. BlueStore achieves its second goal of **no consistency overhead** with two changes. First, it writes data directly to raw disk, resulting in one cache flush for data write. Second, it changes RocksDB to reuse WAL files as a circular buffer, resulting in one cache flush for metadata write, a feature that RocksDB has now officially adopted.

RocksDB itself runs on BlueFS, a minimal file system designed specifically for RocksDB that

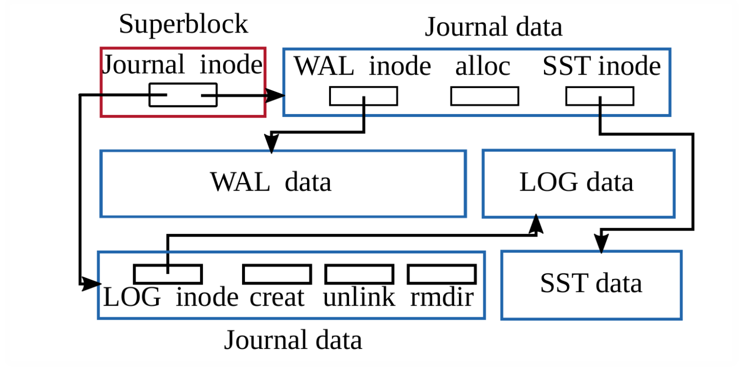


Figure 4.6: A possible on-disk data layout of BlueFS. The metadata in BlueFS lives only in the journal. The journal does not have a fixed location—its extents are interleaved with file data. The WAL, LOG, and SST files are write-ahead log file, debug log file, and a sorted-string table files, respectively, generated by RocksDB.

runs on a raw storage device. RocksDB abstracts its requirements out from the underlying file system in the *Env* interface. BlueFS is an implementation of this interface in the form of a user space, extent-based, and journaling file system. It implements basic system calls required by RocksDB, such as `open`, `mkdir`, and `pwrite`. A possible on-disk layout of BlueFS is shown in Figure 4.6. BlueFS maintains an inode for each file that includes the list of extents allocated to the file. The superblock is stored at a fixed offset and contains an inode for the journal. The journal has the only copy of all file system metadata, which is loaded into memory at mount time. On every metadata operation, such as directory creation, file creation, and extent allocation, the journal and in-memory metadata are updated. The journal is not stored at a fixed location; its extents are interleaved with other file extents. The journal is compacted and written to a new location when it reaches a preconfigured size, and the new location is recorded in the superblock. These design decisions work because large files and periodic compactions limit the volume of metadata at any point in time.

Metadata Organization. BlueStore keeps multiple namespaces in RocksDB, each storing a different type of metadata. For example, object information is stored in the *O* namespace (that is, RocksDB keys start with *O* and their values represent object metadata), block allocation metadata is stored in the *B* namespace, and collection metadata is stored in the *C* namespace. Each collection maps to a PG and represents a shard of a pool's namespace. The collection name includes the pool identifier and a prefix shared by the collection's object names. For example, a key-value pair `C12.e4-6` identifies a collection in pool 12 with objects that have hash values starting with the 6 significant bits of `e4`. Hence, the object `O12.e532` is a member, whereas the object `O12.e832` is not. This organization allows a collection of millions of objects to be split into multiple collections merely by changing the number of significant bits. This collection splitting operation is necessary to rebalance data across OSDs when, for example, a new OSD is added to the cluster, and it was a costly operation with FileStore.

4.4.2 Data Path and Space Allocation

BlueStore is a copy-on-write backend. For incoming writes larger than a *minimum allocation size* (64 KiB for HDDs, 16 KiB for SSDs) the data is written to a newly allocated extent. Once the data is persisted, the corresponding metadata is inserted to RocksDB. This allows BlueStore to provide **efficient clone** operations. A clone operation simply increments the reference count of dependent extents, and writes are directed to new extents. It also allows BlueStore to **avoid journal double-writes** for object writes and partial overwrites that are larger than the minimum allocation size.

For writes smaller than the minimum allocation size, both data and metadata are first inserted to RocksDB as promises of future I/O, and then asynchronously written to disk after the transaction commits. This deferred write mechanism has two purposes. First, it batches small writes to increase efficiency, because new data writes require two I/O operations whereas an insert to RocksDB requires one. Second, it **optimizes I/O based on the device type.** 64 KiB (or smaller) overwrites of a large object on an HDD are performed asynchronously in place to avoid seeks during reads, whereas on SSDs in-place overwrites only happen for I/O sizes less than 16 KiB.

Space Allocation. BlueStore allocates space using two modules: the FreeList manager and the Allocator. The FreeList manager is responsible for a persistent representation of the parts of the disk currently in use. Like all metadata in BlueStore, this free list is also stored in RocksDB. The first implementation of the FreeList manager represented in-use regions as key-value pairs with offset and length. The disadvantage of this approach was that the transactions had to be serialized: the old key had to be deleted first before inserting a new key to avoid an inconsistent free list. The second implementation is bitmap-based. Allocation and deallocation operations use RocksDB's merge operator to flip bits corresponding to the affected blocks, eliminating the ordering constraint. The merge operator in RocksDB performs a deferred atomic read-modify-write operation that does not change the semantics and avoids the cost of point queries [91].

The Allocator is responsible for allocating space for the new data. It keeps a copy of the free list in memory and informs the FreeList manager as allocations are made. The first implementation of Allocator was extent-based, dividing the free extents into power-of-two size bins. This design was susceptible to fragmentation as disk usage increased. The second implementation uses a hierarchy of indexes layered on top of a single-bit-per-block representation to track whole regions of blocks. Large and small extents can be efficiently found by querying the higher and lower indexes, respectively. This implementation has a fixed memory usage of 35 MiB per terabyte of capacity.

Cache. Since BlueStore is implemented in user space and accesses the disk using direct I/O, it cannot use the operating system page cache. As a result, BlueStore implements its own write-through cache in user space, using the scan resistant 2Q algorithm [99]. The cache implementation is sharded for parallelism. It uses an identical sharding scheme to Ceph OSDs, which shard requests to collections across multiple cores. This avoids false sharing, so that the same CPU context processing a given client request touches the corresponding 2Q data structures.

4.5 Features Enabled by BlueStore

In this section we describe new features implemented in BlueStore. These features were previously lacking because implementing them efficiently requires full control of the I/O stack.

4.5.1 Space-Efficient Checksums

Ceph scrubs metadata every day and data every week. Even with scrubbing, however, if the data is inconsistent across replicas it is hard to be sure which copy is corrupt. Therefore, checksums are indispensable for distributed storage systems that regularly deal with petabytes of data, where bit flips are almost certain to occur.

Most local file systems do not support checksums. When they do, like Btrfs, the checksum is computed over 4 KiB blocks to make block overwrites possible. For 10 TiB of data, storing 32-bit checksums of 4 KiB blocks results in 10 GiB of checksum metadata. This makes it difficult to cache checksums in memory for fast verification.

On the other hand, most of the data stored in distributed storage systems is read-only and can be checksummed at a larger granularity. BlueStore computes a checksum for every write and verifies the checksum on every read. While multiple checksum algorithms are supported, `crc32c` is used by default because it is well-optimized on both x86 and ARM architectures, and it is sufficient for detecting random bit errors. With full control of the I/O stack, BlueStore can choose the checksum block size based on the I/O hints. For example, if the hints indicate that writes are from the S3-compatible RGW service, then the objects are read-only and the checksum can be computed over 128 KiB blocks, and if the hints indicate that objects are to be compressed, then a checksum can be computed after the compression, significantly reducing the total size of checksum metadata.

4.5.2 Overwrite of Erasure Coded Data

Ceph has supported erasure coded (EC) pools (§ 4.2) through the FileStore backend since 2014. However, until BlueStore, EC pools only supported object appends and deletions; overwrites were slow enough to make the system unusable. As a result, the use of EC pools were limited to RGW; for RBD and CephFS only replicated pools could be used.

To avoid the "RAID write hole" problem [189], where crashing during a multi-step data update can leave the system in an inconsistent state, Ceph performs overwrites in EC pools using two-phase commit. First, all OSDs that store a chunk of the EC object make a copy of the chunk so that they can roll back in case of failure. After all of the OSDs receive the new content and overwrite their chunks, the old copies are discarded. With FileStore on XFS, the first phase is expensive because each OSD performs a physical copy of its chunk. BlueStore, however, makes overwrites practical because its copy-on-write mechanism avoids full physical copies.

4.5.3 Transparent Compression

Transparent compression is crucial for large-scale distributed storage systems because 3× replication increases storage costs [69, 86]. BlueStore implements transparent compression where written data is automatically compressed before being stored.

Getting the full benefit of compression requires compressing over large 128 KiB chunks, and compression works well when objects are written in their entirety. For partial overwrites of a compressed object, BlueStore places the new data in a separate location and updates metadata to point to it. When the compressed object gets too fragmented due to multiple overwrites, BlueStore

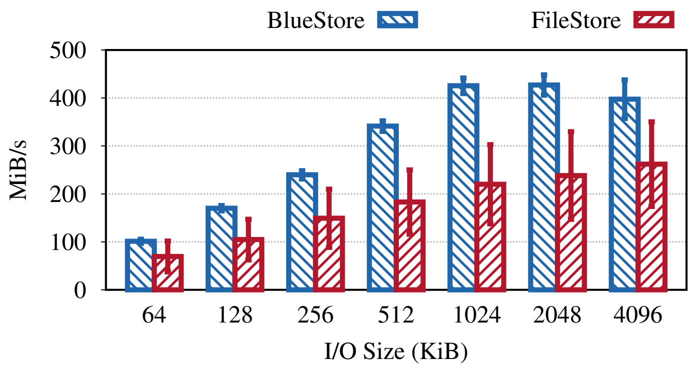


Figure 4.7: Throughput of steady state object writes to RADOS on a 16-node all-HDD cluster with different I/O sizes using 128 threads. Compared to FileStore, the throughput is 50-100% greater on BlueStore and has a significantly lower variance.

compacts the object by reading and rewriting. In practice, however, BlueStore uses hints and simple heuristics to compress only those objects that are unlikely to experience many overwrites.

4.6 Evaluation

This section compares the performance of a Ceph cluster using FileStore, a backend built on a file system, and BlueStore, a backend using the storage device directly. We first compare the throughput of object writes to the RADOS distributed object storage (§ 4.6.1). Then, we compare the end-to-end throughput of random writes, sequential writes, and sequential reads to RBD, the Ceph virtual block device built on RADOS (§ 4.6.2). Finally, we compare the throughput of random writes to an RBD device allocated on an erasure-coded pool (§ 4.6.3).

We run all experiments on a 16-node Ceph cluster connected with a Cisco Nexus 3264-Q 64-port QSFP+ 40GbE switch. Each node has a 16-core Intel E5-2698Bv3 Xeon 2GHz CPU, 64GiB RAM, 400GB Intel P3600 NVMe SSD, 4TB 7200RPM Seagate ST4000NM0023 HDD, and a Mellanox MCX314A-BCCT 40GbE NIC. All nodes run Linux kernel 4.15 on the Ubuntu 18.04 distribution and the Luminous release (v12.2.11) of Ceph. We use the default Ceph configuration parameters.

4.6.1 Bare RADOS Benchmarks

We start by comparing the performance of object writes to RADOS when using the FileStore and BlueStore backends. We focus on write performance improvements because most BlueStore optimizations affect writes.

Figure 4.7 shows the throughput for different object sizes written with a queue depth of 128. At the steady state, the throughput on BlueStore is 50-100% greater than FileStore. The throughput improvement on BlueStore stems from avoiding double writes (§ 4.3.1) and consistency overhead of a journaling file system (§ 4.3.1).

Figure 4.8 shows the 95th and above percentile latencies of object writes to RADOS. BlueStore

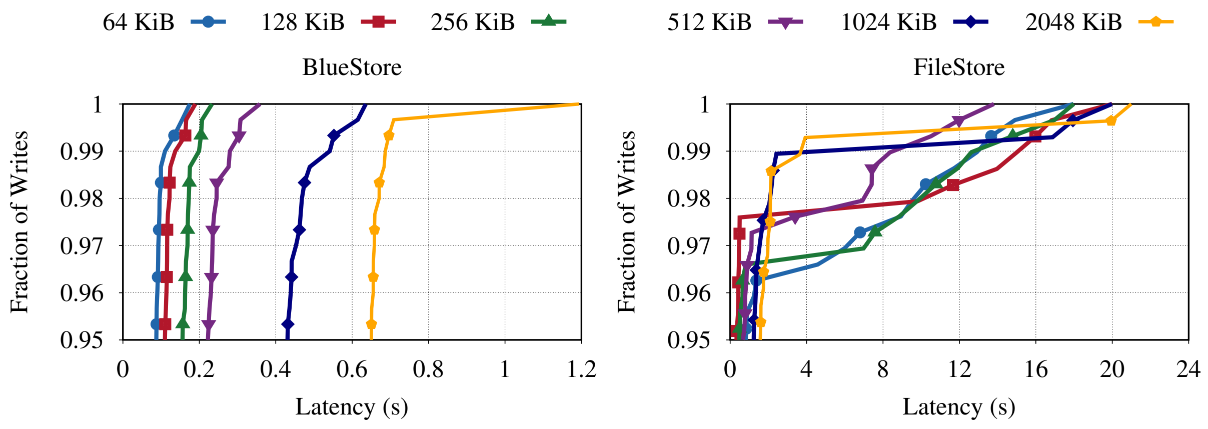


Figure 4.8: 95th and above percentile latencies of object writes to RADOS on a 16-node all-HDD cluster with different sizes using 128 threads. BlueStore has an order of magnitude lower tail latency than FileStore.

has an order of magnitude lower tail latency than FileStore. In addition, with BlueStore the tail latency increases with the object size, as expected, whereas with FileStore even small-sized object writes may have high tail latency, stemming from the lack of control over writes (§ 4.3.3).

The read performance on BlueStore (not shown) is similar or better than on FileStore for I/O sizes larger than 128 KiB; for smaller I/O sizes FileStore is better because of the kernel read-ahead [12]. BlueStore does not implement read-ahead on purpose. It is expected that the applications implemented on top of RADOS will perform their own read-ahead.

BlueStore eliminates the directory splitting effect of FileStore by storing metadata in an ordered key-value store. To demonstrate this, we repeat the experiment that showed the splitting problem in FileStore (§ 4.3.2) on an identically configured Ceph cluster using BlueStore backend. Figure 4.9 shows that the throughput on BlueStore does not suffer the precipitous drop, and in the steady state it is 2× higher than FileStore throughput on SSD (and 3× higher than FileStore throughput on HDD—not shown). Still, the throughput on BlueStore drops significantly before reaching a steady state due to RocksDB compaction whose cost grows with the object corpus.

4.6.2 RADOS Block Device (RBD) Benchmarks

Next, we compare the performance of RBD, a virtual block device service implemented on top of RADOS, when using the BlueStore and FileStore backends. RBD is implemented as a kernel module that exports a block device to the user, which can be formatted and mounted like a regular block device. Data written to the device is striped into 4 MiB RADOS objects and written in parallel to multiple OSDs over the network.

For RBD benchmarks we create a 1 TB virtual block device, format it with XFS, and mount it on the client. We use `fio` [13] to perform sequential and random I/O with queue depth of 256 and I/O sizes ranging from 4 KiB to 4 MiB. For each test, we write about 30 GiB of data. Before starting every experiment, we drop the operating system page cache for FileStore, and we restart OSDs for BlueStore to eliminate caching effects in read experiments. We first run all the experiments on a Ceph cluster installed with FileStore backend. We then tear down the cluster, reinstall it with

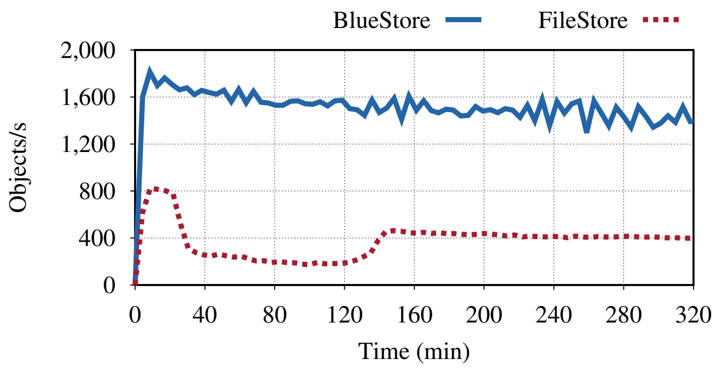


Figure 4.9: Throughput of 4 KiB RADOS object writes with queue depth of 128 on a 16-node all-SSD cluster. At steady state, BlueStore is 2× faster than FileStore. BlueStore does not suffer from directory splitting. However, its throughput is gradually brought down by the RocksDB compaction overhead.

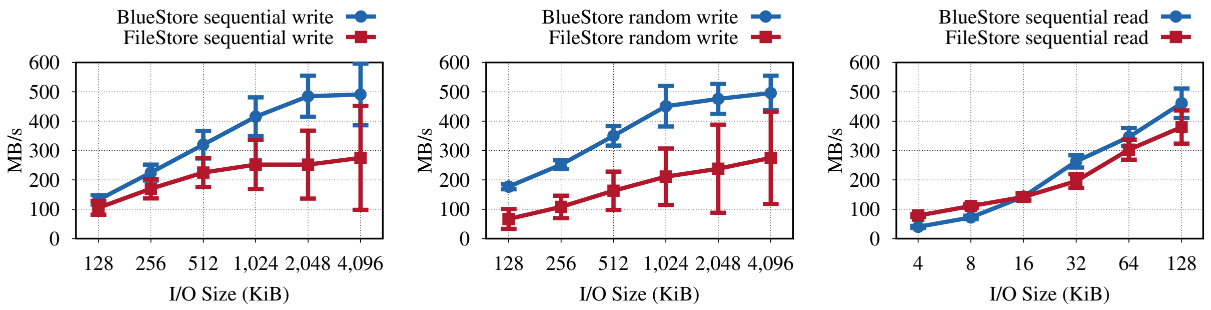


Figure 4.10: Sequential write, random write, and sequential read throughput with different I/O sizes and queue depth of 256 on a 1 TB Ceph virtual block device (RBD) allocated on a 16-node all-HDD cluster. Results for an all-SSD cluster were similar.

BlueStore backend, and repeat all the experiments.

Figure 4.10 shows the results for sequential writes, random writes, and sequential reads. For I/O sizes larger than 512 KiB, sequential and random write throughput is on average 1.7× and 2× higher with BlueStore, respectively, again mainly due to avoiding double-writes. BlueStore also displays a significantly lower throughput variance because it can deterministically push data to disk. In FileStore, on the other hand, arbitrarily-triggered metadata writeback (§ 4.3.3) conflicts with the foreground writes to the WAL and introduces long request latencies.

For medium I/O sizes (128–512 KiB) the throughput difference decreases for sequential writes because XFS masks out part of the cost of double writes in FileStore. With medium I/O sizes the writes to WAL do not fully utilize the disk. This leaves enough bandwidth for another write stream to go through and not have a large impact on the foreground writes to WAL. After writing the data synchronously to the WAL, FileStore then asynchronously writes it to the file system. XFS buffers these asynchronous writes and turns them into one large sequential write before issuing to disk. XFS cannot do the same for random writes, which is why the high throughput difference continues even for medium-sized random writes.

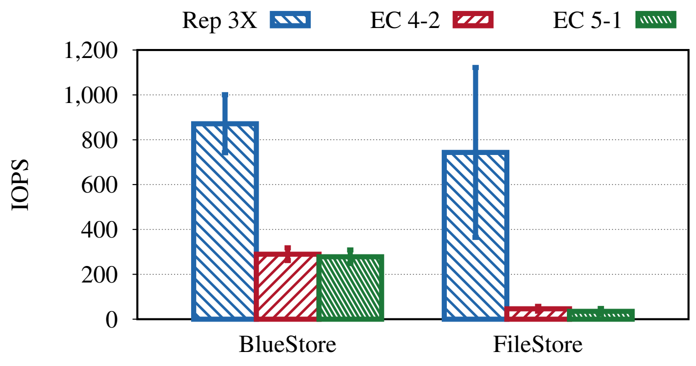


Figure 4.11: IOPS observed from a client performing random 4 KiB writes with queue depth of 256 to a Ceph virtual block device (RBD). The device is allocated on a 16-node all-HDD cluster.

Finally, for I/O sizes smaller than 64 KiB (not shown) the throughput of BlueStore is 20% higher than that of FileStore. For these I/O sizes BlueStore performs deferred writes by inserting data to RocksDB first, and then asynchronously overwriting the object data to avoid fragmentation (§ 4.4.2).

The throughput of read operations in BlueStore is similar or slightly better than that of FileStore for I/O sizes larger than 32 KiB. For smaller I/O sizes, as the rightmost graph in Figure 4.10 shows, FileStore throughput is better because of the kernel readahead. While RBD does implement a readahead, it is not as well-tuned as the kernel readahead.

4.6.3 Overwriting Erasure Coded (EC) Data

One of the features enabled by BlueStore is efficient overwrites of EC data. We measure the throughput of random overwrites for both BlueStore and FileStore. Our benchmark creates 1 TB RBD using one client. The client mounts the block device and performs 5 GiB of random 4 KiB writes with queue depth of 256. Since the RBD is striped in 4 MiB RADOS objects, every write results in an object overwrite. We repeat the experiment on a virtual block device allocated on a replicated pool and on an EC pool with parameters $k = 4$ and $m = 2$ (*EC4-2*), and $k = 5$ and $m = 1$ (*EC5-1*).

Figure 4.11 compares the throughput of replicated and EC pools when using BlueStore and FileStore backends. BlueStore EC pools achieve 6× more IOPS on EC4-2 and 8× more IOPS on EC5-1 than FileStore. This is due to BlueStore avoiding full physical copies during the first phase of the two-phase commit required for overwriting EC objects (§ 4.5.2). As a result, it is practical to use EC pools with applications that require data overwrite, such as RBD and CephFS, with the BlueStore backend.

4.7 Challenges of Building Storage Backends on Raw Storage

This section describes some of the challenges that the Ceph team faced when building a storage backend on raw storage devices from scratch.

4.7.1 Cache Sizing and Writeback

The operating system fully utilizes the machine memory by dynamically growing or shrinking the size of the page cache based on the applications' memory usage. It writes back the dirty pages to disk in the background trying not to adversely affect foreground I/O, so that memory can be quickly reused when applications ask for it.

A storage backend based on a file system automatically inherits the benefits of the operating system page cache. A storage backend that bypasses the file system, however, has to implement a similar mechanism from scratch (§ 4.4.2). In BlueStore, for example, the cache size is a fixed configuration parameter that requires manual tuning. Building an efficient user space cache with the dynamic resizing functionality of the operating system page cache is an open problem shared by other projects, like PostgreSQL [46] and RocksDB [90]. With the arrival of fast NVMe SSDs, such a cache needs to be efficient enough that it does not incur overhead for write-intensive workloads—a deficiency that current page cache suffers from [37].

4.7.2 Key-value Store Efficiency

The experience of the Ceph project demonstrates that moving all metadata to an ordered key-value store, like RocksDB, significantly improves the efficiency of metadata operations. However, the Ceph team has also found embedding RocksDB to be problematic in multiple ways: (1) RocksDB's compaction and high write amplification have been the primary performance limiter when using NVMe SSDs in OSDs; (2) since RockDB is treated as a black box, data is serialized and copied in and out of it, consuming CPU time; (3) RocksDB has its own threading model, which limits the ability to do custom sharding. These and similar problems with RocksDB and other key-value stores keeps the Ceph team researching for better solutions.

4.7.3 CPU and Memory Efficiency

Modern compilers align and pad basic datatypes in memory so that CPU can fetch data efficiently, thereby increasing performance. For applications with complex `structs`, the default layout can waste a significant amount of memory [39, 134]. Many applications are rightly not concerned with this problem because they allocate short-lived data structures. A storage backend that bypasses the operating system page cache, on the other hand, runs continously and controls almost all of a machine's memory. Therefore, the Ceph team spent a lot of time packing structures stored in RocksDB to reduce the total metadata size and the compaction overhead. The main tricks used were delta and variable-integer encoding.

Another observation with BlueStore is that on high-end NVMe SSDs the workloads are becoming increasingly CPU-bound. For its next-generation backend, the Ceph community is exploring techniques that reduce CPU usage, such as minimizing data serialization-deserialization and using the SeaStar framework [168], which avoids context switches due to locking by adopting a shared-nothing model.

4.8 Related Work

The primary motivator for BlueStore is the lack of transactions and unscalable metadata operations in file systems. In this section we compare BlueStore to previous research that aims to address these problems.

Transaction Support. Previous work has generally followed three approaches when introducing transactional interface to file system users.

The first approach is to leverage the in-kernel transaction mechanism present in the file systems. Examples of the this are Btrfs's export of transaction system calls to userspace [43], Transactional NTFS [103], Valor [180], and TxFS [85]. The drawbacks of this approach are the complexity and incompleteness of the interface, and the a significant implementation complexity. For example, Btrfs and NTFS both recently deprecated their transaction interface [26, 104] citing difficulty guaranteeing correct or safe usage, which corroborates FileStore's experience (§ 4.3.1). Valor [180], while not tied to a specific file system, also has a nuanced interface that requires correct use of a combo of seven system calls, and a complex in-kernel implementation. TxFS is a recent work that introduces a simple interface built on ext4's journaling layer, however, its implementation requires non-trivial amount of change to the Linux kernel. BlueStore, informed by FileStore's experience, avoids using file systems' in-kernel transaction infrastructure.

The second approach builds a user space file system atop a database, utilizing existing transactional semantics. For example, Amino [217] relies on Berkeley DB [138] as the backing store, and Inversion [137] stores files in a POSTGRES database [184]. While these file systems provide seamless transactional operations, they generally suffer from high performance overhead because they accrue the overhead of the layers below. BlueStore similarly leverages a transactional database, RocksDB, but incurs zero overhead because it eliminates the file system and runs the database on a raw disk. (RocksDB in BlueStore runs on BlueFS, which is a lightweight user space file system and not a full POSIX file system, like ext4 or XFS.)

The third approach provides transactions as a first-class abstraction in the operating system and implements all services, including the file system, using transactions. QuickSilver [162] is an example of such system that uses built-in transactions for implementing a storage backend for a distributed file system. Similarly, TxOS [150] adds transactions to the Linux kernel and converts ext3 into a transactional file system. This approach, however, is too heavyweight for achieving file system transactions, and such a kernel is tricky to maintain [85].

Metadata Optimizations. A large body of work has produced a plethora of approaches to metadata optimizations in local file systems. BetrFS [96] introduces B$^\varepsilon$-Tree as an indexing structure for efficient large scans. DualFS [145], hFS [224], and ext4-lazy [3] abandon traditional FFS [128] cylinder group design and aggregate all metadata in one place to achieve significantly faster metadata operations. TableFS [156] and DeltaFS [225] store metadata in LevelDB running atop a file system and achieve orders of magnitude faster metadata operations than file systems.

While BlueStore also stores metadata in RocksDB—a LevelDB derivative—to achieve similar speedup, it differs from the above in two important ways. In BlueStore, RocksDB runs on a raw disk incurring zero overhead, and BlueStore keeps all metadata, including the internal metadata, in RocksDB as key-value pairs. Storing internal metadata as variably-sized key-value pairs, as opposed to fixed-sized records on disk, scales more easily. For example, the Lustre distributed file system, which uses an ext4-derivate called LDISKFS for the storage backend, has changed on-disk format twice in a short period to accommodate for increasing disk sizes [20, 21].

4.9 Summary

Distributed storage system developers conventionally build their storage backends atop general-purpose file systems. This convention is attractive at first because general-purpose file systems provide most of the needed functionality, but in the long run it incurs a heavy file system tax—an overhead in code complexity, performance, and flexibility—on distributed storage systems. While the developers are acutely aware of the file system tax, they continue to pay it because they believe that developing a special-purpose storage backend from scratch is an arduous process, akin to developing a new file system, which takes a decade to mature.

Relying on the Ceph team's experience, we show this belief to be inaccurate, and we demonstrate that BlueStore, a special-purpose storage backend developed from scratch, liberates Ceph from the file system tax: First, it reclaims the significant performance left on the table when building a storage backend on top of a file system. Second, it efficiently implements new features, not practically implementable otherwise, by gaining complete control of the I/O stack. Third, it frees a Ceph from being locked into the hardware that the file system supports. In the next chapter, we leverage this freedom to liberate Ceph from the block interface tax as well.

Chapter 5

Freeing Ceph From the Block Interface Tax

The Ceph distributed storage system achieved freedom from the file system tax by implementing a clean-slate, special-purpose storage backend, BlueStore. In this chapter, we leverage this freedom and flexibility to extend BlueStore to work on zoned devices and liberate Ceph from the block interface tax as well. Since BlueStore stores metadata in RocksDB key-value store, we first extend RocksDB to run on zoned devices. We then introduce new components to BlueStore that enable data management on zoned devices. Finally, we demonstrate how freedom from the block interface tax and the file system tax allows Ceph to achieve cost-effective data storage and predictable performance.

5.1 The Emergence of Zoned Storage

The hard drive industry paved the way to zoned storage by introducing Shingled Magnetic Recording (SMR). SMR increases hard drive capacity by over 20% through partially overlapping magnetic tracks on top of each other, constraining writes to large sequential chunks and preventing small random updates [75]. Hard drive manufacturers introduced Drive-Managed SMR (DM-SMR) drives, which expose the block interface through a translation layer (Chapter 2). Realizing the full potential of SMR without paying the block interface tax, however, is only possible with Host-Managed SMR (HM-SMR) drives. HM-SMR drives expose the new and backward-incompatible *zone interface*—technically known as ZBC/ZAC in the hard disk drive context, after the corresponding new command sets added to SCSI and ATA standards [94, 95].

Despite the backward-incompatible interface, HM-SMR drives have seen widespread adoption. They are natively supported in the Linux kernel with a mature storage stack, and they were adopted by cloud storage providers [122, 123, 153] and storage server vendors [121][31]. Over half of data center hard disk drives are expected to use SMR by 2023 [175].

While HDDs embraced sequential-write-only zones only recently through SMR, SSDs always had them in the form of NAND flash erase blocks. SSD designers, however, avoided breaking backward-compatibility in the past by exploiting the fast I/O performance of NAND flash, copious media overprovisioning [206], and sophisticated Flash Translation Layer (FTL) algorithms. Nevertheless, the SSDs have not been able to completely avoid the block interface tax—the garbage collection performed by the FTL has long been identified as a source of unpredictable performance and high tail latency in SSDs [80, 105, 221], and the extra hardware needed for the efficient operation of the FTL has significantly increased the device cost.

The Open-Channel SSD (OCSSD) initiative pioneered the elimination of the FTL by exposing erase blocks to the host for achieving improved and predictable performance, as well as significant cost reduction [18]. Although OCSSD had major early backers [38, 70], the lack of standardization led to vendor-specific implementations, impeding widespread adoption.

Zoned Namespaces (ZNS) [19] is a new NVMe standard that draws inspiration from the experience of the OCSSD architecture as well as from the success of ZBC/ZAC interface for HM-SMR drives. OCSSD takes, in some implementations, an extreme approach in exposing the raw NAND flash to host, requiring the host to manage low-level details, such as error correction and wear leveling. ZNS, on the other hand, exposes sequentially written erase blocks using a clean interface, hiding complex details of raw flash management from host. ZNS leverages the similarity of SMR zones to erase blocks and extends the ZBC/ZAC zoned storage model to manage NAND flash media, resulting in a new single interface—the zone interface—for managing both ZNS SSDs and HM-SMR HDDs. (Although ZNS is an NVMe extension and ZBC/ZAC are SCSI/ATA extensions, a library is in the works that hides this difference and enables the same application written for the zone interface to run on both ZNS SSDs and HM-SMR HDDs [208].) The zone interface aligns a zone with sequentially written erase blocks in ZNS SSDs and a physical zone in HM-SMR HDDs, obviating the need for in-device garbage collection. As such, it moves the responsibility for data management and garbage collection to the host.

In this chapter, we demonstrate how the Ceph distributed storage system can avoid paying the block interface tax through the adoption of the zone interface. Ceph is already avoiding the file system tax by implementing a special-purpose, clean-slate storage backend, BlueStore. We extend BlueStore to work on zoned devices and demonstrate that Ceph can now (1) store data more cost-effectively by leveraging the extra 20% of capacity in HM-SMR drives, without paying for the performance penalty of the translation layer in DM-SMR drives and (2) reduce the I/O tail latency by leveraging the control over the garbage collection and the redundancy of data in a distributed setting. Also, since BlueStore relies on RocksDB—a widely used key-value store—for storing metadata, as a part of this work we extend RocksDB to work on zoned devices, and we demonstrate how RocksDB too avoids the block interface tax and eliminates in-device write amplification through intelligent data management.

5.2 Background

BlueStore stores metadata in RocksDB and data on a raw block device (Figure 4.5). Hence, we extend BlueStore to work on zoned devices in two steps: first, we handle the metadata path and extend RocksDB to run on zoned devices, and second, we handle the data path and extend BlueStore to store data on raw zoned devices. Before going into details how we accomplish these tasks, we introduce the zone interface and give an overview of RocksDB's architecture; the overview of BlueStore's architecture can be found in § 4.4.

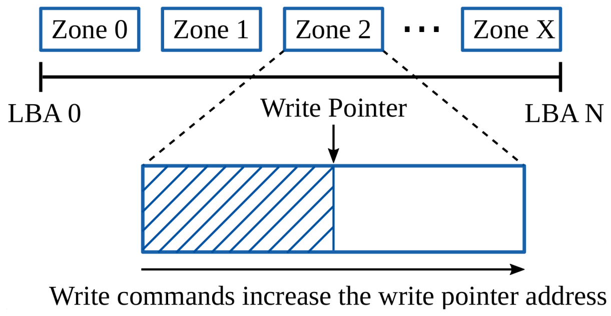


Figure 5.1: Zoned storage model.

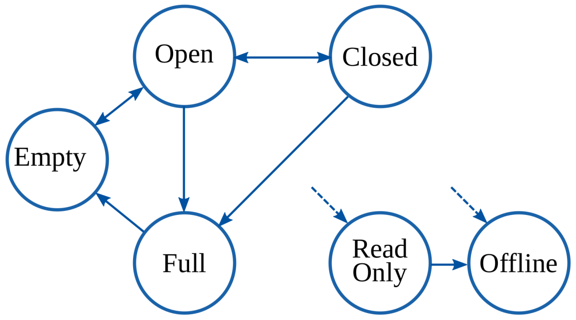


Figure 5.2: Zone state transition diagram.

5.2.1 Zoned Storage Overview

The zoned storage model partitions the device's logical block address (LBA) space into fixed-sized partitions called *zones*, as Figure 5.1 shows, and it introduces the following constraints: writes to a zone must be sequential in the LBA order, and a zone must be reset through a *zone reset* command before LBAs in it can be (sequentially) written again. Each zone also maintains a write pointer: writes to a zone increment the write pointer and it points to the next writable LBA within the zone.

A zone can be in one of the following states, as the simplified state diagram from the ZNS technical proposal [19] in Figure 5.2 shows:

- *Empty*: No writes have been issued since the last zone reset operation.
- *Open*: A zone is partially written.
- *Full*: All the LBAs in the zone have been written.
- *Closed*: A zone has been transitioned from Open to Closed state due to resource constraints.
- *Read Only* or *Offline*: A vendor-specific event has transitioned the zone to either Read Only or Offline state.

When the zone is in the *empty* state, the write pointer points to the first LBA of a given zone. When the zone is in the *open* state, the write pointer points to an LBA within the zone, and when the zone is *full*, the write pointer is invalid. If a write command attempts to write anywhere other than the LBA pointed to by the write pointer, an I/O error occurs.

In addition to the zone state transitions caused by writes and device-side events, zone commands are available to manipulate the zone states: The *open zone* command transitions a zone in the empty or closed state to the open state. The *close zone* command transitions a zone from the open state to the closed state. The *reset zone* command transitions a zone from the open or full state to the empty state. If the transition is not valid, the commands return an I/O error.

We leave the full treatment of the zone interface to respective specifications [19, 94, 95], and conclude our overview of the zone interface by highlighting two of the differences between HM-SMR HDDs and ZNS SSDs that are relevant to our discussion in the rest of this chapter. The first difference is about the *zone size* and *zone capacity*. The *zone size* is fixed and equal for all zones in both HM-SMR HDDs and ZNS SSDs and it is typically a power of two—a property used by block layer [17] for fast serialization, boundary checks, and lookup of zone attributes. The *zone capacity*, on the other hand, identifies the usable capacity of a zone: it is equal to the zone size in HM-SMR

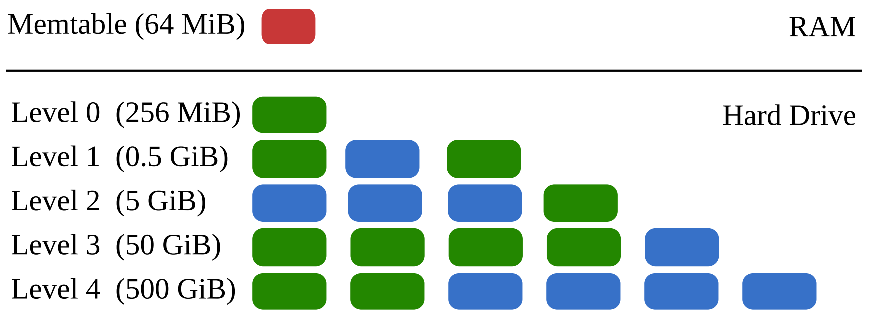


Figure 5.3: Data organization in RocksDB. Green squares represent Sorted String Tables (SSTs). Blue squares represent SSTs selected for two different concurrent *compactions*.

HDDs, but it is variable and usually smaller than the zone size in ZNS SSDs due to inherently variable erase block sizes.

The second difference is about how long a zone can stay in the *open* state. Contrary to the zones in HM-SMR HDDs, the zones in ZNS SSDs cannot stay indefinitely in a partially written—that is in the *open*—state. A partially written zone contains partially written erase blocks, which are prone to errors from read disturbances or physical properties of the media unless they are fully written. Thus, to maintain data reliability, the host can pad the partially written zone or the SSD itself can detect and pad such zones and transition them to the *full* state.

5.2.2 RocksDB Overview

RocksDB [62] is an instance of a Log-Structured Merge-Tree (LSM-Tree), an indexing data structure that increases the effective use of bandwidth by reducing seeks in hard disk drives, over the alternative, the B-Tree [41]. Every key-value inserted to RocksDB is first individually written to a Write-Ahead Log (WAL) file using the `pwrite` system call, and then buffered in an in-memory data structure called *memtable*. By default, RocksDB performs *asynchronous inserts*: `pwrite` returns as soon as the data is buffered in the OS page cache and the actual transfer of data from the page cache to storage is done later by the kernel writeback threads. Hence, a machine crash may result in loss of data for an insert that was acknowledged. For applications that require the durability and consistency of transactional writes RocksDB also supports *synchronous inserts* that do not return until data is persisted on storage.

RocksDB stores data in files called Sorted String Tables (SSTs), which are organized in levels as shown in Figure 5.3. When the memtable reaches a preconfigured size, its content is written out to an SST in level L_0, and a new memtable is created. The aggregate size of each level L_i is a multiple of L_{i-1}, starting with a fixed size at L_1. When the number of L_0 SSTs reaches a threshold, the *compaction* process selects all of L_0 SSTs, reads them into memory, sorts and merges them, and writes them out as new L_1 SSTs. For higher levels, compactions are triggered when the aggregate size of the level exceeds a threshold, in which case one SST from the lower level and multiple SSTs from the higher level are compacted. For example, Figure 5.3 shows two concurrent compactions, with the one of them happening between L_1 and L_2 and another one happening between L_3 and L_4. If memtable flushes or compactions cannot keep up with the rate of inserts, RocksDB stalls inserts to avoid filling storage and to prevent lookups from slowing down.

5.3 Challenges of RocksDB on Zoned Storage

RocksDB in BlueStore runs on top of BlueFS (§ 4.4.1), a minimal user space file system that runs on a block device. Every I/O system call issued by RocksDB is eventually issued by BlueFS to the raw storage device. Hence, to get RocksDB to run on a zoned device we need to adapt BlueFS to run on a zoned device, which comes with several challenges. Below we describe these challenges and our solutions to them.

5.3.1 Zone Cleaning

Placing SSTs produced by RocksDB one after another into the zones of a zoned device leads to the segment cleaning problem of the Log-Structured File System (LFS) [158]. Since SST sizes are much smaller than the zone size, after multiple compactions zones become fragmented: in addition to live SSTs, they end up containing dead SSTs that have been merged to a new SST that was written to another zone. Reclaiming the space occupied by the dead SSTs requires migrating live SSTs to another zone, which increases the write amplification.

Recent work proposes a new data format and compaction algorithm to eliminate zone cleaning and achieve the ideal write amplification of 1 [222]. Cleaning can be eliminated, however, without making any changes to the compaction algorithm or data format, by simply matching SST and zone sizes and aligning the start of an SST with the start of a zone. Hence, by mapping a complete SST to a zone, a dead SST space can be reclaimed by merely resetting the zone's write pointer, thereby eliminating zone cleaning and achieving the write amplification of 1. There are other compelling reasons for increasing the SST size, such as enabling disks to do streaming reads with fewer seeks, reducing expensive `sync` calls, and reducing the number of open file handles. This approach is similar to that of SMRDB [148], however, unlike SMRDB we do not restrict the number of LSM-Tree levels and we do not introduce a new data format that breaks backward compatibility.

5.3.2 Small SSTs

While we can specify the size of an SST as a configuration option to RocksDB, it is not guaranteed that all generated SSTs will be of the specified size. For example, if compaction takes nine SSTs from L_3 and one SST from L_2 and merges them, unless there is no overlap among the merged data, the process may produce ten SSTs with the last one being smaller than the rest. Although having an SST smaller than the zone is not a problem for HM-SMR drives, it is a problem for ZNS SSDs because partially written zones may lose data after some time(§ 5.2.1).

To address this issue, as a first step, when running on ZNS SSDs we null-pad the zones that are written a small SST. Since padding consumes device bandwidth and is a side-effect of the device's design, we count padding bytes as write amplification. Our experiments that insert semi-random data to RocksDB shows that padding results in a write amplification of 1.2. We leave it as future work to fill the zones with useful data in case of small SSTs and reach the ideal write amplification of 1.

5.3.3 Reordered Writes

Like most LSM-Tree implementations, RocksDB uses buffered I/O when writing compacted SSTs to disk. This keeps SSTs in the operating system page cache and improves performance significantly for two reasons: (1) during compaction cached SSTs are read from memory, and (2) key lookups in cached SSTs are served from memory.

Using buffered I/O, however, does not guarantee write ordering that is essential for zoned devices. Page writeback can happen from different contexts at the same time, and the pages picked up by each context may not be zone-aligned. Furthermore, there are no write-alignment constraints with buffered writes, so an application may write parts of a page across different operations. In this case, however, the same last page cannot be overwritten to add the remaining data when the sequential write stream resumes. Thus, to guarantee write ordering, (1) we must use *direct I/O*, which is enabled by `O_DIRECT` flag to the `open` system call, (2) issue writes to a zone sequentially, and (3) use an I/O scheduler that preserves ordering of writes issued from user space, such as `deadline` or `mq-deadline` I/O schedulers [42].

While direct I/O does not cause performance problems for ZNS SSDs due to low latency and high internal parallelism, for HM-SMR drives it consumes a big chunk of storage bandwidth for reading SSTs during compaction. To avoid this, we implement whole file caching in BlueFS, thereby serving SST files from memory during compaction.

5.3.4 Synchronous writes to the WAL

The `libzbc` [207] library is the de-facto method for interacting with zoned devices [124, 222]. It provides the `zbc_pwrite` call for positional writes to the device, with similar semantics to the `pwrite` system call. Even though `pwrite` is a synchronous call, when used with buffered I/O, as in RocksDB (§ 5.2.2), it is effectively made asynchronous.

With zoned devices, however, we have to use direct I/O, which means the `zbc_pwrite` call must wait for the device to acknowledge the write. This has negligible overhead when data is written in large chunks, as is the case for memtable flushes and SST writes during compaction. Writes to the WAL, on the other hand, happen after each key-value insertion. As a result, every insertion must be acknowledged by the device, limiting the throughput to that of small synchronous writes to the device.

To avoid this bottleneck, we switch to using `libaio` library—the in-kernel asynchronous I/O framework—in BlueFS. This approach works as long as the asynchronous I/O operations are issued in order and the kernel I/O scheduler does not reorder I/Os [132]. We still use `libzbc` for resetting zones and flushing the drive.

5.3.5 Misaligned writes to the WAL

Random and misaligned writes violate the zone interface and therefore cannot be used with zoned devices. In RocksDB, there are three sources of such writes. First, when the key and value sizes are smaller than 4 KiB, the last block of SSTs may get overwritten when writing SSTs to disk. We have found this to be a bug in RocksDB, which we have reported and the RocksDB team has fixed [54]. Second, if an append operation to the WAL leaves empty space in the last written block, the following append operation overwrites that block to fill the empty space. Third, RocksDB produces a handful of small files—such as the manifest file—that receive negligible amounts of I/O, which also requires page overwrites. In the following section we describe how we address the second and third source of random writes.

5.4 Handling Metadata Path—RocksDB on Zoned Storage

All the I/O system calls issued by RocksDB are emulated by BlueFS, which is a user space file system that opens a raw block device and issues actual I/O system calls. Among other functionalities, BlueFS also implements a block allocator and journaling for metadata consistency. Below we describe several design changes we make to BlueFS to make it work on zoned devices.

5.4.1 File types and space allocation

There are three file types that are essential for the correct operation of RocksDB: SSTs, WAL files, and manifest files that act as a transactional log of RocksDB state changes.

SST and WAL files are large files that receive the bulk of the I/O. SSTs are the simplest to handle—they are append-only files and we map them to individual zones upon creation and align their size to the zone size. We handle misaligned writes to the WAL (§ 5.3.5) by wrapping every WAL write in a record with the write length stored inline and padded to the 4 KiB boundary. With this change, the WAL reads can no longer directly know the data offset in an extent without first reading the record lengths. This, however, is not a problem because reading the WAL is not on the hot path—it is only read sequentially and only during crash recovery. In addition, we allocate a fixed number of zones for WAL files and use them as a circular buffer.

We handle last block overwrites in manifest files by dedicating two zones to them, only one of which is active at a time. When a manifest file reaches a certain size, a new one is created and appended to the end of the zone, invalidating the older version. When a zone becomes full, the latest manifest file is written to the other zone, the write pointer of the full zone is reset, and these two zones are used as a circular buffer. During boot, the latest manifest file is identified by scanning both of the zones.

5.4.2 Journaling and Superblock

BlueFS maintains an inode for each file with a serial number, the list of extents allocated to the file (complete zones in case of SST and WAL files), the actual size of the file, and so on. BlueFS uses an internal journal for consistency, which contains the only copy of all metadata. At mount, the journal is replayed and the inodes are loaded in memory. For every metadata operation, such as file creation or zone allocation, the journal and in-memory metadata are updated. When the journal reaches a preconfigured size, it is compacted and written to a new zone, and the superblock is updated to point to it.

Since a superblock cannot be updated in-place, we use the same two-zone circular buffer method that we used for manifest files and append the latest version of the superblock to the end of the zone. At mount, the latest version of the superblock is found by scanning the first two zones of the drive.

5.4.3 Caching

To reduce the number of SST files read from disk during compaction (§ 5.3.3), we implement a FIFO whole file cache in BlueFS. Every SST file that is written is also placed to a cache of preconfigured size. A disadvantage of our cache implementation is that its unit of eviction is a file. While the operating system page cache will evict only a part of a file when it runs out of space and continue to serve the remaining blocks from memory, by evicting a whole file, our implementation, for example, leaves 12.5% of the 2 GiB cache empty when using 256 MiB SST files. We leave a more efficient cache implementation as future work.

5.5 Evaluation of RocksDB on HM-SMR HDDs

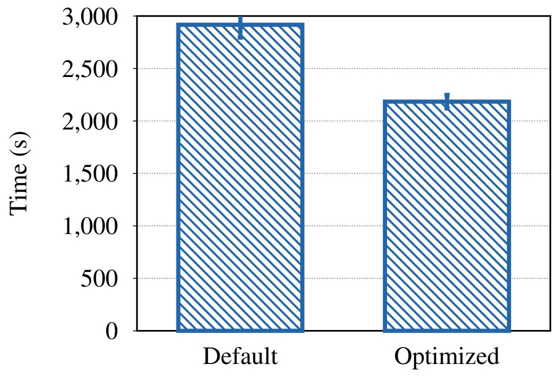
We evaluate RocksDB with BlueFS running on zoned devices in two parts. In this section, we evaluate it on an HM-SMR drive, where we incrementally show how each of our optimizations improves performance. In the next section, we take our optimized code and evaluate it on a prototype ZNS SSD.

5.5.1 Evaluation Setup

We run all experiments on a system with AMD Opteron 6272 2.1 GHz CPU with 16 cores and 128 GiB of RAM, running Linux kernel 4.18 on the Ubuntu 18.04 distribution. For evaluation we use a 12 TB CMR drive (HGST HUH721212AL), a 14 TB HM-SMR drive (HGST HSH721414AL), and an 8 TB DM-SMR drive (Seagate ST8000AS0022). These drives are similar mechanically and have 200+ MiB/s sequential I/O throughput at the outer diameter.

Unless otherwise noted, all our experiments run a benchmark that inserts 150 million pairs of 20-byte keys and 400-byte values using the db_bench tool that comes with RocksDB. During the benchmark run, RocksDB writes 200 GiB of data through memtable flushes, compaction, and WAL inserts, and reads 100 GiB of data due to compaction. The size of the database is 59 GiB uncompressed and 31 GiB compressed. To emulate a realistic environment where the amount of data in the operating system page cache is a small fraction of the data stored on a high-capacity drive, we limit the operating system memory to 6 GiB. This leaves slightly more than 2 GiB of RAM for the page cache after the memory used by the operating system and benchmark application, resulting in 1:15 ratio of cached to on-disk data. We repeat every experiment at least three times and report the average and the standard deviation.

We establish two baselines. The first baseline is RocksDB running on the XFS file system (recommended by the RocksDB team [63]) running on a CMR drive. This is the baseline whose performance we want to match with RocksDB running on BlueFS on an HM-SMR drive, given that we are running RocksDB on a higher capacity device with a more restrictive interface. The second baseline is RocksDB running on the XFS file system running on a DM-SMR drive. This



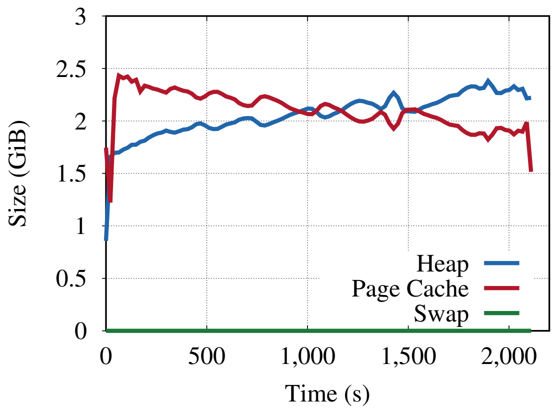


Figure 5.4: Benchmark runtimes of RocksDB with default and optimized settings on a CMR drive.

Figure 5.5: Memory used by the OS page cache and heap allocations, and the swap usage during the optimized RocksDB run.

is a baseline we want to beat given that it is the only viable option for running RocksDB on a high-capacity SMR drive, but has suboptimal performance due to block interface tax.

5.5.2 Establishing CMR Baseline

We run RocksDB on XFS and tune RocksDB settings for optimal performance on a CMR drive [64]. Figure 5.4 shows that with optimized settings, the benchmark completes 34% faster. Figure 5.5 shows the heap memory allocated by RocksDB and `db_bench`, the memory used up by the operating system page cache, and the swap usage. The two key observations are that the page cache almost always uses more than 2 GiB of RAM, and no swapping occurs. Next, we look at the settings we change to get this effect.

We focus on two tunables that impact performance most: `compaction_readahead_size` and `write_buffer_size`. We increase the former from the default of 0 MiB to 2 MiB. This setting determines the size with which `pread` system call issues read requests for reading SSTs during compaction. With the default setting, RocksDB issues 4 KiB requests and prefetches 256 KiB at a time, however, given that the compaction threads read large SSTs sequentially into memory, 256 KiB request size is suboptimal and incurs unnecessary seeks due to interruptions from other ongoing disk operations. Figure 5.6 (a) shows that by increasing the request size to 2 MiB, the number of seeks drops from the average of 110 seeks per second to almost 20 seeks per second.

We also increase the `write_buffer_size` from the default of 64 MiB to 256 MiB. This setting determines the memtable size that will be buffered in memory before being flushed to disk. As expected, increasing the memtable size by 4×, reduces the number of expensive `fsync`/`fdatasync` system calls issued by a similar amount, as Figure 5.6 (b) shows.

These two configuration changes are the reason behind the 34% speedup shown in Figure 5.4. We also experimented with the maximum number of concurrent background jobs and settled on 4 threads: lower or higher values result in worse performance. Finally, we increased SST sizes from 64 MiB to 256 MiB, but this change did not have a noticeable effect on performance; it did, however, allow us to exactly map an SST to an HM-SMR zone. We choose RocksDB with optimized

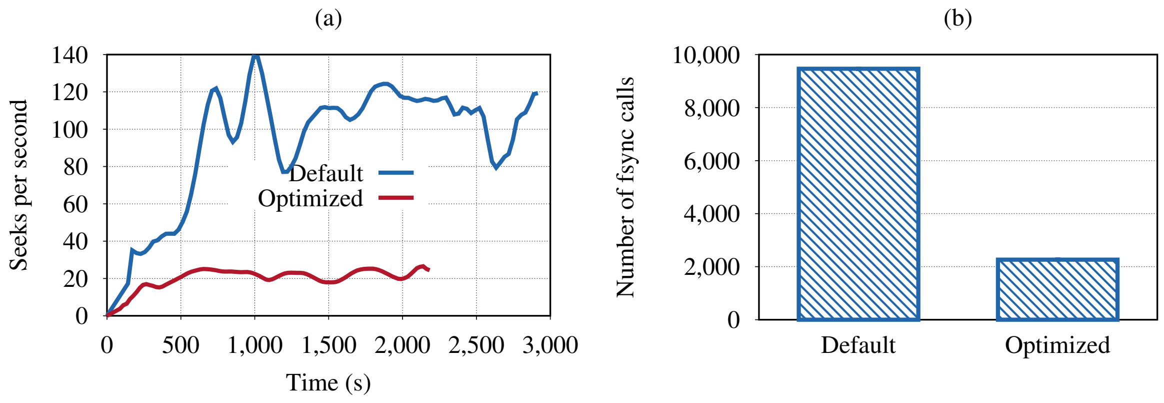


Figure 5.6: (a) The number of seeks per second due to reads and (b) the number of `fsync` and `fdatasync` system calls during the benchmark run with default and optimized RocksDB settings on a CMR drive.

settings running on XFS on a CMR drive as our CMR baseline.

5.5.3 Establishing DM-SMR Baseline

The optimized settings that we used for the CMR drive are beneficial to a DM-SMR drive for the same reasons. The settings, however, have even bigger impact on the DM-SMR drive, where the benchmark completes 63% faster than when running with the default settings. This may sound surprising, given two facts: (1) LSM-Trees are known to produce large sequential writes [119, 195][136], and (2) we have already established that DM-SMR drives handle sequential writes with no overhead (§ 3.4.3). Then why does RocksDB, which produces sequential writes, suffer a performance overhead on a DM-SMR drive? It turns out that current DM-SMR drives can detect sequential writes only if there is a single stream. RocksDB, on the other hand, produces concurrent sequential streams due to memtable flushes and concurrent compactions (§ 5.2.2), and the block layer in the operating system splits large I/O requests into small chunks—for hard drives the size of this chunk is 512 KiB by default in Linux. Chunks from different streams get mixed and sent to the drive in an arbitrary order, which appears as non-sequential writes to the drive. Metadata writes of the underlying file system further complicates detecting sequential streams. Hence, writes to otherwise append-only files end up in the persistent cache and cause garbage collection, incurring performance penalty to RocksDB running on DM-SMR drives.

We choose RocksDB with optimized settings running on XFS on a DM-SMR drive as our DM-SMR baseline. The left two bars in Figure 5.7 shows both of our baselines, where the CMR baseline is about 86% faster than the DM-SMR baseline. Our aim is to get RocksDB running on BlueFS on an HM-SMR drive to beat RocksDB on a DM-SMR drive and to perform at least as good as RocksDB on a CMR drive. For brevity, from now on we omit the file system when discussing the baselines and our work: RocksDB on CMR or DM-SMR drives implies that RocksDB is running on XFS, and RocksDB on HM-SMR drive implies that RocksDB is running on BlueFS.

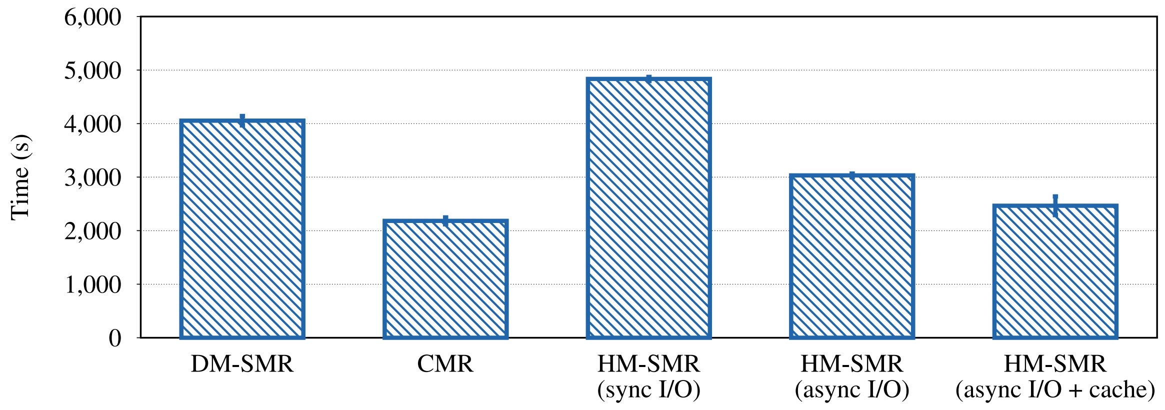


Figure 5.7: Benchmark runtimes of the baselines—RocksDB on DM-SMR and CMR drives (on the left)—and RocksDB on BlueFS iterations on an HM-SMR drive. The benchmark asynchronously inserts 150 million key-value pairs with 20-byte keys and 400-byte values.

5.5.4 Getting RocksDB to Run on an HM-SMR Drive

In our first iteration of changes to BlueFS we use `libzbc` and perform synchronous I/O on the HM-SMR drive, which causes the writes to the WAL to become a bottleneck (§ 5.3.4). Figure 5.7 shows that on this iteration—denoted by HM-SMR (sync I/O)—the benchmark completes in 4,800 seconds—slower than both DM-SMR and CMR baselines.

Figure 5.8 (a) gives a detailed look at the first 50 seconds of the run. We see that in the first 7 seconds, during which only writes to the WAL happen, the write throughput is fixed at 40 MiB/s. In the next 3 seconds a memtable is flushed, which increases the write throughput to 80 MiB/s and reduces the insert throughput to 60 Kops/s. This is expected because insert throughput is determined by the speed of writes to WAL, and during memtable flush, we have two threads sharing the bandwidth: one is flushing memtable data and the other is writing to the WAL. Once the memtable flush completes at the 10th second, the insert throughput jumps back and the write throughput is again at 40 MiB/s. Another memtable flush happens and the pattern repeats, and right at the end of the second memtable flush a compaction starts at the 17th second. This increases read and write throughput because the compaction is done by a single thread that reads old SSTs and writes new ones, and reduces insert throughput to the same level as during the memtable flush because we still have two threads sharing the bandwidth. Before the compaction completes, another memtable flush starts at the 23rd second. Now we have three threads sharing the disk bandwidth. The write bandwidth is the highest because all of them are writing, read bandwidth is slightly lower, and most importantly, the insert throughput is at the lowest.

During the whole run, RocksDB never stalls inserts (§ 5.2.2) because the slow writes to the WAL produce small enough work that memtable flushes and compactions can keep up with.

5.5.5 Running Fast with Asynchronous I/O

In our second iteration we switch to `libaio` and perform asynchronous I/O on the HM-SMR drive—we still rely on `libzbc` for zone operations. Figure 5.7 shows that on this iteration—

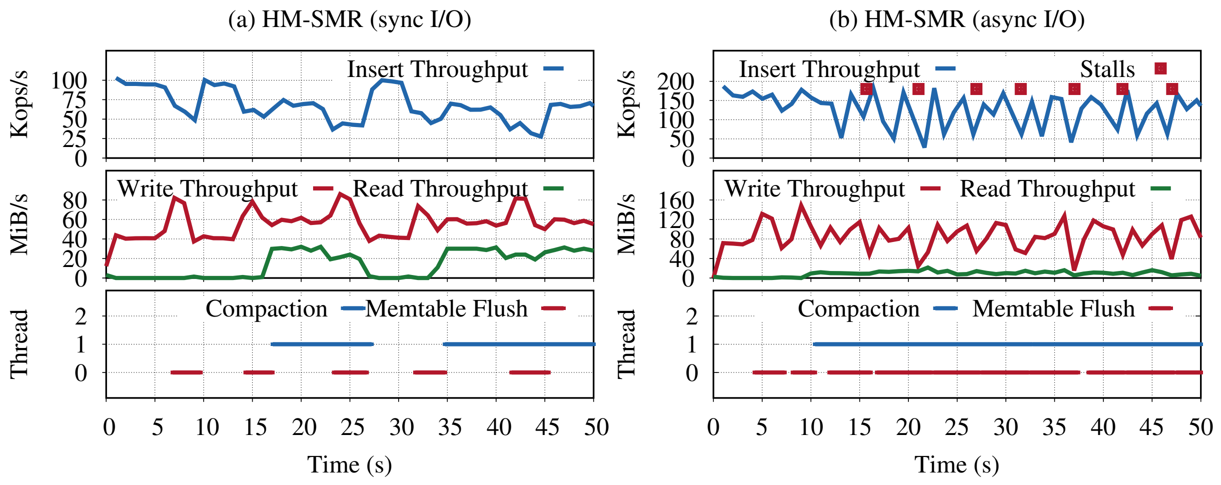


Figure 5.8: Insertion throughput, HM-SMR drive read/write throughput, and compaction and memtable flush operations during the first 50 seconds of the benchmark with RocksDB on HM-SMR drive using (a) synchronous and (b) asynchronous I/O.

denoted by HM-SMR (async I/O)—the benchmark completes in 3,000 seconds, 60% faster than when performing synchronous I/O. At this point, RocksDB on HM-SMR is already 30% faster than RocksDB on DM-SMR, but it is not as fast as RocksDB on CMR because during compaction it reads SSTs from disk using direct I/O and consumes the drive bandwidth that could be used for writes.

Figure 5.8 (b) gives a detailed look at the first 50 seconds of the run. We see that writes to the WAL during the first 3 seconds are almost twice as fast, and memtable flush starts 3 seconds earlier, compared to Figure 5.8 (a). Fast inserts result in continuous memtables flushes, and similarly, compaction starts 7 seconds earlier and does not stop. Unable to keep up with the work, RocksDB stalls inserts almost every 5 seconds, which results in a spiky throughput graph. Despite the stalls, the average throughput is higher and RocksDB completes the benchmark 60% faster than when using synchronous I/O.

Figure 5.9 shows the similar graph for the whole run. The most outstanding pattern in Figure 5.9 (a) are the long periods of very low throughput, such as the one between 450th and 660th seconds. Figure 5.9 (c) shows that during this period only compaction and no memtable flushing is happening, suggesting that RocksDB has stalled inserts until compaction backlog is cleared. During this period, effectively only the compaction thread is running, and as the period between 450th and 660th seconds shows in Figure 5.9 (b), it divides the disk bandwidth between reading SSTs from the drive and writing them out. To speed up the reads and thereby compactions, in our next iteration we add whole file caching of SSTs to BlueFS.

5.5.6 Running Faster with a Cache

In our third iteration we add a simple write-through FIFO cache to BlueFS for caching the SST files produced at the end of compaction. Figure 5.7 shows that on this iteration—denoted by HM-SMR (async I/O + cache)—the benchmark completes in 2,500 seconds—20% faster than the version

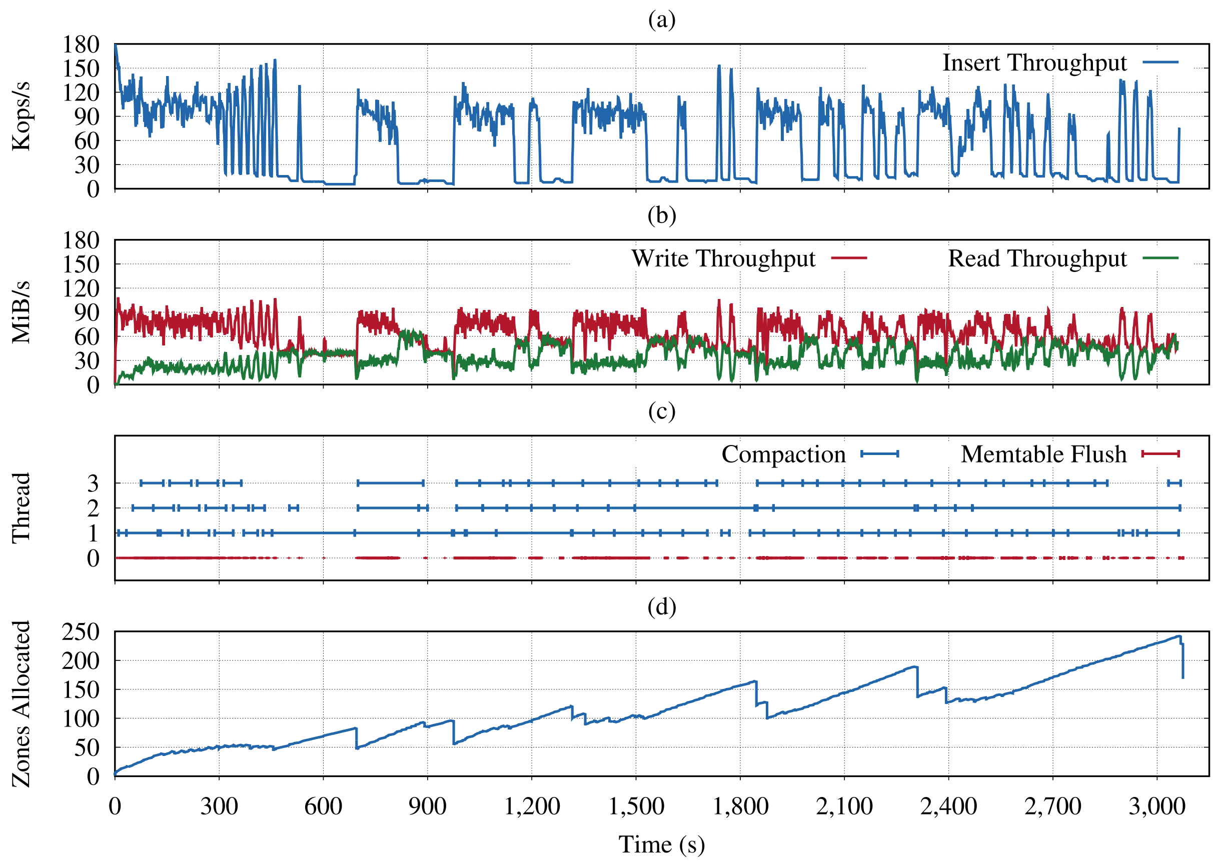


Figure 5.9: (a) Insertion throughput, (b) read and write throughput, (c) compaction and memtable flush operations, and (d) number of zones allocated during the whole benchmark run of RocksDB on HM-SMR drive using asynchronous I/O.

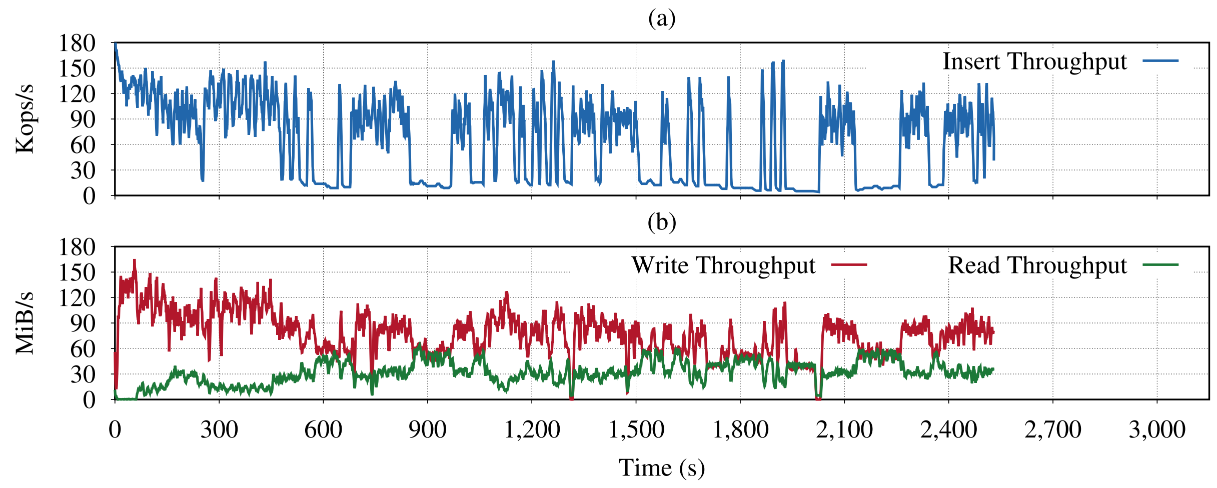


Figure 5.10: (a) Insertion throughput and (b) read and write throughput during the whole benchmark on run of RocksDB on HM-SMR drive using asynchronous I/O and caching.

with no cache. In this run, we set the SST cache size to 2 GiB—the same amount of memory that the operating system page cache uses when we run the benchmark on the CMR drive (§ 5.5.2). Figure 5.10 shows the detailed graph for the whole run. Comparing it to Figure 5.9 we see that, for example, up to 450th second, the read throughput is lower, and between 450th and 660th seconds the read throughput stays below 30 MiB/s compared to rising 40 MiB/s, suggesting that some of the reads are being served from the cache. Similarly, the write throughput stays above 60 MiB/s compared to staying at 40 MiB/s, and the insertion throughput is not flat at the bottom, indicating that writes to the WAL are still happening and the inserts are not stalled as badly as in the case with no file caching. Overall, Figure 5.10 has shorter periods of low insertion throughput compared to Figure 5.9 and therefore, higher average insertion throughput. Thus, with all our optimizations we almost match the CMR baseline thereby avoiding the block interface tax—the small difference is caused by the inefficiency of whole file caching, which can be improved with a better design (§ 5.4.3).

5.5.7 Space Efficiency

Figure 5.9 (d) shows that RocksDB on HM-SMR makes optimal use of disk space. Matching the figure with Figure 5.9 (c) shows that allocated zones grow during long compaction processes and they are released at the end. This is because RocksDB first merges multiple SSTs into one large temporary file and then deletes the temporary file after creating new SSTs as a result of the merge. We see that the number of zones drop to 168, which corresponds to about 42 GiB, which is surprisingly larger than the 31 GiB database size (§ 5.5.1). This is the result of the benchmarking application, `db_bench`, shutting down as soon as it finishes the benchmark, without waiting for all compactions to complete, leaving some large temporary SSTs around. We modified `db_bench` to wait until all compactions have completed, and observed that the number of bands dropped to 135, which is about 33 GiB, only slightly larger than the database size because some of the space is occupied by the WAL files, and some of the SSTs produced as a result of compaction may end up being smaller than 256 MiB.

5.6 Evaluation of RocksDB on ZNS SSDs

In this section we compare RocksDB on BlueFS running on ZNS SSD to RocksDB running on XFS on a conventional enterprise SSD with an FTL inside. We show two results. First, we demonstrate that despite popular belief, LSM-Trees in general and RocksDB in particular are not ideal workloads for conventional SSDs—they can result in device write amplification of 5 on enterprise SSDs. Second, we demonstrate that the zone interface is a perfect match for LSM-Trees, and we show how RocksDB on BlueFS uses zone interface to avoid the block interface tax and achieve a write amplification of 1.2.

5.6.1 Evaluation Setup

We perform all experiments on a system with a six-core Intel i7-5930K (Haswell-E) 3.5 GHz CPU, running Linux kernel 5.2 on the Ubuntu 18.04 distribution. We fix the memory size to 6 GiB using

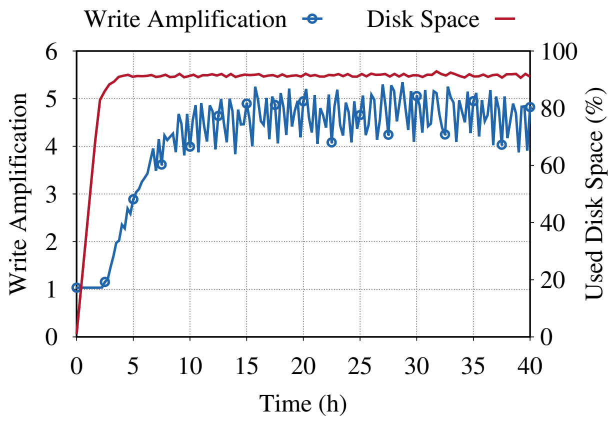


Figure 5.11: Device write amplification of the enterprise SSD during RocksDB benchmarks. First, the `fillseq` benchmark sequentially inserts 7 billion key-value pairs; it completes in about two hours, filling the drive up to 80%, and the write amplification is 1 during this period. Next, the `overwrite` benchmark overwrites 7 billion key-value pairs in about 40 hours, during which the write amplification rises to about 5 and stays there.

a kernel boot setting to emulate a real-world setting where data does not fit into memory. As a ZNS SSD we use a prototype device with a custom firmware that implements ZNS as a shim layer inside a 1 TB Western Digital PC SN720 NVMe SSD [50] on top of the existing FTL. As a conventional SSD we use an enterprise data center SSD with 3.84 TB capacity.

5.6.2 Quantifying the Block Interface Tax of RocksDB on Conventional SSDs

Like most LSM-Trees, RocksDB generates large sequential writes when compacting SSTs and flushing memtables. While this pattern is ideal for low write amplification, when these operations result in concurrent sequential streams, they lead to surprisingly high write amplification on conventional SSDs.

To demonstrate the high write amplification caused by RocksDB when running on a conventional SSD, we perform the following experiment. We install the XFS file system on the enterprise SSD and configure RocksDB to use 512 MiB SST files. Since enterprise SSDs overprovision 28% NAND flash [206], they do not start garbage collection until most of drive has been written. Therefore, we first fill 80% of the drive by inserting 7 billion key-value pairs of 20-byte keys and 400-byte values, using the `fillseq` benchmark of `db_bench`. We then start randomly overwriting these key-value pairs using the `overwrite` benchmark. We measure the write amplification of the enterprise SSD in 15-minute intervals, using proprietary tools obtained from the vendor.

Figure 5.11 shows the disk capacity usage and the write amplification of the conventional enterprise SSD under these workloads. The fillseq benchmark completes in about two hours, filling 80% of the drive. During `fillseq`, a memtable is buffered in memory and flushed to disk, and no compaction happens due to the ordered nature of the inserted keys: moving an SST to a higher level is a fast rename operation. As a result, write amplification is 1 during the first two hours.

Once the `fillseq` benchmark completes, the `overwrite` benchmark starts, resulting in continuous concurrent compactions, taking almost 40 hours to complete. As Figure 5.11 shows, the

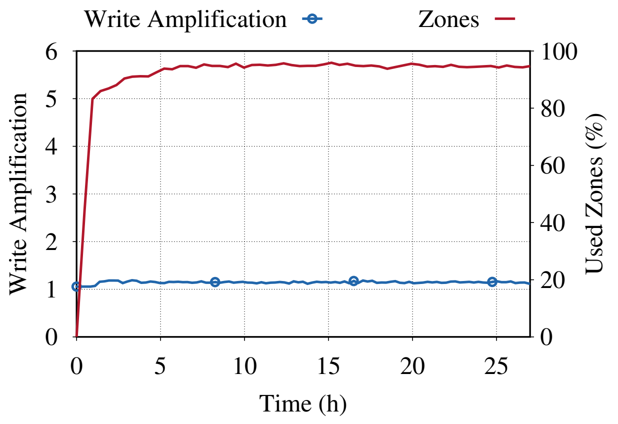


Figure 5.12: Device write amplification of the prototype ZNS SSD during RocksDB benchmarks. First, the `fillseq` benchmark sequentially inserts 1.8 billion key-value pairs; it completes in about an hour, filling the drive up to 80%, and the write amplification is 1 during this period. Next, the `overwrite` benchmark overwrites 1.8 billion key-value pairs in about 26 hours, during which the write amplification stays around 1.

write amplification gradually increases to around 5 and stays there until the end. The primary reason for this surprisingly high write amplification is the mixing of concurrent sequential streams. Due to NAND flash constraints the SSD has to buffer certain amount of data—either in NVRAM or capacitor-backed RAM—before writing data to the NAND flash. Hence, although memtable flushes and compactions result in sequential writes, when they happen concurrently, the SSD mixes these streams into a single internal buffer that it then writes across NAND flash dies to increase parallelism. The data from different streams end up in different erase blocks, effectively resulting in random write behavior, which eventually leads to garbage collection and high write amplification.

5.6.3 Avoiding the Block Interface Tax with RocksDB on ZNS SSDs

To demonstrate how RocksDB running on BlueFS avoids high write amplification, we repeat the same experiment on the ZNS SSD, but using fewer keys due to prototype device's smaller capacity, and we measure the write amplification at the bottom of the shim layer.

Figure 5.12 shows the zone usage and the write amplification measured at the shim layer of the prototype ZNS SSD under the similar workload. The `fillseq` benchmark completes in about an hour during which the write amplification is 1 because no compaction occurs and the generated SSTs are squarely filled into zones.

Once the `fillseq` benchmark completes, the `overwrite` benchmark starts, during which concurrent compactions produce SSTs smaller than the zone size. Like we described before (§ 5.3.2), we pad the zones containing small SSTs with nulls to avoid read disturbances (§ 5.2.1). Therefore, we compute the write amplification as (data bytes + pad bytes) / (data bytes). As Figure 5.12 shows, the write amplification in ZNS SSD is close to 1.2 on average.

Finally, although the benchmark we run on the ZNS SSD has 3.8× fewer keys, it completes only 1.5× quicker than when running on the enterprise SSD with an FTL. There are two reasons for this.

First, the ZNS SSD prototype is based on a client SSD [50] that has less internal parallelism than the enterprise SSD. Second, and most importantly, the shim layer in the prototype is still running on top of an FTL of the client SSD. So while our measurements at the bottom of the shim layer demonstrate a significant reduction in write amplification, this reduction does not translate to faster completion time because writes to zones are still handled by the FTL running down below. In a real ZNS SSD, reduced write amplification will result in a significantly faster completion time.

In summary, we demonstrate how LSM-Trees in general, and RocksDB in particular, can leverage the zone interface to avoid the block interface tax and achieve a significant reduction in the write amplification. A work based on our contributions described here is currently being merged to the RocksDB project [62]. We leave it as a future work—to the time when production ZNS SSDs become available—to demonstrate how reduced write amplification translates into better performance.

5.7 Handling Data Path—BlueStore on Zoned Storage

The work described so far in this chapter covered handling the *metadata path* for BlueStore on zoned storage, which entailed getting RocksDB to run on zoned devices. In this section we describe handling the *data path* for BlueStore on zoned storage, which entails getting BlueStore to manage data on zoned devices.

As described before (§ 4.2), out of the box Ceph provides three services on top of RADOS: the RADOS Gateway (RGW), an object storage similar to Amazon S3 [8]; the RADOS Block Device (RBD), a virtual block device similar to Amazon EBS [7]; and CephFS, a distributed file system with POSIX semantics.

Although all of these services run on top of the RADOS layer, they all result in different I/O patterns with varying complexity at the RADOS layer. In this work we target the service that results in the simplest I/O patterns—the RGW service—which provides operations for reading and writing variable-sized immutable objects. Choosing to support RGW service restricts the object operations at the RADOS layer to creating, deleting, truncating, appending, and fully overwriting objects.

5.7.1 Additions and Modifications to BlueStore

To manage objects on zoned devices, we introduce a new space allocator, a new freelist manager, and a new garbage collector to BlueStore. Next, we give a high-level overview of these new components and other changes that we introduced to BlueStore.

ZonedAllocator

BlueStore already comes with several space allocators optimized for different workloads (§ 4.4). None of these allocators, however, can work with zoned devices due to a more restrictive interface. More specifically, since block devices allow in-place overwrite of arbitrary blocks, no region of a block device ever becomes *stale* due to deletion or overwrite—it only becomes *free* and immediately reusable. Thus, existing allocators designed for block devices do not track stale regions, which regularly occur on zoned devices.

We introduce *ZonedAllocator*, which keeps in memory two bits of information per zone: a write pointer and the number of stale bytes within the zone. When ZonedAllocator receives an allocation request, it finds the first zone that can fit the allocation request—starting at the lowest numbered zone. Once a zone is found, ZonedAllocator returns the value of the in-memory write pointer for that zone to the caller and advances the write pointer by the size of the allocation. The in-memory write pointer in ZonedAllocator is unrelated to the actual write pointer within the device, which advances when data is written to the allocated space. When an object gets deleted or truncated, ZonedAllocator receives a deallocation request with an offset and size. In this case ZonedAllocator computes from the offset the zone in which the deallocation happens, and it increments the number of stale bytes for that zone by the amount of the passed in size.

The concurrent nature of storage backend operations in Ceph creates a challenge that is specific to ZonedAllocator. As explained before (§ 4.2), RADOS objects are sharded among *placement groups* (PGs), and multiple PGs are associated with a single *object storage device* (OSD), which runs the storage backend—BlueStore in our case. BlueStore maintains a group of threads for handling object writes to all of the PGs associated with the OSD. Although all of the object writes within a PG are serialized, object writes to different PGs may happen concurrently in different threads. Furthermore, BlueStore writes an object using a transaction mechanism that goes through multiple states, where *space allocation* and *data write* are two separate states. This poses a problem for ZonedAllocator because two threads may allocate space from the same zone in one order, but they may get rescheduled and write data to the zone in a different order: for example, thread A may call to ZonedAllocator and receive the offset 0 from a zone, and then thread B may receive the offset 65,536 from the same zone. Later the threads may get rescheduled and thread B may run first and issue a write to offset 65,536 followed by thread A issuing a write to offset 0—a violation of the sequential write requirement.

One simple solution to this problem may seem to pin the threads to the zones, so that writes within a zone are always sequential. This solution, however, is costly for HM-SMR HDDs because given the 256 MiB zone size, 8 threads, for example, may issue writes within a 2 GiB span, causing significant seek overhead. We implemented this solution and found over 50% reduction in write throughput on a single OSD.

A better solution is to utilize the new ZONE APPEND command in the zone interface. This command, inspired by *Nameless Writes* [223], was introduced to avoid in-kernel lock contention when multiple writers are writing to the same zone in a high-end ZNS SSD [16]. The ZONE APPEND command works as follows: A writer issues the command specifying the data and the zone number to which the data should be written. The drive (1) internally decides where within the zone to write the data, (2) writes the data, and (3) returns the offset of the write to the host. Thus, multiple threads can concurrently issue ZONE APPEND to the same zone and the kernel does not have to serialize access to the zone using a lock.

The ZONE APPEND command effectively moves space allocation to the device and further simplifies ZonedAllocator. Now, ZonedAllocator only tracks how much space is left in each zone and returns the zone number—instead of an offset—in which the space was allocated. Rescheduling of two threads performing concurrent space allocation and data write is not a problem anymore because the offsets of data writes are determined by the device at the time of writes. Unfortunately, as of this writing the ZONE APPEND command has not stabilized in the Linux kernel;

therefore, we solved this problem by combining the space allocate and the data write steps in BlueStore into a single atomic step, using a lock.

ZonedFreelistManager

The freelist manager module in BlueStore is responsible for a persistent representation of the parts of the disk currently in use (§ 5.7). Like the existing space allocators, the existing freelist manager is not adequate for zoned devices, for the same reason.

We introduce ZonedFreelistManager, whose design mostly parallels that of ZonedAllocator: it also keeps a write pointer and the number of stale bytes for each zone. Unlike ZonedAllocator, however, ZonedFreelistManager stores these persistently in RocksDB in the `Z` namespace using the zone number as the key and two concatenated 32-bit unsigned integers as the value.

When the OSD boots, ZonedFreelistManager loads the state of each zone—the write pointer and the number of stale bytes—from RocksDB. ZonedAllocator then reads the persistent state of each zone from ZonedFreelistManager and initializes the in-memory state of each zone.

ZonedFreelistManager receives the same allocation request that ZonedAllocator has received, after the data has been safely written to disk in the allocated space and in-device write pointer has advanced. At this point ZonedFreelistManager advances the write-pointer in RocksDB to match the in-device write pointer. Similarly, ZonedFreelistManager receives the same deallocation request that ZonedAllocator has received, after the metadata of the object residing in the deallocated space has been deleted from RocksDB. At this point ZonedFreelistManager computes the corresponding zone and increments the number of stale bytes in RocksDB for that zone by the amount of the passed in size. To avoid the cost of point queries, ZonedFreelistManager uses the merge operator in RocksDB [91] for updating the write pointer and the number of stale bytes.

ZonedCleaner

We introduce *ZonedCleaner*, a simple garbage collector that reclaims stale space from the fragmented zones. We call it ZonedCleaner because BlueStore already contains a component called *GarbageCollector*, which serves an unrelated purpose: it defragments compressed objects that have been fragmented by too many overwrites (§ 4.5.3) by reading and rewriting them.

We make several changes to I/O paths in BlueStore for the proper operation of ZonedCleaner. BlueStore implements a transactional interface where a single transaction may create, delete, or overwrite multiple objects. A *transaction context* tracks the state related to a single transaction—it is created when a transaction starts and destroyed when a transaction completes. We maintain an in-memory map per *transaction context* from an object identifier to a list of offsets, and we update this map as follows: When a new object is created, we append its offset to the list for that object identifier. When an object is deleted, we append the negative of its offset to the list for that object identifier. And when an object is overwritten, we append the negative of its previous offset and its new offset to the list for that object identifier. When a transaction completes, we use the in-memory map to update persistent cleaning information in RocksDB, as described next.

In addition to the in-memory map, we also maintain persistent cleaning information about objects in RocksDB: for every object we store a key-value pair in the `G` namespace, where a key is the concatenation of the zone number in which the object is located and the object identifier, and the value is the offset of the object within that zone. When a transaction completes, we use the in-memory map from the transaction context to update the persistent cleaning information in RocksDB as follows: we go through the in-memory map and for every object identifier, we process the list of offsets; for negative offsets we take their absolute value, compute the zone number, and remove the keys formed by the concatenation of the zone number and the object identifier; for positive offsets we compute the zone number and insert a new key-value pair where a key is the concatenation of the zone number and the object identifier, and the value is the offset. Hence, with these updates happening at the end of every transaction, the object identifiers of all live objects within the zone `ABC` can be found by querying the `G` namespace for keys that have `ABC` as the prefix.

With all the necessary cleaning information in place, the operation of ZonedCleaner is straightforward. It runs in a separate thread that starts when the OSD boots and goes to sleep until it is woken up by a trigger event—currently this event is the drive becoming 80% full—to perform cleaning. When it wakes up, ZonedCleaner asks ZonedAllocator for a zone number to clean. ZonedAllocator sorts the zones by the number of stale bytes and returns the zone number with the most stale bytes. ZonedCleaner then queries the `G` namespace for the keys that have the zone number as the prefix and obtains the object identifiers of all live objects within a zone. It then reads those objects from the fragmented zone, writes them to a new zone obtained from ZonedAllocator, resets the fragmented zone, and informs ZonedAllocator and ZonedFreelistManager of a newly freed zone.

Summary of Changes to BlueStore

Our initial set of changes to BlueStore described in this section has been merged to the Ceph project [71]. In this first iteration of changes we aimed to keep things simple but correct: ZonedAllocator is a simple allocator that aims to improve throughput by filling the disk starting at the outer diameter, and ZonedCleaner implements the simplest of the cleaning policies—the *greedy* cleaning policy [158]. We leave researching and designing a more optimized space allocation, data placement, and cleaning policies in a distributed setting as a future work.

Despite its simplicity, our initial set of changes are enough to demonstrate a key advantage of the zone interface—reducing the tail latency by controlling the garbage collection. We demonstrate this in the next section.

5.8 Evaluation of Ceph on HM-SMR HDDs

In this section we combine all the improvements described so far in the chapter and show how they collectively enable Ceph to avoid paying the block interface tax. To this end, we run a set of benchmarks on a Ceph cluster and demonstrate two things: First, Ceph is now more cost-effective—it can leverage the extra 20% capacity offered by SMR without suffering performance loss [48]. Second, Ceph is now more performant—it can reduce the tail latency of I/O operations by controlling the timing of garbage collection. Unfortunately, the COVID-19 pandemic [212] delayed the availability of production-quality ZNS SSDs; therefore, we are only able to demonstrate these improvements on HM-SMR HDDs.

Type	Vendor	Model	Capacity	Throughput
CMR	Hitachi	HUA72303	3 TB	160 MiB/s
DM-SMR	Western Digital	WD30EFAX, WD40EFAX, WD60EFAX	3 TB, 4 TB, 6 TB	186 MiB/s (avg.)
HM-SMR	HGST	HSH721414AL	14 TB	220 MiB/s

Table 5.1: HDDs used in evaluation and their bandwidth at the first 125 GiB of the LBA space. We measured both sequential read and sequential write throughput for all of the drives to be the same.

We run all experiments on an 8-node Ceph cluster connected with a Cisco Nexus 3264-Q 64-port QSFP+ 40GbE switch. Each node has a 16-core AMD Opteron 6272 2.1 GHz CPU, 128 GiB of RAM, and a Mellanox MCX314A-BCCT 40GbE NIC. All nodes run Linux kernel 5.5.9 on the Ubuntu 18.04 distribution and the Octopus release (v15.2.4) of Ceph. Although we developed our changes on the development version of Ceph, which is significantly ahead of the Octopus release (v15.2.4), we backported our changes to v15.2.4 for a fair comparison. We use the default Ceph configuration parameters unless otherwise noted.

In the experiments described next, we use three different Ceph hardware configurations called CMR, DM-SMR, and HM-SMR. The configurations are so called the hard drives used in their storage nodes. Table 5.1 shows the capacity and sequential I/O bandwidth of these drives. The CMR and HM-SMR cluster configurations use HUA72303 and HSH721414AL drives, respectively, on all of their eight nodes. The DM-SMR cluster configuration uses WD30EFAX drives on three nodes, WD40EFAX on three nodes, and WD60EFAX on two nodes. (Unfortunately, we could not purchase a uniform set of drives for the DM-SMR configuration because there was a limit of three drives per customer per drive type, due to the COVID-19 pandemic.) Each one of our experiments writes about 1 TB of aggregate data to the cluster, which roughly corresponds to 125 GiB of each drive. Hence, we measure the throughput of each drive by performing sequential I/O to the first 125 GiB of the drive and report it in Table 5.1. (For the DM-SMR configuration we measured the throughput of the WD30EFAX, WD40EFAX, and WD60EFAX drives as 180 MiB/s, 190 MiB/s, and 200 MiB/s, respectively, and reported their weighted average.)

5.8.1 RADOS Write Throughput

Our first experiment compares the performance of writes to the RADOS layer in each of these cluster configurations. Since we target the Amazon S3-like object storage (§ 5.7), we focus on large I/O sizes: we write objects of sizes 1 MiB, 2 MiB and 4 MiB using 128 threads. Figure 5.13 (a) shows the throughput for all I/O sizes and cluster configurations: we observe that DM-SMR is 17%, 34%, and 34% slower than CMR for I/O sizes of 1 MiB, 2 MiB, and 4 MiB, respectively, although raw sequential write throughput of DM-SMR drives is 16% higher (Table 5.1).

This result is not surprising. As explained before (§ 3.4.3), DM-SMR drives can detect a single sequential write stream and bypass the persistent cache; therefore, in Table 5.1 we achieve the maximal drive throughput. In the presence of multiple sequential streams, however, the operating system mixes chunks from different streams and sends them to the drive, which forces the drive to write data into the persistent cache and introduce garbage collection overhead (§ 5.5.3). Therefore, even though the benchmark writes large objects, DM-SMR drive suffers performance degradation.

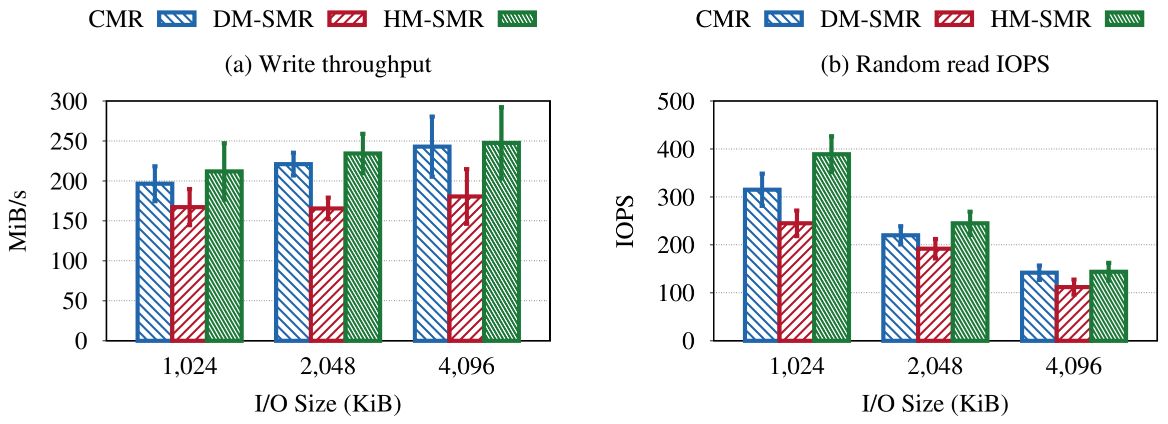


Figure 5.13: (a) Write throughput and (b) random read IOPS, at steady state to the RADOS layer on an 8-node Ceph cluster with CMR, DM-SMR, and HM-SMR configurations. The benchmark issues reads and writes using 128 threads with three different I/O sizes.

Hence, in this case, the block interface tax displays itself as a garbage collection overhead, even though no garbage collection is needed for the workload.

As Figure 5.13 (a) shows, HM-SMR does not suffer from any overhead because data is written directly to zones and no unnecessary garbage collection is performed. However, although Table 5.1 shows HM-SMR drive to have 37% higher sequential write throughput than the CMR drive, this difference is not reflected in Figure 5.13 (a)—HM-SMR has only slightly higher throughput than CMR. We believe there are two reasons for this. First, adding a lock to BlueStore transaction mechanism for making two separate states—space allocation and data write—a single atomic state introduces overhead (§ 5.7.1). This overhead, however, is temporary, and it is likely to disappear as the ZONE APPEND command stabilizes in the kernel and the lock is eliminated. Second, the HM-SMR implementation is the first working code and it has not been optimized as the CMR implementation. We expect the write throughput of HM-SMR to increase proportional to its throughput advantage over CMR as the HM-SMR implementation gets optimized.

5.8.2 RADOS Random Read IOPS

Our second experiment compares the performance of random reads at the RADOS layer in the same cluster configurations. We read the objects that were written in the previous experiment (§ 5.8.1) using 128 threads. Figure 5.13 (b) shows the random read IOPS for all object sizes and cluster configurations: we observe that DM-SMR is 29%, 14%, and 27% slower than CMR for I/O sizes of 1 MiB, 2 MiB, and 4 MiB, respectively, although raw sequential read throughput of DM-SMR drives is 16% higher (Table 5.1).

Unlike the previous result (§ 5.8.1), however, this result is surprising because we do not expect garbage collection operations to interfere with read operations: we paused for a few hours after the write benchmark completed before starting the read benchmark, to ensure that all pending garbage collection operations have completed. We leave the exploration of this phenomenon as a future work, and for now we speculate that to reduce the garbage collection work, these DM-SMR drives end up fragmenting a single RADOS object into individual writes they receive for the

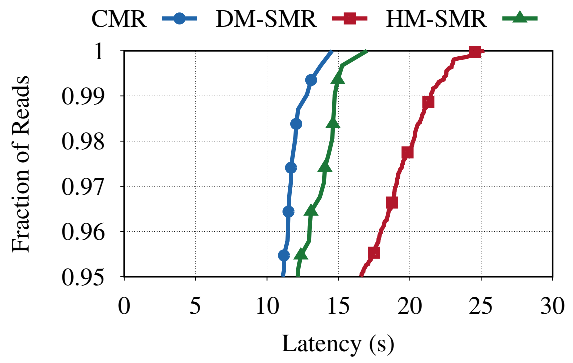


Figure 5.14: 95th and above percentile latencies of random 1 MiB object reads at the RADOS layer during garbage collection on an 8-node Ceph cluster with CMR, DM-SMR, and HM-SMR configurations. The benchmark issues reads using 128 threads. Garbage collection happens within the device in DM-SMR and in the host in HM-SMR. No garbage collection happens in CMR.

object, and the drives incur seek overhead when reading the objects. In any case, this is another instance of block interface tax that should not happen in the first place.

As Figure 5.13 (b) shows, HM-SMR does not suffer from any overhead because objects are read sequentially from the zones exploiting the full throughput of the drive. HM-SMR has 23% and 11% higher IOPS than CMR for 1 MiB and 2 MiB objects, respectively, but its IOPS is similar to that of CMR for 4 MiB objects. Again, we expect the read throughput of HM-SMR to increase proportional to its throughput advantage over CMR as the HM-SMR implementation gets optimized.

5.8.3 Tail Latency of RADOS Random Reads During Garbage Collection

Our third and last experiment compares the tail latency of reads during garbage collection in the same cluster configurations. We demonstrate that by moving garbage collection to the host, the zone interface allows a distributed storage system to reduce tail latency by leveraging the control over the timing of garbage collection and the redundancy of data.

For DM-SMR and CMR we run the experiment as follows: We first write half a million 1 MiB objects using 128 threads. After these writes complete, we start writing another half a million objects using 128 threads while at the same time randomly reading using 128 threads the objects written earlier. Since DM-SMR drives regularly perform garbage collection during writes, we expect the garbage collection to affect the tail latency of random reads. Obviously, no garbage collection happens in the CMR case.

For HM-SMR we run the experiment as follows: To trigger early garbage collection, we restrict the number of zones to 500 in each host. We then write one million 1 MiB objects almost filling all of the zones in all of the nodes and then delete half a million of these objects. To simulate a controlled garbage collection, we then start garbage collection on two of the nodes and start reading half a million objects that weren't deleted. We use Ceph's OSD affinity mechanism [30] to redirect reads only to the nodes that are not performing garbage collection.

Figure 5.14 shows 95th and above percentile latencies of random 1 MiB object reads from RA-

DOS: we observe that 99th and above tail latency of HM-SMR is 53% lower than DM-SMR. Furthermore, the HM-SMR latency is within 13% of the CMR latency, and we expect it to improve with future optimizations.

5.9 Summary

In this chapter, we leveraged the agility of BlueStore, a special-purpose storage backend in Ceph, to swiftly embrace the zone interface and liberate Ceph from the block interface tax. By adapting BlueStore to work on HM-SMR drives, which expose the zone interface, we demonstrated that Ceph can utilize the extra capacity offered by SMR at high throughput and low tail latency by avoiding the garbage collection overhead of DM-SMR drives, which expose the block interface through emulation.

As a part of this work, we also liberated RocksDB, a widely used key-value store powering many large-scale internet services, from the block interface tax. Specifically, we demonstrated that LSM-Trees in general, and RocksDB in particular, suffer unnecessary in-device garbage collection and high write amplification when emulating a block interface through a translation layer—all of which can be eliminate by adopting the zone interface.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this dissertation, we propose a new approach to distributed storage system design. More specifically, we demonstrate that to unlock the full potential of modern data center storage devices the distributed storage systems should abandon the decades old conventions in their storage backends—the reliance on the block interface and general-purpose file systems—and embrace the novel zone interface and special-purpose storage backend design.

We first argue that the block interface is a poor match for modern data center storage devices. The block interface, which was designed for early hard disk drives, allows random updates of small blocks, and it has been the dominant interface for the past three decades. As a result, almost every file system in use today was developed for the block interface. Modern storage technologies such as solid-state drives or Shingled Magnetic Recording (SMR) hard drives, on the other hand, are different: unlike early storage media, which contained randomly writable bits, newer storage media consist of large regions that must be written sequentially. So that we could continue using our current file systems, we currently shoehorn the block interface onto modern storage devices by emulating it using a translation layer inside drives. This emulation, however, introduces significant cost and performance overheads. These overheads can be eliminated by adopting the novel zone interface, which exposes sequential regions to the host. But since the zone interface is not compatible with the block interface, no current file system can run on devices that expose the zone interface.

The performance overhead of emulating the block interface using translation layers is well studied in solid-state drives. In our work, we study the emulation overhead in modern high-capacity hard drives that use SMR, which are also known as drive-managed SMR (DM-SMR) drives. To this end, we introduce Skylight, a novel methodology for measuring and characterizing DM-SMR drives. We develop a series of micro-benchmarks for this characterization and augment these timing measurements with a novel technique that tracks actual drive head movements using a high-speed camera. Using our approach, we fully reverse engineer how the translation layer emulates the block interface in DM-SMR drives. We discover that these drives can handle sustained sequential writes with no overhead, but they suffer orders of magnitude performance degradation in the presence of sustained random writes.

We then leverage the results of our characterization work to improve the performance of ext4, a general-purpose file system, on DM-SMR drives. We do so because ext4 is used by many distributed storage systems as a storage backend, and DM-SMR drives increase capacity by 20% or more over conventional drives. Therefore, alleviating the emulation overhead by optimizing ext4 can increase cost-effectiveness of data storage without sacrificing performance in distributed storage systems. Consequently, we introduce ext4-lazy, an extension of ext4, which significantly improves the performance over ext4 on key workloads when running on top of DM-SMR drives. Ext4-lazy achieves this by introducing a well-chosen, small but effective change that utilizes the existing journaling mechanism: unlike ext4, which writes metadata twice—first, sequentially to the journal and second, randomly to disk—ext4-lazy keeps often-updated metadata in the journal and avoids the second write, thereby significantly reducing random writes. Although ext4-lazy achieves remarkable performance improvements, it also shows that eliminating the emulation overhead using evolutionary changes is hard: on workloads with non-trivial amount of random data writes, DM-SMR throughput running ext4-lazy still suffers compared to the throughput of conventional drives running ext4. Hence, the only way to eliminate the emulation overhead is to not to emulate and adopt the backward-incompatible zone interface.

One approach to adopting the zone interface is to modify current file systems that were designed for the block interface to work with the zone interface. This approach did not pan out for major file systems because it required a major redesign. Hence, the only remaining approach for adopting the zone interface is to design a new general-purpose file system for the zone interface. However, before designing yet another file system for running a distributed storage backend on top, we take a step back and ask the following question: how appropriate is the file system interface for building distributed storage backends?

To answer this question, we perform a longitudinal study of storage backends in Ceph, a widely used distributed storage system, over ten years. We find that for eight of these ten years, the Ceph project has followed the conventional wisdom of building its storage backend on top of local file systems. This is a preferred choice for most distributed storage systems because it allows them to benefit from the convenience and maturity of battle-tested file system code. The experience of the Ceph team, however, shows that this comes with a high performance overhead. More specifically, first, it is hard to develop efficient transaction mechanism on top of a general-purpose file system, and second, the file system metadata management is too heavyweight—it does not scale to the needs of a distributed storage backend. Another disadvantage of relying on file systems is that they take a long time to mature and once mature, they are averse to major changes. Effectively adopting new hardware, on the other hand, often does require major changes. As a result, after eight years the Ceph team departed from the conventional wisdom and in just two years developed BlueStore, a clean-slate special-purpose storage backend that outperformed existing backends.

Finally, we demonstrate that a distributed storage system that is not bound by the progress of general-purpose file systems, can quickly adopt the zone interface and unlock the full potential of modern storage devices. To this end, we adapt BlueStore to the zone interface in two steps. First, we handle the metadata path in BlueStore and extend RocksDB, a key-value store that BlueStore uses for storing metadata, to run on the zone interface. RocksDB is an instance of a data structure called Log-Structured Merge Tree (LSM-Tree), and LSM-Trees are at the core of many large-scale online services and applications. As a result of this work, we demonstrate how RocksDB leverages smart data placement enabled by the zone interface to eliminate write amplification inside solid-state drives by 5×. Second, we handle the data path in BlueStore and introduce a garbage collector that the host can explicitly control. We combine this control with the redundancy of data in a distributed setting to eliminate garbage collection in some cases and to reduce the tail latency of I/O operations in other cases.

In 2015, Dave Chinner, the maintainer of XFS—a widely used high-performance file system—claimed that in 20 years all current file systems will be legacy file systems, and since it takes a decade to mature a file system, we should start working on future file systems soon [36]. While this is a good call, we think it would be a mistake to build yet another monolithic POSIX file system that runs on everything, from our smartphones to large-scale distributed storage systems powering the cloud. The key implication of our work is that when it comes to distributed storage systems—which power the enterprise and public clouds and are expected to host over 80% of all of data [155] by 2025—we should build specialized storage systems that leverage the available hardware and the domain knowledge to the fullest, delivering the best performance and greatest user experience.

6.2 Future Work

This dissertation is the first step towards improving the cost-effectiveness and performance of distributed storage systems through elimination of layers on top of raw storage medium. It demonstrates that abandoning the block interface and general-purpose file systems is feasible and useful when designing a storage backend, but by starting from scratch on a raw storage device with a new interface it brings up new research questions, a couple of which we describe next.

6.2.1 Index Structures for Zoned Devices

BlueStore is the first distributed storage backend that stores all of its metadata, including low-level metadata, such as extent bitmaps, in a key-value store—RocksDB. Although RocksDB was central to improving the metadata performance in BlueStore, its compaction overhead has become the new performance bottleneck. In addition, with low I/O latency on high-end NVMe SSDs, the CPU time spent in serializing and deserializing data when reading from or writing to RocksDB is becoming significant. Finally, RocksDB compaction has 10× or more application-level write amplification, which is substantial considering the upcoming high-capacity SSDs that are based on QLC NAND, which is even more susceptible to wear. Hence, the research question is how to design an index structure that exploits the properties of storage backend workloads to reduce write amplification and also adheres to the zone interface constraints?

6.2.2 Shrinking Capacity Zoned Devices

Most storage systems are built assuming that the raw capacity of the underlying storage device does not change. This assumption precludes the possibility of a partially failed device that can continue to provide usable storage. It was forged by early single-head HDDs that became unusable after a head crash—it does not hold for SSDs and even for modern HDDs with multiple heads and actuators. Most importantly, it comes at a high cost.

Enterprise SSDs overprovision 28% of NAND flash for efficient garbage collection. As pages wear out, some of the overprovisioned flash gets used as replacement. This increases write amplification due to less efficient garbage collection and further accelerates wear-out of the remaining pages. After losing the overprovisioned capacity, a drive declares itself dead, since it is unable to meet the vendor-specified QoS, despite having over 70% usable capacity.

The zone interface accounts for the worn-out pages, allowing ZNS SSDs to shrink the zone capacity and continue to operate normally otherwise. Yet the file systems that are used as storage backends in most distributed storage systems, as well as BlueStore, are unable to cope with shrinking storage device. It appears that a distributed storage system is well suited to coping with shrinking device capacity due to redundancy of data and availability of more storage devices for data migration, but identifying the right interface and mechanisms for this purpose requires further research.

Bibliography

[1] Abutalib Aghayev and Peter Desnoyers. Log-Structured Cache: Trading Hit-Rate for Storage Performance (and Winning) in Mobile Devices. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324625. doi: 10.1145/2527792.2527797. URL https://doi.org/10.1145/2527792.2527797. [Cited on page 26.]

[2] Abutalib Aghayev and Peter Desnoyers. Skylight—A Window on Shingled Disk Operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 135–149, Santa Clara, CA, USA, February 2015. USENIX Association. ISBN 978-1-931971-201. URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/aghayev. [Cited on page 3.]

[3] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving Ext4 for Shingled Disks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 105–120, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-36-2105. URL https://www.usenix.org/conference/fast17/technical-sessions/presentation/aghayev. [Cited on page 77.]

[4] Abutalib Aghayev, Sage Weil, Greg Ganger, and George Amvrosiadis. Reconciling LSM-Trees with Modern Hard Drives using BlueFS. Technical Report CMU-PDL-19-102, CMU Parallel Data Laboratory, April 2019. URL http://www.pdl.cmu.edu/PDL-FTP/FS/CMU-PDL-19-102_abs.shtml. [Cited on page 5.]

[5] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 353–369, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359656. URL https://doi.org/10.1145/3341301.3359656. [Cited on page 4.]

[6] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. The Case for Custom Storage Backends in Distributed Storage Systems. *ACM Trans. Storage*, 16(2), May 2020. ISSN 1553-3077. doi: 10.1145/3386362. URL https://doi.org/10.1145/3386362. [Cited on page 4.]

[7] Amazon.com, Inc. Amazon Elastic Block Store. https://aws.amazon.com/ebs/, 2019. [Cited on pages 61 and 95.]

[8] Amazon.com, Inc. Amazon S3. https://aws.amazon.com/s3/, 2019. [Cited on pages 61 and 95.]

[9] Ahmed Amer, Darrell D. E. Long, Ethan L. Miller, Jehan-Francois Paris, and S. J. Thomas Schwarz. Design Issues for a Shingled Write Disk System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496991. URL http://dx.doi.org/10.1109/MSST.2010.5496991. [Cited on pages 9, 28, and 35.]

[10] Andreas Dilger. Lustre Metadata Scaling. http://storageconference.us/2012/Presentations/T01.Dilger.pdf, 2012. [Cited on pages 1 and 4.]

[11] AskUbuntu. Is there any faster way to remove a directory than "rm -rf"? http://askubuntu.com/questions/114969, 2012. [Cited on page 49.]

[12] Jens Axboe. Queue sysfs files. https://www.kernel.org/doc/Documentation/block/queue-sysfs.txt, February 2009. [Cited on page 73.]

[13] Jens Axboe. Flexible I/O Tester. git://git.kernel.dk/fio.git, 2016. [Cited on pages 14 and 73.]

[14] Jens Axboe. Throttled Background Buffered Writeback. https://lwn.net/Articles/698815/, August 2016. [Cited on page 67.]

[15] 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. *Trans. Storage*, 4(3):8:1–8:28, November 2008. ISSN 1553-3077. doi: 10.1145/1416944.1416947. URL http://doi.acm.org/10.1145/1416944.1416947. [Cited on page 31.]

[16] Matias Bjørling. Zone Append: A New Way of Writing to Zoned Storage. Santa Clara, CA, February 2020. USENIX Association. [Cited on page 96.]

[17] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th international systems and storage conference (SYSTOR)*. ACM, 2013. [Cited on page 81.]

[18] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-36-2. URL https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling. [Cited on page 80.]

[19] Matias Bjørling et al. Zoned Namespaces. Technical Proposal 4053, NVM Express, June 2020. Available from https://nvmexpress.org/wp-content/uploads/NVM-Express-1.4-Ratified-TPs.zip. [Cited on pages 80 and 81.]

[20] Artem Blagodarenko. Scaling LDISKFS for the future. https://www.youtube.com/watch?v=ubbZGpxV6zk, 2016. [Cited on pages 60 and 78.]

[21] Artem Blagodarenko. Scaling LDISKFS for the future. Again. https://www.youtube.com/watch?v=HLfEd0_Dq0U, 2017. [Cited on pages 60 and 78.]

[22] Luc Bouganim, Bjorn Jónsson, and Philippe Bonnet. uFLIP: understanding flash IO pat-

terns. In *Proceedings of the Int'l Conf. on Innovative Data Systems Research (CIDR)*, Asilomar, California, USA, 2009. [Cited on page 10.]

[23] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855741.1855745`. [Cited on page 40.]

[24] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924943.1924944`. [Cited on page 40.]

[25] Frederick P Brooks Jr. No Silver Bullet—Essence and Accident in Software Engineering, 1986. [Cited on page 4.]

[26] Btrfs. Btrfs Changelog. `https://btrfs.wiki.kernel.org/index.php/Changelog`, 2019. [Cited on pages 63 and 77.]

[27] David C. [ceph-users] Luminous | PG split causing slow requests. `http://lists.ceph.com/pipermail/ceph-users-ceph.com/2018-February/024984.html`, 2018. [Cited on page 66.]

[28] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the first Dutch International Symposium on Linux*, volume 1, 1994. [Cited on page 41.]

[29] Yuval Cassuto, Marco A. A. Sanvido, Cyril Guyot, David R. Hall, and Zvonimir Z. Bandic. Indirection Systems for Shingled-recording Disk Drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–14, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496971. URL `http://dx.doi.org/10.1109/MSST.2010.5496971`. [Cited on pages 9, 10, 12, 13, 15, 16, 19, 23, 28, 35, and 55.]

[30] ceph.io. Ceph: get the best of your SSD with primary affinity. `https://ceph.io/geen-categorie/ceph-get-the-best-of-your-ssd-with-primary-affinity/`, 2015. [Cited on page 101.]

[31] Luoqing Chao and Thunder Zhang. Implement Object Storage with SMR based key-value store. `https://www.snia.org/sites/default/files/SDC15_presentations/smr/QingchaoLuo_Implement_Object_Storage_SMR_Key-Value_Store.pdf`, 2015. [Cited on page 79.]

[32] 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 Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-511-6. doi: 10.1145/1555349.1555371. URL `http://doi.acm.org/10.1145/1555349.1555371`.

[Cited on page 10.]

[33] Stepehen P. Morgan Chi-Young Ku. An SMR-aware Append-only File System. In *Storage Developer Conference*, Santa Clara, CA, USA, September 2015. [Cited on page 39.]

[34] Dave Chinner. XFS Delayed Logging Design. https://www.kernel.org/doc/Documentation/filesystems/xfs-delayed-logging-design.txt, 2010. [Cited on page 63.]

[35] Dave Chinner. SMR Layout Optimization for XFS. http://xfs.org/images/f/f6/Xfs-smr-structure-0.2.pdf, March 2015. [Cited on pages 5 and 39.]

[36] Dave Chinner. XFS: There and Back... ...and There Again? In *Vault Linux Storage and File System Conference*, Boston, MA, USA, April 2016. [Cited on pages 39 and 105.]

[37] Dave Chinner. Re: pagecache locking (was: bcachefs status update) merged). https://lkml.org/lkml/2019/6/13/1794, June 2019. [Cited on page 76.]

[38] Alibaba Clouder. Alibaba Deploys Alibaba Open Channel SSD for Next Generation Data Centers. https://www.alibabacloud.com/blog/alibaba-deploys-alibaba-open-channel-ssd-for-next-generation-data-centers_593802, July 2018. [Cited on page 80.]

[39] William Cohen. How to avoid wasting megabytes of memory a few bytes at a time. https://developers.redhat.com/blog/2016/06/01/how-to-avoid-wasting-megabytes-of-memory-a-few-bytes-at-a-time/, 2016. [Cited on page 76.]

[40] Jonathan Darrel Coker and David Robison Hall. Indirection memory architecture with reduced memory requirements for shingled magnetic recording devices, November 5 2013. US Patent 8,578,122. [Cited on pages 10, 15, and 35.]

[41] Douglas Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. ISSN 0360-0300. doi: 10.1145/356770.356776. URL http://doi.acm.org/10.1145/356770.356776. [Cited on page 82.]

[42] The Kernel Development Community. Switching Scheduler. https://www.kernel.org/doc/html/latest/block/switching-sched.html, 2020. [Cited on page 84.]

[43] Jonathan Corbet. Supporting transactions in Btrfs. https://lwn.net/Articles/361457/, Nov 2009. [Cited on pages 63 and 77.]

[44] Jonathan Corbet. No-I/O dirty throttling. https://lwn.net/Articles/456904/, August 2011. [Cited on page 67.]

[45] Jonathan Corbet. Kernel quality control, or the lack thereof. https://lwn.net/Articles/774114/, December 2018. [Cited on page 4.]

[46] Jonathan Corbet. PostgreSQL's fsync() surprise. https://lwn.net/Articles/752063/, 2018. [Cited on page 76.]

[47] Jeffrey Dean and Luiz AndrÃľ Barroso. The Tail at Scale. *Communications of the ACM*, 56: 74–80, 2013. URL http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext. [Cited on page 5.]

[48] Wido den Hollander. Do not use SMR disks with Ceph. https://blog.widodh.nl/2017/

02/do-not-use-smr-disks-with-ceph/, 2017. [Cited on page 98.]

[49] Linux Device-Mapper. Device-Mapper Resource Page. https://sourceware.org/dm/, 2001. [Cited on page 10.]

[50] Western Digital. Western Digital PC SN720 NVMe SSD. https://www.westerndigital.com/products/internal-drives/pc-sn720-ssd, 2019. [Cited on pages 93 and 95.]

[51] Western Digital. Zoned Storage. http://zonedstorage.io, 2019. [Cited on page 4.]

[52] Anton Dmitriev. [ceph-users] All OSD fails after few requests to RGW. http://lists.ceph.com/pipermail/ceph-users-ceph.com/2017-May/017950.html, 2017. [Cited on page 66.]

[53] Elizabeth A Dobisz, Z.Z. Bandic, Tsai-Wei Wu, and T. Albrecht. Patterned Media: Nanofabrication Challenges of Future Disk Drives. *Proceedings of the IEEE*, 96(11):1836–1846, November 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.2007600. [Cited on page 9.]

[54] Siying Dong. Direct I/O Close() shouldn't rewrite the last page. https://github.com/facebook/rocksdb/pull/4771, 2018. [Cited on page 84.]

[55] DRAMeXchange. NAND Flash Spot Price, September 2014. URL http://dramexchange.com. http://dramexchange.com. [Cited on page 9.]

[56] The Economist. The world's most valuable resource is no longer oil, but data. https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data, 2017. [Cited on page 1.]

[57] Jake Edge. Ideas for supporting shingled magnetic recording (SMR). https://lwn.net/Articles/592091/, Apr 2014. [Cited on page 39.]

[58] Jake Edge. The OrangeFS distributed filesystem. https://lwn.net/Articles/643165/, 2015. [Cited on pages 2, 4, and 59.]

[59] Jake Edge. Filesystem support for SMR devices. https://lwn.net/Articles/637035/, Mar 2015. [Cited on page 39.]

[60] Jake Edge. XFS: There and back ... and there again? https://lwn.net/Articles/638546/, Apr 2015. [Cited on pages 4 and 68.]

[61] 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, pages 251–266, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: 10.1145/224056.224076. URL http://doi.acm.org/10.1145/224056.224076. [Cited on page 59.]

[62] Facebook. RocksDB. https://github.com/facebook/rocksdb/, 2020. [Cited on pages 82 and 95.]

[63] Facebook Inc. Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks, 2018. [Cited on page 86.]

[64] Facebook Inc. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2018. [Cited on page 87.]

[65] Robert M Fallone and William B Boyle. Data storage device employing a run-length map-

ping table and a single address mapping table, May 14 2013. US Patent 8,443,167. [Cited on page 35.]

[66] Tim Feldman. Personal communication, August 2014. [Cited on page 14.]

[67] Tim Feldman. Host-Aware SMR, November 2014. Available from `https://www.youtube.com/watch?v=b1yqjV8qemU`. [Cited on page 18.]

[68] Timothy Richard Feldman. Dynamic storage regions, February 14 2011. US Patent App. 13/026,535. [Cited on pages 15, 34, and 35.]

[69] Andrew Fikes. Storage Architecture and Challenges. `https://cloud.google.com/files/storage_architecture_and_challenges.pdf`, 2010. [Cited on page 71.]

[70] Mary Jo Foley. Microsoft readies new cloud SSD storage spec for the Open Compute Project. `https://www.zdnet.com/article/microsoft-readies-new-cloud-ssd-storage-spec-for-the-open-compute-project/`, March 2018. [Cited on page 80.]

[71] The Ceph Foundation. Ceph. `https://github.com/ceph/ceph/`, 2020. [Cited on page 98.]

[72] FreeNAS. ZFS and lots of files. `https://forums.freenas.org/index.php?threads/zfs-and-lots-of-files.7925/`, 2012. [Cited on page 49.]

[73] Sanjay Ghemawat and Jeff Dean. LevelDB. `https://github.com/google/leveldb`, 2019. [Cited on page 56.]

[74] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945450. URL `http://doi.acm.org/10.1145/945445.945450`. [Cited on pages 2, 4, and 59.]

[75] Garth Gibson and Greg Ganger. Principles of Operation for Shingled Disk Devices. Technical Report CMU-PDL-11-107, CMU Parallel Data Laboratory, April 2011. URL `http://repository.cmu.edu/pdl/7`. [Cited on pages 9 and 79.]

[76] Garth Gibson and Milo Polte. Directions for Shingled-Write and Two-Dimensional Magnetic Recording System Architectures: Synergies with Solid-State Disks. Technical Report CMU-PDL-09-104, CMU Parallel Data Laboratory, May 2009. URL `http://repository.cmu.edu/pdl/7`. [Cited on page 35.]

[77] Jongmin Gim and Youjip Won. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. *ACM Transactions on Storage (TOS)*, 6(2):6:1–6:26, July 2010. ISSN 1553-3077. doi: 10.1145/1807060.1807063. URL `http://doi.acm.org/10.1145/1807060.1807063`. [Cited on pages 10 and 36.]

[78] David Hall, John H Marcos, and Jonathan D Coker. Data Handling Algorithms For Autonomous Shingled Magnetic Recording HDDs. *IEEE Transactions on Magnetics*, 48(5): 1777–1781, 2012. [Cited on pages 9, 10, 12, and 35.]

[79] David Robison Hall. Shingle-written magnetic recording (SMR) device with hybrid E-region, April 1 2014. US Patent 8,687,303. [Cited on pages 13, 15, 28, and 35.]

[80] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien,

and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, Santa Clara, CA, 2016. USENIX Association. ISBN 978-1-931971-28-7. URL https://www.usenix.org/conference/fast16/technical-sessions/presentation/hao. [Cited on pages 1, 3, 5, and 80.]

[81] Weiping He and David H. C. Du. Novel Address Mappings for Shingled Write Disks. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, pages 5–5, Berkeley, CA, USA, 2014. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2696578.2696583. [Cited on page 35.]

[82] Christoph Hellwig. XFS: The Big Storage File System for Linux. *USENIX ;login issue*, 34(5), 2009. [Cited on page 62.]

[83] HGST. HGST Unveils Intelligent, Dynamic Storage Solutions To Transform The Data Center, September 2014. Available from http://www.hgst.com/press-room/. [Cited on page 9.]

[84] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and M. West. Scale and Performance in a Distributed File System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, pages 1–2, New York, NY, USA, 1987. ACM. ISBN 0-89791-242-X. doi: 10.1145/41457.37500. URL http://doi.acm.org/10.1145/41457.37500. [Cited on pages 2, 4, and 59.]

[85] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 879–891, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/hu. [Cited on pages 60, 63, and 77.]

[86] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, Boston, MA, 2012. USENIX. ISBN 978-931971-93-5. URL https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang. [Cited on page 71.]

[87] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The XtreemFS Architecture – a Case for Object-based File Systems in Grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, December 2008. ISSN 1532-0626. doi: 10.1002/cpe.v20:17. URL http://dx.doi.org/10.1002/cpe.v20:17. [Cited on pages 2, 4, 59, and 60.]

[88] IBM. IBM 350 disk storage unit. https://www.ibm.com/ibm/history/exhibits/storage/storage_350.html, 2020. [Cited on pages 3 and 9.]

[89] Shuichi Ihara and Shilong Wang. Lustre/ldiskfs Metadata Performance Boost. https://www.eofs.eu/_media/events/lad17/19_shuichi_ihara_lad17_ihara_1004.pdf, October 2017. [Cited on page 4.]

[90] Facebook Inc. RocksDB Direct IO. https://github.com/facebook/rocksdb/wiki/Direct-IO, 2019. [Cited on page 76.]

[91] Facebook Inc. RocksDB Merge Operator. https://github.com/facebook/rocksdb/wiki/Merge-Operator, 2019. [Cited on pages 70 and 97.]

[92] Facebook Inc. RocksDB Synchronous Writes. https://github.com/facebook/rocksdb/wiki/Basic-Operations#synchronous-writes, 2019. [Cited on page 65.]

[93] Silicon Graphics Inc. XFS Allocation Groups. http://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure/tmp/en-US/html/Allocation_Groups.html, 2006. [Cited on page 66.]

[94] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). Draft Standard T10/BSR INCITS 536, American National Standards Institute, Inc., September 2014. Available from https://www.t10.org/ftp/zbcr01.pdf. [Cited on pages 11, 79, and 81.]

[95] INCITS T13 Technical Committee. Information technology - Zoned Device ATA Command Set (ZAC). Draft Standard T13/BSR INCITS 537, American National Standards Institute, Inc., December 2015. Available from http://www.t13.org/Documents/UploadedDocuments/docs2015/di537r05-Zoned_Device_ATA_Command_Set_ZAC.pdf. [Cited on pages 79 and 81.]

[96] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: Write-optimization in a kernel file system. *Trans. Storage*, 11(4):18:1–18:29, November 2015. ISSN 1553-3077. doi: 10.1145/2798729. URL http://doi.acm.org/10.1145/2798729. [Cited on page 77.]

[97] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 309–320, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-01-0. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/jeong. [Cited on page 65.]

[98] Chao Jin, Wei-Ya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. HiSMRfs: a High Performance File System for Shingled Storage Array. In *Proceedings of the 2014 IEEE 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, June 2014. doi: 10.1109/MSST.2014.6855539. [Cited on pages 35 and 39.]

[99] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. ISBN 1-55860-153-8. URL http://dl.acm.org/citation.cfm?id=645920.672996. [Cited on page 70.]

[100] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 52–65, New York, NY, USA, 1997. ACM. ISBN 0-89791-916-5. doi: 10.1145/268998.266644. URL

http://doi.acm.org/10.1145/268998.266644. [Cited on page 59.]

[101] Jurgen Kaiser, Dirk Meister, Tim Hartung, and Andre Brinkmann. ESB: Ext2 Split Block Device. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS '12, pages 181–188, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4903-3. doi: 10.1109/ICPADS.2012.34. URL http://dx.doi.org/10.1109/ICPADS.2012.34. [Cited on page 56.]

[102] Jeffrey Katcher. Postmark: A New File System Benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997. [Cited on page 49.]

[103] John Kennedy and Michael Satran. About Transactional NTFS. https://docs.microsoft.com/en-us/windows/desktop/fileio/about-transactional-ntfs, May 2018. [Cited on page 77.]

[104] John Kennedy and Michael Satran. Alternatives to using Transactional NTFS. https://docs.microsoft.com/en-us/windows/desktop/fileio/deprecation-of-txf, May 2018. [Cited on pages 63 and 77.]

[105] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 183–189, Santa Clara, CA, 2015. USENIX Association. ISBN 978-1-931971-201. URL https://www.usenix.org/conference/fast15/technical-sessions/presentation/kim_jaeho. [Cited on pages 3 and 80.]

[106] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *Consumer Electronics, IEEE Transactions on*, 48(2):366–375, May 2002. ISSN 0098-3063. doi: 10.1109/TCE.2002.1010143. [Cited on page 32.]

[107] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux Implementation of a Log-structured File System. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006. ISSN 0163-5980. doi: 10.1145/1151374.1151375. URL http://doi.acm.org/10.1145/1151374.1151375. [Cited on page 39.]

[108] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. Disks Are Like Snowflakes: No Two Are Alike. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 14–14, Berkeley, CA, USA, 2011. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1991596.1991615. [Cited on page 36.]

[109] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dussea. Parity lost and parity regained. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 9:1–9:15, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1364813.1364822. [Cited on page 31.]

[110] Mark Kryder, E.C. Gage, T.W. McDaniel, W.A Challener, R.E. Rottmayer, Ganping Ju, Yiao-Tee Hsia, and M.F. Erden. Heat Assisted Magnetic Recording. *Proceedings of the IEEE*, 96 (11):1810–1835, November 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.2004315. [Cited on page 9.]

[111] Aneesh Kumar KV, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, volume 1, 2008. [Cited on pages 41 and 42.]

[112] Butler Lampson and Howard E Sturgis. Crash recovery in a distributed data storage system. 1979. [Cited on page 60.]

[113] Rob Landley. Red-black Trees (rbtree) in Linux. https://www.kernel.org/doc/Documentation/rbtree.txt, January 2007. [Cited on page 44.]

[114] Quoc M. Le, Kumar SathyanarayanaRaju, Ahmed Amer, and JoAnne Holliday. Workload Impact on Shingled Write Disks: All-Writes Can Be Alright. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 444–446, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4430-4. doi: 10.1109/MASCOTS.2011.58. URL http://dx.doi.org/10.1109/MASCOTS.2011.58. [Cited on page 35.]

[115] D. Le Moal, Z. Bandic, and C. Guyot. Shingled file system host-side management of Shingled Magnetic Recording disks. In *Proceedings of the 2012 IEEE International Conference on Consumer Electronics (ICCE)*, pages 425–426, January 2012. doi: 10.1109/ICCE.2012.6161799. [Cited on pages 9, 11, 35, and 39.]

[116] Adam Leventhal. APFS in Detail: Overview. http://dtrace.org/blogs/ahl/2016/06/19/apfs-part1/, 2016. [Cited on pages 4 and 68.]

[117] Libata FAQ. https://ata.wiki.kernel.org/index.php/Libata_FAQ, 2011. [Cited on page 17.]

[118] Chung-I Lin, Dongchul Park, Weiping He, and David H. C. Du. H-SWD: Incorporating Hot Data Identification into Shingled Write Disks. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 321–330, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4793-0. doi: 10.1109/MASCOTS.2012.44. URL http://dx.doi.org/10.1109/MASCOTS.2012.44. [Cited on page 35.]

[119] Chen Luo and Michael J. Carey. LSM-based Storage Techniques: A Survey. *CoRR*, abs/1812.07527, 2018. URL http://arxiv.org/abs/1812.07527. [Cited on page 88.]

[120] Colm MacCárthaigh. Scaling Apache 2.x beyond 20,000 concurrent downloads. In *ApacheCon EU*, July 2005. [Cited on page 49.]

[121] Peter Macko, Xiongzi Ge, John Haskins Jr., James Kelley, David Slik, Keith A. Smith, and Maxim G. Smith. SMORE: A Cold Data Object Store for SMR Drives (Extended Version). *CoRR*, abs/1705.09701, 2017. URL http://arxiv.org/abs/1705.09701. [Cited on page 79.]

[122] Magic Pocket & Hardware Engineering Teams. Extending Magic Pocket Innovation with the first petabyte scale SMR drive deployment. https://blogs.dropbox.com/tech/2018/06/extending-magic-pocket-innovation-with-the-first-petabyte-scale-smr-drive-deployment/, 2018. [Cited on page 79.]

[123] Magic Pocket & Hardware Engineering Teams. SMR: What we learned in our first

year. https://blogs.dropbox.com/tech/2019/07/smr-what-we-learned-in-our-first-year/, 2019. [Cited on page 79.]

[124] Adam Manzanares, Noah Watkins, Cyril Guyot, Damien LeMoal, Carlos Maltzahn, and Zvonimr Bandic. ZEA, A Data Management Approach for SMR. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, USA, June 2016. USENIX Association. URL https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/manzanares. [Cited on pages 39 and 84.]

[125] Lars Marowsky-Brée. Ceph User Survey 2018 results. https://ceph.com/ceph-blog/ceph-user-survey-2018-results/, 2018. [Cited on page 60.]

[126] Chris Mason. Compilebench. https://oss.oracle.com/~mason/compilebench/, 2007. [Cited on page 43.]

[127] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, 2007. [Cited on page 41.]

[128] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984. [Cited on pages 41, 56, 66, and 77.]

[129] Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2014. [Cited on page 55.]

[130] Chris Mellor. Toshiba embraces shingling for next-gen MAMR HDDs. https://blocksandfiles.com/2019/03/11/toshiba-mamr-statements-have-shingling-absence/, 2019. [Cited on page 3.]

[131] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 221–234, Santa Clara, CA, 2015. USENIX Association. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-session/presentation/min. [Cited on pages 60 and 63.]

[132] Damien Le Moal. blk-mq support for ZBC disks. https://lwn.net/Articles/742159/, 2017. [Cited on page 84.]

[133] Keith Muller and Joseph Pasquale. A High Performance Multi-structured File System Design. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 56–67, New York, NY, USA, 1991. ACM. ISBN 0-89791-447-3. doi: 10.1145/121132.286600. URL http://doi.acm.org/10.1145/121132.286600. [Cited on page 56.]

[134] Sumedh N. Coding for Performance: Data alignment and structures. https://software.intel.com/en-us/articles/coding-for-performance-data-alignment-and-structures, 2013. [Cited on page 76.]

[135] NetBSD-Wiki. How to install a server with a root LFS partition. https://wiki.netbsd.

org/tutorials/how_to_install_a_server_with_a_root_lfs_partition/, 2012. [Cited on page 39.]

[136] Juan Carlos Olamendy. An LSM-tree engine for MySQL. https://blog.toadworld.com/2017/11/15/an-lsm-tree-engine-for-mysql, 2017. [Cited on page 88.]

[137] Michael A. Olson. The Design and Implementation of the Inversion File System. In *USENIX Winter*, Berkeley, CA, USA, 1993. USENIX Association. [Cited on pages 60, 63, and 77.]

[138] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1268708.1268751. [Cited on page 77.]

[139] OpenStack Foundation. 2017 Annual Report. https://www.openstack.org/assets/reports/OpenStack-AnnualReport2017.pdf, 2017. [Cited on page 66.]

[140] Adrian Palmer. SMRFFS-EXT4—SMR Friendly File System. https://github.com/Seagate/SMR_FS-EXT4, 2015. [Cited on pages 5 and 39.]

[141] Adrian Palmer. SMR in Linux Systems. In *Vault Linux Storage and File System Conference*, Boston, MA, USA, April 2016. [Cited on page 39.]

[142] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):38:1–38:23, August 2008. ISSN 1539-9087. doi: 10.1145/1376804.1376806. URL http://doi.acm.org/10.1145/1376804.1376806. [Cited on page 32.]

[143] Swapnil Patil and Garth Gibson. Scale and Concurrency of GIGA+: File System Directories with Millions of Files. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association. ISBN 978-1-931971-82-9. URL http://dl.acm.org/citation.cfm?id=1960475.1960488. [Cited on pages 1 and 65.]

[144] PerlMonks. Fastest way to recurse through VERY LARGE directory tree. http://www.perlmonks.org/?node_id=883444, 2011. [Cited on page 49.]

[145] Juan Piernas, Toni Cortes, and José M. García. Dualfs: A new journaling file system without meta-data duplication. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, page 137âĂŞ146, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134835. doi: 10.1145/514191.514213. URL https://doi.org/10.1145/514191.514213. [Cited on pages 49, 50, 56, and 77.]

[146] S. N. Piramanayagam. Perpendicular recording media for hard disk drives. *Journal of Applied Physics*, 102(1):011301, July 2007. ISSN 0021-8979, 1089-7550. doi: 10.1063/1.2750414. [Cited on page 9.]

[147] Rekha Pitchumani, Andy Hospodor, Ahmed Amer, Yangwook Kang, Ethan L. Miller, and Darrell D. E. Long. Emulating a Shingled Write Disk. In *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 339–346, Washington, DC, USA, 2012. IEEE

Computer Society. ISBN 978-0-7695-4793-0. doi: 10.1109/MASCOTS.2012.46. URL http://dx.doi.org/10.1109/MASCOTS.2012.46. [Cited on pages 13 and 35.]

[148] Rekha Pitchumani, James Hughes, and Ethan L. Miller. SMRDB: Key-value Data Store for Shingled Magnetic Recording Disks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 18:1–18:11, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3607-9. doi: 10.1145/2757667.2757680. URL http://doi.acm.org/10.1145/2757667.2757680. [Cited on page 83.]

[149] Poornima G and Rajesh Joseph. Metadata Performance Bottlenecks in Gluster. https://www.slideshare.net/GlusterCommunity/performance-bottlenecks-for-metadata-workload-in-gluster-with-poornima-gurusiddaiah-rajesh-joseph, 2016. [Cited on pages 1, 4, and 65.]

[150] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating System Transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 161–176, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629591. URL http://doi.acm.org/10.1145/1629575.1629591. [Cited on pages 60, 63, and 77.]

[151] Sundar Poudyal. Partial write system, March 13 2013. US Patent App. 13/799,827. [Cited on page 22.]

[152] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, USA, April 2005. [Cited on pages 40 and 41.]

[153] Lee Prewitt. SMR and ZNS – Two Sides of the Same Coin. https://www.youtube.com/watch?v=jBxzO6YyMxU, 2019. [Cited on page 79.]

[154] Red Hat Inc. GlusterFS Architecture. https://docs.gluster.org/en/latest/Quick-Start-Guide/Architecture/, 2019. [Cited on pages 2, 4, 59, and 60.]

[155] David Reinsel, John Gantz, and John Rydning. Data Age 2025: The Evolution of Data to Life-Critical. https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf, 2017. [Cited on pages 1 and 105.]

[156] Kai Ren and Garth Gibson. TABLEFS: Enhancing Metadata Efficiency in the Local File System. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, San Jose, CA, USA, 2013. USENIX. ISBN 978-1-931971-01-0. URL https://www.usenix.org/conference/atc13/technical-sessions/presentation/ren. [Cited on pages 49, 56, and 77.]

[157] Drew Riley. Samsung's SSD Global Summit: Samsung: Flexing Its Dominance In The NAND Market, August 2013. URL http://www.tomshardware.com/reviews/samsung-global-ssd-summit-2013,3570.html. [Cited on page 9.]

[158] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. ACM. ISBN 0-89791-

447-3. doi: 10.1145/121132.121137. URL http://doi.acm.org/10.1145/121132.121137. [Cited on pages 11, 39, 56, 83, and 98.]

[159] Ricardo Santana, Raju Rangaswami, Vasily Tarasov, and Dean Hildebrand. A Fast and Slippery Slope for File Systems. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3945-2. doi: 10.1145/2819001.2819003. URL http://doi.acm.org/10.1145/2819001.2819003. [Cited on page 39.]

[160] SATA-IO. Serial ATA Revision 3.1 Specification. Technical report, SATA-IO, July 2011. [Cited on page 12.]

[161] Steven W. Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On Multidimensional Data and Modern Disks. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251028.1251045. [Cited on page 36.]

[162] Frank Schmuck and Jim Wylie. Experience with Transactions in QuickSilver. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 239–253, New York, NY, USA, 1991. ACM. ISBN 0-89791-447-3. doi: 10.1145/121132.121171. URL http://doi.acm.org/10.1145/121132.121171. [Cited on pages 60, 63, and 77.]

[163] Thomas J. E. Schwarz, Qin Xin, Ethan L. Miller, Darrell D. E. Long, Andy Hospodor, and Spencer Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '04, pages 409–418, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2251-3. URL http://dl.acm.org/citation.cfm?id=1032659.1034226. [Cited on page 65.]

[164] Seagate. Seagate Technology PLC Fiscal Fourth Quarter and Year End 2013 Financial Results Supplemental Commentary, July 2013. Available from http://www.seagate.com/investors. [Cited on page 9.]

[165] Seagate. Seagate Ships World's First 8TB Hard Drives, August 2014. Available from http://www.seagate.com/about/newsroom/. [Cited on page 9.]

[166] Seagate Technology LLC. Seagate Desktop HDD: ST5000DM000, ST4000DM001. Product Manual 100743772, Seagate Technology LLC, December 2013. [Cited on page 9.]

[167] Seagate Technology PLC. Terascale HDD. Data sheet DS1793.1-1306US, Seagate Technology PLC, June 2013. [Cited on page 9.]

[168] Seastar. Shared-nothing Design. http://seastar.io/shared-nothing/, August 2019. [Cited on page 76.]

[169] Margo Seltzer, Keith Bostic, Marshall Kirk Mckusick, and Carl Staelin. An Implementation of a Log-structured File System for UNIX. In *Proceedings of the USENIX Winter 1993 Conference*, USENIX'93, pages 3–3, Berkeley, CA, USA, 1993. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267303.1267306. [Cited on page 39.]

[170] Margo I. Seltzer. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 503–510, Washington, DC, USA, 1993. IEEE Computer Society. ISBN 0-8186-3570-3. URL http://dl.acm.org/citation.cfm?id=645478.654970. [Cited on pages 60 and 63.]

[171] ServerFault. Doing an rm -rf on a massive directory tree takes hours. http://serverfault.com/questions/46852, 2009. [Cited on page 49.]

[172] Mansour Shafaei, Mohammad Hossein Hajkazemi, Peter Desnoyers, and Abutalib Aghayev. Modeling SMR Drive Performance. *SIGMETRICS Perform. Eval. Rev.*, 44(1):389âĂŞ390, June 2016. ISSN 0163-5999. doi: 10.1145/2964791.2901496. URL https://doi.org/10.1145/2964791.2901496. [Cited on page 36.]

[173] Mansour Shafaei, Mohammad Hossein Hajkazemi, Peter Desnoyers, and Abutalib Aghayev. Modeling Drive-Managed SMR Performance. *ACM Trans. Storage*, 13(4), December 2017. ISSN 1553-3077. doi: 10.1145/3139242. URL https://doi.org/10.1145/3139242. [Cited on page 36.]

[174] Kai Shen, Stan Park, and Men Zhu. Journaling of Journal Is (Almost) Free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 287–293, Santa Clara, CA, 2014. USENIX. ISBN ISBN 978-1-931971-08-9. URL https://www.usenix.org/conference/fast14/technical-sessions/presentation/shen. [Cited on page 65.]

[175] A. Shilov. Western Digital: Over Half of Data Center HDDs Will Use SMR by 2023. https://www.anandtech.com/show/14099/western-digital-over-half-of-dc-hdds-will-use-smr-by-2023, 2019. [Cited on pages 3 and 79.]

[176] Anton Shilov. Seagate Ships 35th Millionth SMR HDD, Confirms HAMR-Based Drives in Late 2018. https://www.anandtech.com/show/11315/seagate-ships-35th-millionth-smr-hdd-confirms-hamrbased-hard-drives-in-late-2018, 2017. [Cited on page 3.]

[177] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. doi: 10.1109/MSST.2010.5496972. URL http://dx.doi.org/10.1109/MSST.2010.5496972. [Cited on pages 2, 4, and 59.]

[178] Chris Siebenmann. About the order that readdir() returns entries in. https://utcc.utoronto.ca/~cks/space/blog/unix/ReaddirOrder, 2011. [Cited on page 65.]

[179] Chris Siebenmann. ZFS transaction groups and the ZFS Intent Log. https://utcc.utoronto.ca/~cks/space/blog/solaris/ZFSTXGsAndZILs, 2013. [Cited on page 63.]

[180] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 29–42, USA, 2009. USENIX Association. [Cited on pages 60, 63, and 77.]

[181] Stas Starikevich. [ceph-users] RadosGW performance degradation on the 18 millions

objects stored. http://lists.ceph.com/pipermail/ceph-users-ceph.com/2016-September/012983.html, 2016. [Cited on page 66.]

[182] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix Hupfeld. BabuDB: Fast and Efficient File System Metadata Storage. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, SNAPI '10, pages 51–58, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4025-2. doi: 10.1109/SNAPI.2010.14. URL http://dx.doi.org/10.1109/SNAPI.2010.14. [Cited on page 60.]

[183] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981. ISSN 0001-0782. doi: 10.1145/358699.358703. URL http://doi.acm.org/10.1145/358699.358703. [Cited on page 59.]

[184] Michael Stonebraker and Lawrence A. Rowe. The Design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 340–355, New York, NY, USA, 1986. ACM. ISBN 0-89791-191-1. doi: 10.1145/16894.16888. URL http://doi.acm.org/10.1145/16894.16888. [Cited on page 77.]

[185] Miklos Szeredi et al. FUSE: Filesystem in userspace. https://github.com/libfuse/libfuse/, 2020. [Cited on page 56.]

[186] Nisha Talagala, Remzi H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report UCB/CSD-99-1063, EECS Department, University of California, Berkeley, 1999. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/1999/6275.html. [Cited on pages 10 and 36.]

[187] S. Tan, W. Xi, Z.Y. Ching, C. Jin, and C.T. Lim. Simulation for a Shingled Magnetic Recording Disk. *IEEE Transactions on Magnetics*, 49(6):2677–2681, June 2013. ISSN 0018-9464. doi: 10.1109/TMAG.2013.2245872. [Cited on page 35.]

[188] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX ;login issue*, 41(1), 2016. [Cited on page 45.]

[189] ZAR team. "Write hole" phenomenon. http://www.raid-recovery-guide.com/raid5-write-hole.aspx, 2019. [Cited on page 71.]

[190] ThinkParQ. An introduction to BeeGFS. https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2018. [Cited on pages 2, 4, and 59.]

[191] D.A Thompson and J.S. Best. The future of magnetic data storage techology. *IBM Journal of Research and Development*, 44(3):311–322, May 2000. ISSN 0018-8646. doi: 10.1147/rd.443.0311. [Cited on pages 9 and 10.]

[192] Linus Torvalds and Peter Zijlstra. __wb_calc_thresh. http://lxr.free-electrons.com/source/mm/page-writeback.c?v=4.6#L733, 2016. [Cited on page 45.]

[193] Theodore Ts'o. Release of e2fsprogs 1.43.2. http://www.spinics.net/lists/linux-ext4/msg53544.html, September 2016. [Cited on page 41.]

[194] Stephen C Tweedie. Journaling the Linux ext2fs Filesystem. In *The Fourth Annual Linux Expo*, Durham, NC, USA, May 1998. [Cited on pages 42, 46, and 63.]

[195] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-tree Based Key-value Store on

Open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 16:1–16:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592804. URL http://doi.acm.org/10.1145/2592798.2592804. [Cited on page 88.]

[196] Sumei Wang, Yao Wang, and R.H. Victora. Shingled Magnetic Recording on Bit Patterned Media at 10 Tb/in^2. *IEEE Transactions on Magnetics*, 49(7):3644–3647, July 2013. ISSN 0018-9464. doi: 10.1109/TMAG.2012.2237545. [Cited on page 9.]

[197] WDC. My Passport Ultra. https://www.wdc.com/products/portable-storage/my-passport-ultra-new.html, July 2016. [Cited on page 55.]

[198] Sage Weil. [RFC] big fat transaction ioctl. https://lwn.net/Articles/361439/, 2009. [Cited on page 63.]

[199] Sage Weil. Re: [RFC] big fat transaction ioctl. https://lwn.net/Articles/361472/, 2009. [Cited on page 63.]

[200] Sage Weil. [PATCH v3] introduce sys_syncfs to sync a single file system. https://lwn.net/Articles/433384/, March 2011. [Cited on page 64.]

[201] Sage Weil. Goodbye XFS: Building a New, Faster Storage Backend for Ceph. https://www.snia.org/sites/default/files/SDC/2017/presentations/General_Session/Weil_Sage%20_Red_Hat_Goodbye_XFS_Building_a_new_faster_storage_backend_for_Ceph.pdf, 2017. [Cited on page 1.]

[202] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298485. [Cited on pages 2, 4, 59, and 60.]

[203] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 122–es, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 0769527000. doi: 10.1145/1188455.1188582. URL https://doi.org/10.1145/1188455.1188582. [Cited on page 61.]

[204] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, pages 35–44, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-899-2. doi: 10.1145/1374596.1374606. URL http://doi.acm.org/10.1145/1374596.1374606. [Cited on page 60.]

[205] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 2:1–2:17, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1364813.1364815. [Cited on pages 2, 4, 56, 59, and 60.]

[206] Western Digital—Jorge Campello De Souza. What is Zoned Storage and the Zoned Storage Initiative? https://blog.westerndigital.com/what-is-zoned-storage-initiative/, 2019. [Cited on pages 1, 3, 79, and 93.]

[207] Western Digital Inc. ZBC device manipulation library. https://github.com/hgst/libzbc, 2018. [Cited on page 84.]

[208] Western Digital Inc. Zoned block device manipulation library. https://github.com/westerndigitalcorporation/libzbd, 2020. [Cited on page 80.]

[209] Lustre Wiki. Introduction to Lustre Architecture. http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf, 2017. [Cited on pages 2, 4, 56, 59, and 60.]

[210] Wikipedia. Btrfs History. https://en.wikipedia.org/wiki/Btrfs#History, 2018. [Cited on pages 4 and 68.]

[211] Wikipedia. XFS History. https://en.wikipedia.org/wiki/XFS#History, 2018. [Cited on pages 4 and 68.]

[212] Wikipedia. COVID-19 pandemic. https://en.wikipedia.org/wiki/COVID-19_pandemic, 2019. [Cited on page 98.]

[213] Wikipedia. Cache flushing. https://en.wikipedia.org/wiki/Disk_buffer#Cache_flushing, 2019. [Cited on pages 12 and 65.]

[214] Wikipedia. Parallel ATA. https://en.wikipedia.org/wiki/Parallel_ATA#IDE_and_ATA-1, 2020. [Cited on page 3.]

[215] R. Wood, Mason Williams, A Kavcic, and Jim Miles. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *IEEE Transactions on Magnetics*, 45 (2):917–923, February 2009. ISSN 0018-9464. doi: 10.1109/TMAG.2008.2010676. [Cited on pages 3, 9, and 11.]

[216] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '95/PERFORMANCE '95, pages 146–156, New York, NY, USA, 1995. ACM. doi: 10.1145/223587.223604. [Cited on pages 10 and 36.]

[217] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *ACM Trans. Storage*, 3(2):4–es, June 2007. ISSN 1553-3077. doi: 10.1145/1242520.1242521. URL https://doi.org/10.1145/1242520.1242521. [Cited on pages 60, 63, and 77.]

[218] F. Wu, Z. Fan, M. Yang, B. Zhang, X. Ge, and D. H. C. Du. Performance Evaluation of Host Aware Shingled Magnetic Recording (HA-SMR) Drives. *IEEE Transactions on Computers*, 66(11):1932–1945, 2017. [Cited on page 36.]

[219] Fenggang Wu, Ming-Chang Yang, Ziqi Fan, Baoquan Zhang, Xiongzi Ge, and David H. C. Du. Evaluating host aware smr drives. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, page 31âĂŞ35, USA, 2016. USENIX Association. [Cited on page 36.]

[220] Fengguang Wu. I/O-less Dirty Throttling. https://events.linuxfoundation.org/

images/stories/pdf/lcjp2012_wu.pdf, June 2012. [Cited on page 67.]

[221] 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 *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 15–28, Santa Clara, CA, 2017. USENIX Association. ISBN 978-1-931971-36-2. URL https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan. [Cited on pages 1, 5, and 80.]

[222] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: A gc-free key-value store on hm-smr drives with gear compaction. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 159–171, Boston, MA, February 2019. USENIX Association. ISBN 978-1-939133-09-0. URL https://www.usenix.org/conference/fast19/presentation/yao. [Cited on pages 83 and 84.]

[223] 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*, FASTâĂŹ12, page 1, USA, 2012. USENIX Association. [Cited on page 96.]

[224] Zhihui Zhang and Kanad Ghose. hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 175–187, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273016. URL http://doi.acm.org/10.1145/1272996.1273016. [Cited on pages 49, 56, and 77.]

[225] Qing Zheng, Charles D. Cranor, Danhao Guo, Gregory R. Ganger, George Amvrosiadis, Garth A. Gibson, Bradley W. Settlemyer, Gary Grider, and Fan Guo. Scaling Embedded In-situ Indexing with deltaFS. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 3:1–3:15, Piscataway, NJ, USA, 2018. IEEE Press. URL http://dl.acm.org/citation.cfm?id=3291656.3291660. [Cited on page 77.]

[226] Alexey Zhuravlev. ZFS: Metadata Performance. https://www.eofs.eu/_media/events/lad16/02_zfs_md_performance_improvements_zhuravlev.pdf, 2016. [Cited on pages 60 and 65.]

[227] Peter Zijlstra. sysfs-class-bdi. https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-class-bdi, January 2008. [Cited on page 45.]