UNIVERSITY OF CALIFORNIA
SANTA CRUZ

ENHANCING FLASH STORAGE PERFORMANCE AND LIFETIME WITH HOST-GUIDED DATA PLACEMENT

A dissertation submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Devashish R. Purandare

June 2024

The Dissertation of Devashish R. Purandare is approved:

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

API	Application Programming Interface
CXL	Compute eXpress Link
DBSCAN	Density Based Spatial Clustering of Applications with Noise
SSD	Solid State Drive
eBPF	extended Berkeley Packet Filter

ACRONYMS

F2FS	Flash-First FileSystem
FDP	Flexible Data Placement
FTL	Flash Translation Layer
LBA	Logical Block Address
LSM	Log-Structured Merge tree
MAT	Metadata Address Table
MLC	Multi Level Cell
NAT	Node Address Table
NVMe	Non-Volatile Memory express
POSIX	Portable Operating System Interface
QLC	Quad Level Cell
SIT	Segment Information Table
SLC	Single Level Cell
SSA	Segment Summary Area
SSD	Solid State Drive
SST	Sorted String Table
TLC	Three Level Cell
VFS	Virtual FileSystem
WAL	Write-Ahead Log
WASM	Web Assembly
ZB	Zone Bitmap
ZIT	Zone Information Table
ZNS	Zoned Namespace SSDs

ABSTRACT

ENHANCING FLASH STORAGE PERFORMANCE AND LIFETIME WITH HOST-GUIDED DATA PLACEMENT

Devashish R. Purandare

Solid State Drives (SSDs) form the core of modern data center storage, propelled by rapid growth in capacity and improved performance and reliability over hard drives. However, with shrinking feature sizes, capacity gains come at the cost of performance degradation and reduced lifetime. NAND flash's inability to support in-place updates further degrades performance and lifetime due to the overhead of remapping and garbage collection. Even in SSD-optimized log-structured systems, the lack of coordination can result in misaligned and interleaved logs across the storage stack, requiring garbage collection at multiple levels, increasing wear due to write amplification. Efforts to address these issues require rethinking the SSD interface, leveraging host insights to co-locate related data. Such co-location can improve write isolation across unrelated data streams and reduce write amplification by lowering data movement caused by garbage collection.

However, host-device coordination interfaces have struggled with large-scale adoption. The complexity introduced by integrating interface changes in applications and filesystems, coupled with the difficulty of grouping related data, makes adoption expensive and non-portable. Further, useful placement directives are hard to generate as applications are storage-unaware and filesystems application-unaware. Abstraction breaking changes can increase security risks and limit generalizability. Unless we tackle the complexity of abstraction changes and placement planning head-on, embracing these new interfaces will be limited to proof-of-concept applications or proprietary hyperscalar silos.

In this dissertation, we identify and address the pressing issues that limit host-device coordination: (i) Enabling host-device coordination without large-scale application or filesystem rewrites, (ii) Generating intelligent placement directives based on the application and workload, and (iii) Simplified data placement and storage management for current and future SSD interfaces. We present systems that can coordinate or work independently to support modern interfaces: (i) Shimmer allows applications to support requirements of changing interfaces without modifying the application or the storage system. (ii) Parakeet presents a rich API that can capture data relations and lifetimes and generate heuristic-based or dynamic data placement plans. (iii) Persimmon and *shimmer-vfs* can manage the placement and maintenance of flash storage and utilize the placement indicators from Parakeet. Our approach enables embracing host-device coordination without requiring changes to data management systems.

Together, these systems eliminate in-place updates and present a highly optimized data path for flash without kernel bypass or application-specific approaches. We demonstrate a 30-90% improvement in write throughput, a reduction in read and write latency, with a factor of 14 reduction in tail latency, and a large reduction in garbage collection overhead over traditional approaches. Further, we demonstrate that shim layers can keep up with highly-tuned per-application approaches, reducing application and filesystem complexity, reducing CPU, memory, and space utilization, and enabling more applications to unlock the full performance of SSDs.

For

Sampada,

Ravindra,

and Rucha.

PUBLICATIONS

[1] Devashish Purandare, Pete Wilcox, Heiner Litz, and Shel Finkelstein. "Append is Near: Log-based Data Management on ZNS SSDs." In: *Conference on Innovative Data Systems Research 2022 (CIDR '22)*. 2022. URL: https://www.cidrdb.org/cidr2022/papers/p93-purandare.pdf.

[2] Devashish R. Purandare, Daniel Bittman, and Ethan L. Miller. "Analysis and Workload Characterization of the CERN EOS Storage System." In: *SIGOPS Oper. Syst. Rev.* 56.1 (June 2022), pp. 55–61. ISSN: 0163-5980. DOI: 10.1145/3544497.3544507. URL: https://doi.org/10.1145/3544497.3544507.

[3] Devashish R. Purandare, Daniel Bittman, and Ethan L. Miller. "Analysis and workload characterization of the CERN EOS storage system." In: *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. CHEOPS '22. Rennes, France: Association for Computing Machinery, 2022, pp. 1–7. ISBN: 9781450392099. DOI: 10.1145/3503646.3524293. URL: https://doi.org/10.1145/3503646.3524293.

[4] Devashish R. Purandare, Sam Schmidt, and Ethan L. Miller. "Persimmon: an append-only ZNS-first filesystem." In: *2023 IEEE 41st International Conference on Computer Design (ICCD)*. 2023, pp. 308–315. DOI: 10.1109/ICCD58817.2023.00054.

ACKNOWLEDGMENTS

My research journey would not have been possible without the help, advice, and guidance from friends and colleagues. My advisor, Ethan L. Miller, has provided invaluable insights throughout this work, from framing to data analysis, and this work would not have been possible without him. My committee members, Darrell D. E. Long and Peter Alvaro, played a pivotal role in my research, and their ideas, feedback, critique, and support was essential for the successful completion of this work. They went above and beyond to support my career, and I am deeply grateful. I want to highlight my co-author Sam Schmidt, whose contributions to Persimmon played an essential role in the implementation.

I want to appreciate the feedback and support from Daniel Bittman, without whom this dissertation would not have been possible, and Achilles Benetopoulos for his input on the work. Several mentors, co-authors, and colleagues helped me shape this research, in no particular order: Eugene Chou, Shel Finkelstein, Avani Wildani, Pat Helland, Adrian Cockraft, Heiner Litz, Pete Wilcox, and other CRSS grad students and faculty.

The steadfast support of my parents (Sampada and Ravindra Purandare), my sister (Rucha Gangal), and friends has been equally important in this research journey; I am grateful to be given this opportunity and surrounded by such wonderful people. This research has been possible due to the support from the Baskin School of Engineering, the industry sponsors for the Center for Research in Systems and Storage, and the National Science Foundation. I am grateful to Matias Bjørling and Western Digital for donating ZN540 SSDs.

The author battled LaTeX for typesetting, and was helped by Daniel Bittman's template using an unabashedly strong influence from Aaron Turon's dissertation[1] as inspiration for formatting. The document uses a modified `classicthesis` by André Miede.

1 http://aturon.github.io/academic/turon-thesis.pdf

FOREWORD

“To produce a mighty book, you must choose a mighty theme. No great and enduring volume can ever be written on the flea, though many there be who have tried it.”

—Herman Melville

This dissertation combines work done over several years in grad school and has changed considerably from the proposed work. While its narrative puts the work in order, I want to warn readers that wisdom is with hindsight. The narrative was put together while writing this dissertation, and unlike in this dissertation, the work has not been chronological or planned to be interoperable. Designing around interface boundaries enabled us to achieve such systems. This acknowledgment is not to say that the narrative is untrue. Still, that research is non-linear, and the process periodically requires you to reflect, learn, and reorganize findings to see the big picture.

The design of Shimmer stems from lessons from earlier work and failed approaches. For instance, I realized the optimal optimal target after several failed attempts at kernel-level updates or optimized applications. The research journey is long, arduous, lonely, and full of dead ends. To any graduate student reading this work, I encourage you to withdraw and reflect when things get overwhelming. I have had to scratch months of work leading to dead ends, but only in hindsight do you see that the setbacks shape something better. Just have a little faith.

Devashish Purandare

April 15 2024

PART I

PRELUDE

“It’s a poor sort of memory that only works backwards,” the Queen remarked.

—Lewis Carroll • Alice’s Adventures in Wonderland

1
INTRODUCTION

As the transistors became increasingly unpredictable, the foundations of John's world began to crumble. So, John did what any reasonable person would do: he cloaked himself in a wall of denial and acted like nothing had happened. "Making processors faster is increasingly difficult," John thought, "but maybe people won't notice if I give them more processors."

—James Mickens • The Slow Winter

With shrinking feature sizes approaching physical limits, performance and durability gains require rethinking interfaces, architectures, and allowing host-device coordination. Storage abstractions from the twentieth century add overhead on modern hardware and limit expressivity. Yet, the same abstractions ensure portability, interoperability, and security of these systems. So while we need to express primitives not possible with these abstractions, eliminating these abstractions presents tougher challenges. Is there another way?

1.1 Motivation

NAND flash-based storage has become central to modern storage stacks due to its rapid growth in capacity, compact nature, efficiency, performance, and reliability. Yet, sustaining such capacity growth is becoming increasingly difficult without adversely impacting storage performance and lifetime. With traditional scaling techniques like shrinking transistor sizes and increasing cell density close to their physical limit, each generation adversely impacts performance and durability. To make matters worse, NAND flash cannot perform in-place updates and has a large erase granularity, requiring complex and expensive controllers and data structures to simulate a random write interface.

To simulate random writes, these SSDs need excess memory (about 1 GB of RAM for every terabyte of flash) and excess over-provisioned capacity (17–28% of the capacity), driving up the cost of hardware. Further, background operations like garbage collection impact performance and wear out the device even faster. Even in the SSD-optimized log-structured systems, we see a performance degradation due to the lack of alignment between host logs and device logs (the *"log-on-log"* problem), exacerbating write amplification and wear [68].

Flash manufacturers have proposed various interfaces, such as Open Channel SSDs [15] or Zoned Namespace SSDs (ZNS) [13] that offer the host control over on-device data placement to reduce the impact of garbage collection. The goal is simple: grouping data related by lifetime and application can reduce the need for on-device garbage collection and consequently improve the performance and lifetime of the device. Indication from the host about related data can further improve isolation in allocating device buffers across write streams and unlock the device's internal parallelism. However, to achieve this, the host must provide directives about the device's data lifetime.

Enabling coordination requires rewriting systems, often breaking Portable Operating System Interface (POSIX) abstractions that allow application portability and interoperability. Further, despite the availability of such interfaces, application developers have little guidance on how to use them, and differing implementations and changing standards make picking a target difficult.

These interfaces present three main challenges:

① Placement logic integration requires large-scale changes to systems.

② Abstractions make storage layers are unaware of each other.

③ Systems to use placement directives are limited or unavailable.

Unsurprisingly, while there have been several demonstrations of such interfaces, utilization is limited to proof-of-concept applications or is quickly deprecated due to limited adoption. The solutions we see involve either well-tuned approaches limited to a single application like ZenFS [13] or filesystems like F2FS [36] constrained by the need to support legacy devices. The challenge therefore is not limited to introduction of such interfaces, but is about the design around them to enable low-overhead adoption that unlocks most of the benefits.

1.2 Shaping the future of flash storage

To simplify host-device coordination in SSDs, a system will need to provide:

① Placement planning: Generate placement directives per-application

② Plan provisioning: Provide directives to varying interfaces of SSDs

③ Data placement: Utilize directives to co-locate data

In the traditional storage stack, Item 1 would be provided by the application, while Items 2 and 3 would be the responsibility of the filesystem. However, the application must be storage-aware for this approach and provide directives according to its best workload estimate. With Linux's limited support for placement directives (limited to 4 temperature levels across the system), this approach limits the number of data streams that can be isolated. Typically, applications are isolated from other running applications and multiple applications could provide the same hint for data with different characteristics. Plus these hints are optional and inherently lossy, accounting for temperature of writes over lifetime of data. More importantly, this approach requires application rewrites and can limit workload-based placement directives.

Keeping placement directive generation in the filesystem is unlikely to provide any benefits as filesystems are designed to be application-agnostic. Application-specific approaches like ZenFS implement all three responsibilities within the application, which performs well, but such an implementation is expensive in terms of time and developer effort and not generalizable to other applications. We observe that the two traditional solutions to adopting modern interfaces, application changes and modified filesystems, increase complexity and introduce security vulnerabilities while limiting the generalizability and performance of such systems.

In this dissertation, we demonstrate a new approach that isolates added complexity of host-device coordination in a dedicated layer which rests between the filesystem and application. This layer can provide rich, workload-based directives *without modifying the application or the filesystem*. We introduce a modern interface which can provide placement directives across different types of SSDs, and demonstrate that with a low programming overhead, we can match highly tuned approaches and enable easy adoption of modern SSDs, boosting performance and reducing write amplification.

1.3 Implementation

With the append-friendly nature of flash storage, we model our systems around an *append* primitive, eliminating in-place updates and random writes. One of our key design goals is to insulate applications and systems from storage interface changes while unlocking their benefits. We focus on applications that are flash-optimized and use append-only structures, enabling fast and easy adoption with maximum benefits. We leverage their design to enable better optimization, using the data immutability and temporal nature of updates to inform our directives and placement. We tap into heuristics, operator expertise, and machine learning techniques to generate high-quality directives and present a modern interface that provides richer directives than possible with previous approaches. We achieve all this with minimal code changes and no modification to current applications and systems, presenting a streamlined, optimized, flash-first system that works with any application.

In this dissertation we present three interacting systems addressing each of the three issues discussed above:

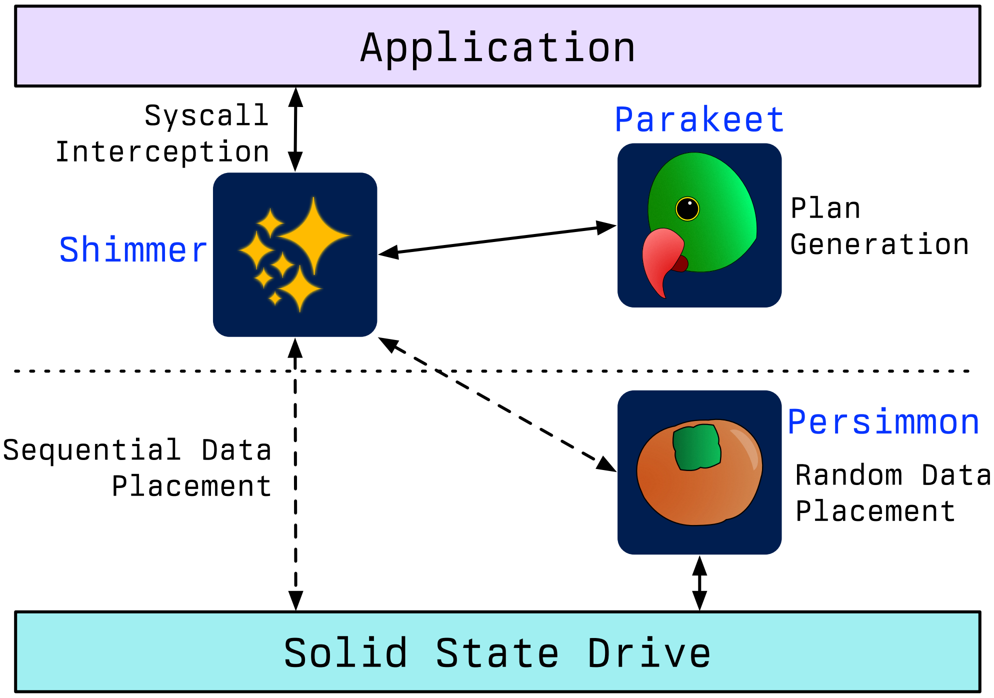

Figure 1.1: Our contributions: Shimmer, Parakeet, and Persimmon address the missing elements of a host-device coordination stack. They enable unmodified applications to get the benefits of modern interfaces to improve performance and device lifetime.

The systems in Fig. 1.1 can work independently, or in tandem, and provide a bridge for adapting applications to modern interfaces. Shimmer can be used with *shimmer-vfs* for direct data placement or as a layer to provide placement directives to a capable filesystem. Persimmon can work as a standalone filesystem, and Parakeet can provide application-specific directives to Shimmer or capable filesystems.

Shimmer can improve the write throughput of IO-intensive applications 30–90% while reducing the tail latency by a factor of 14 over tuned approaches like ZenFS as discussed in Chapter 7. Persimmon is a filesystem centered around appends, reducing tail latency and garbage collection overhead over F2FS, and Parakeet can enable dynamic or heuristic-based directives for any application, across device interfaces.

1.4 Key insights

① **Limitations of Current Approaches**: We demonstrate issues with current systems, like the limited ability of F2FS to utilize both the bandwidth and the capacity of modern SSDs. We show how even append-only systems depend on in-place updates, fixed addresses, and primitives like preallocate and truncate, which add overhead on log-structured devices.

② **Persimmon**: We introduce Persimmon the first filesystem that requires *no in-place updates*— even for metadata, and adapts a design specially tuned for modern SSDs.

③ **Shimmer**: We explore using *shim* layers for modern host-device coordination, allowing us to intercept data movement across the storage stack and influence it without modifying applications or filesystems.

④ **Parakeet**: We explore the limitations of currently available interfaces for provisioning placement hints to storage devices and present a modern interface that considers data temperature and lifetime.

⑤ **Results**: We demonstrate that our design can match application-tuned approaches in performance with lower effort and improve performance 30-90% over flash-optimized filesystems while incurring 14× lower tail latency in a variety of write-intensive applications.

This dissertation provides a blueprint for host-device co-design across abstractions while avoiding large-scale application rewrites. These systems can be scaled from smartphones and personal devices to datacenter scale architectures, allowing the adoption of cheaper, faster, and longer-lasting SSDs.

1.5 Organization

I Prelude

② Chapter 2 looks at the evolution of SSDs and storage interfaces, exploring how we arrived at the place we find ourselves at and the efforts to improve SSD interface over the years and their outcomes.

③ Chapter 3 compares different types of modern SSDs, looks at the state of the art and discusses where it falls short. Then we talk about how our approach differs from these approaches.

II Systems for Modern Flash

④ Chapter 4 looks at the design of special-purpose and POSIX filesystems, comparing their approaches to motivate Persimmon. We explore the design space, implementation, and evaluation of the Persimmon filesystem, comparing it to btrfs and F2FS.

⑤ Chapter 5 presents Shimmer, a dynamic library to embrace modern interfaces without rewriting applications, and discusses the design tradeoffs.

⑥ Chapter 6 showcases our directive generation approach which takes into account temperature *and* data relationships, addressing the limitations of traditional approaches.

⑦ Chapter 7 combines the Shimmer and Parakeet into a unified system and evaluates a number of log-structured data-intensive applications to compare our approach against the state of the art.

III Epilogue

⑧ Chapter 8 presents the lessons learned along with a blueprint for storage and other accelerators which require host-device co-design.

⑨ Chapter 9 discusses the lessons learned and our vision for the future.

2
FALLING FLASH ENDURANCE

"SSDs will continue to improve by some metrics (notably density and cost per bit), but everything else about them is poised to get worse".

—Grupp *et. al.* • The Bleak Future of NAND Flash Memory [26]

Synopsis

Flash storage has seen the fastest growth of any storage media, but the techniques that allowed this growth are quickly reaching their limits. While flash will continue growing in density, the growth comes at the cost of performance and reliability. With this degradation, we want to minimize the impact of on-device garbage collection on cost, performance, and lifetime degradation on SSDs. We can significantly reduce on-device garbage collection with host insight allowed by modern interfaces like ZNS or Flexible Data Placement (FDP).

⁂

To understand the growth of flash and the limits to its growth, we need to start with the structure of flash cells.

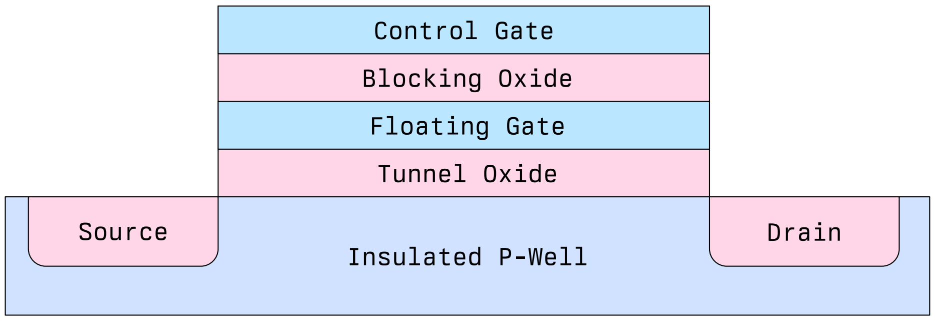

Figure 2.1: Flash cells use floating gate MOSFETs to trap charge which can encode values.

2.1 Scaling NAND flash

NAND flash is made up of sets of individual flash cells connected in series resembling a NAND gate. Each cell is made up of a floating gate MOSFET: a transistor made out of a metal-oxide semiconductor, typically Silicon. These cells store bits by storing charge in the form of electrons or holes which can be programmed at page granularity. Typical page sizes range from 512 B to 16 KiB. NAND gives up the random bit-level access capability of NOR flash to allow smaller cell sizes and increased density and is used widely in SSDs due to lower cost. NAND flash has grown faster than any other storage, doubling in capacity every year due to a combination of effective scaling techniques:

① **Process Shrink:** Each feature shrink allows 10^2–$10^3\times$ more bits.

② **Increased bit density:** More bits/cell have increased capacity $4\times$.

③ **Vertical Stacking:** With 192 layers, density goes up $192\times$.

As feature sizes reach single-digit nanometers, increasing bit density requires detecting charge levels with small numbers of electrons, slowing down access. Further fine margins in charge states mean even a minor degradation in the dielectric can make flash cells unreliable.

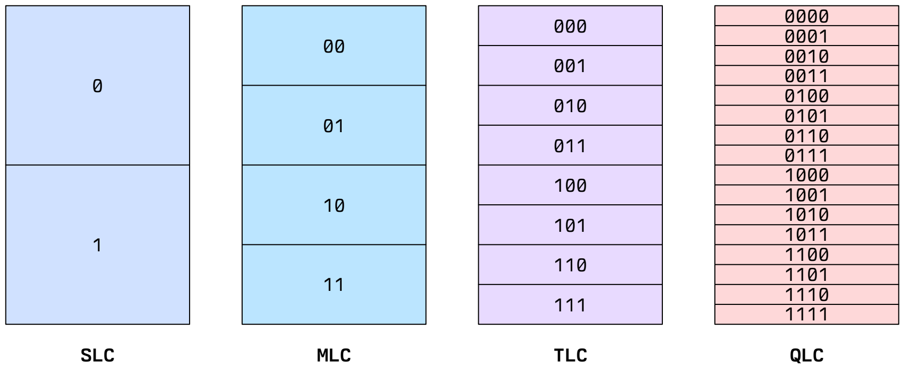

Figure 2.2: As we increase the density of bits in flash storage, the margin of error in charge detection shrinks, causing storage to degrade faster.

As seen in Fig. 2.2, measuring more charge levels increases bit density in flash storage. Multi Level Cell (MLC) flash doubled capacity with the same number of cells as Single Level Cell (SLC) flash, and Three Level Cell (TLC), Quad Level Cell (QLC) cells similarly achieved dramatic increases in capacity with the same amount of silicon. However, as we can see in Table 2.1, such an increase comes with performance and lifetime degradation. As margins for error become finer in reads, writes slow down, and bit errors increase. With a small feature size and increased number of voltage levels to maintain different bit values, writes are slowed down as more voltage pulses and read-verify operations are needed to store the exact values, along with rares issues like voltage drift and read disturb. Further, the likelihood of read errors increases; with a lower tolerance, tunneling is much more likely to impact stored values.

Performance and reliability degradation limits capacity increases by techniques like feature shrink and density increase. leaving any scaling improvements to layering and intelligent engineering which can absorb frequent updates on a more durable medium. We can see examples of QLC SSDs with large caches [58] or QLC-based cold storage shielded by more durable storage tiers [20]. A major reason for such degradation is the additional work an SSD needs to perform to ac-

Technology	bits / cell	Endurance (cycles)	Relative Capacity Increase
SLC	1	10^5	
MLC	2	10^4	100%
TLC	3	10^3	50%
QLC	4	10^2	33%
PLC	5	unknown	25%
HLC	6	unknown	20%

Table 2.1: Each generation of density increase lowers endurance and offers a diminished improvement over previous generation

commodate the block interface, which has largely remained unchanged from the tape pool days. Supporting in-place updates and random writes requires excess provisioning, greater SRAM and DRAM, and on-device garbage collection, raising costs and impacting the SSD's performance [41].

These requirements are particularly harsh on environments like smartphones, where the size and cost limitations require mapping the Flash Translation Layer (FTL) in host memory, consuming the already limited host memory for storage operations. With RAM and additional flash for over-provisioning, storage quickly becomes one of the most expensive components on a smartphone [29].

2.2 Impact of the flash translation layer

As seen in Fig. 2.3, SSDs are internally organized into channels, which are composed of super blocks, which are append-only regions where writes are persisted. Since NAND flash cannot perform random writes or in-place updates, these SSDs maintain in internal mapping, mapping logical blocks, which are exposed through the interface to physical blocks which reside on the device. Random write are simulated by maintaining the same Logical Block Address (LBA) while changing the mapping to point to a different physical block. This mapping is called the FTL.

The SSD controller besides maintaining the FTL, can leverage such mapping to perform operations like wear leveling, error correction, and garbage collection.

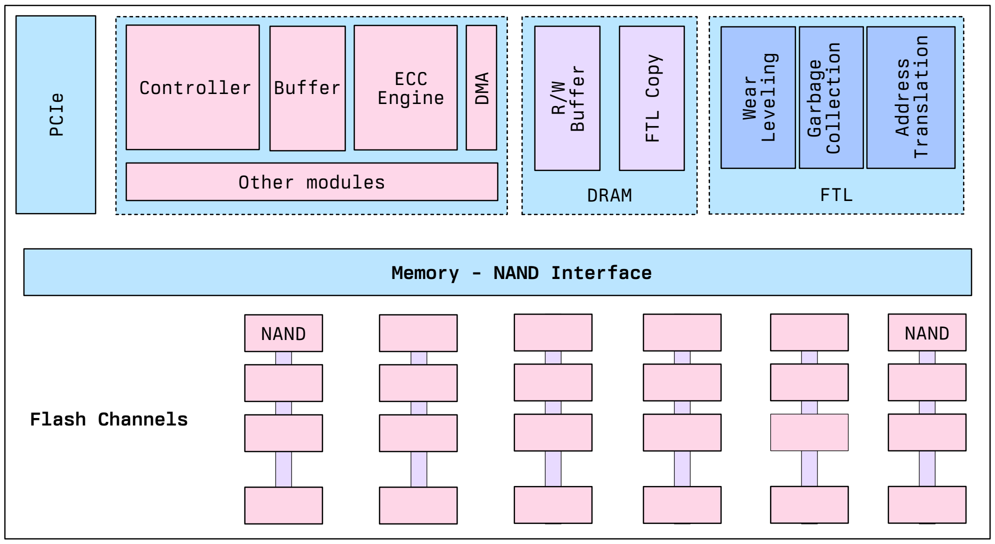

Figure 2.3: Besides the flash media, SSDs contain specialized processing units, buffers, flash translation layers, and logic to maintain flash.

The FTL needs to be cached in the device RAM to avoid slowdowns on random access, taking up about 1 GB for 1 TB of flash [14]. As SSDs get larger, DRAM costs go up significantly, impacting the price or performance of flash. To make matters worse, while writes can be persisted at page sizes (4 KiB), once written, data cannot be updated unless erased first. Further, as NAND flash requires a large voltage to reset cells, erase units are hundreds of pages, and can grow to sizes of several megabytes. Garbage collection is required to move valid data in the range being erased to a new location.

On-device garbage collection is the leading cause of write amplification and slowdowns on SSDs [61]. This operation must select regions to erase, relocate valid data, and then perform the erase. Even with internal SSD bandwidth, copying data takes hundreds of microseconds and an erase operation of dozens of microseconds. Further, if an application tries to read data being relocated, it may have unexpected millisecond-scale latency spikes. On-device garbage collection adds a factor of 3 to 5 write amplification, adversely impacting device lifetime. To minimize the impact of garbage collection, SSDs typically have 17-28% over-provisioned space, which further drives up the cost of SSDs [67].

Cost reduction and flash scaling limitations have driven manufacturers towards newer interfaces that simplify the FTL and address the impact of garbage collection. However, as we will see in § 2.3, such changes can break compatibility with existing interfaces and blur clearly defined storage stack abstractions with the need for host input.

2.3 Efforts to update flash interfaces

Data management systems have used the same block storage interface for decades while assuming that the underlying storage devices have similar characteristics. Efforts to update SSD interfaces fall into three broad categories: 1. Hints, 2. Host-managed, 3. Specialized. In the first approach, hints can improve performance and lifetime, but are *optional*, and applications can use these SSDs as conventional SSDs. Stream SSDs, and FDP SSDs follow this approach. In the *host-managed* approach, the host takes over certain responsibilities, such as data placement, resource management, and garbage collection, while the device gets simpler. Open Channel SSDs and their successors, ZNS SSDs follow this approach. Finally, several specialized SSDs offer a domain-specific interface; efforts such as key-value SSDs, computational SSDs, and memory-semantic SSDs open up an interface designed for a specific workload. To limit the scope of this work, we will focus on general-purpose interfaces that use hints or are host-managed. Since specialized SSDs may not offer classical storage interfaces, our systems do not target them.

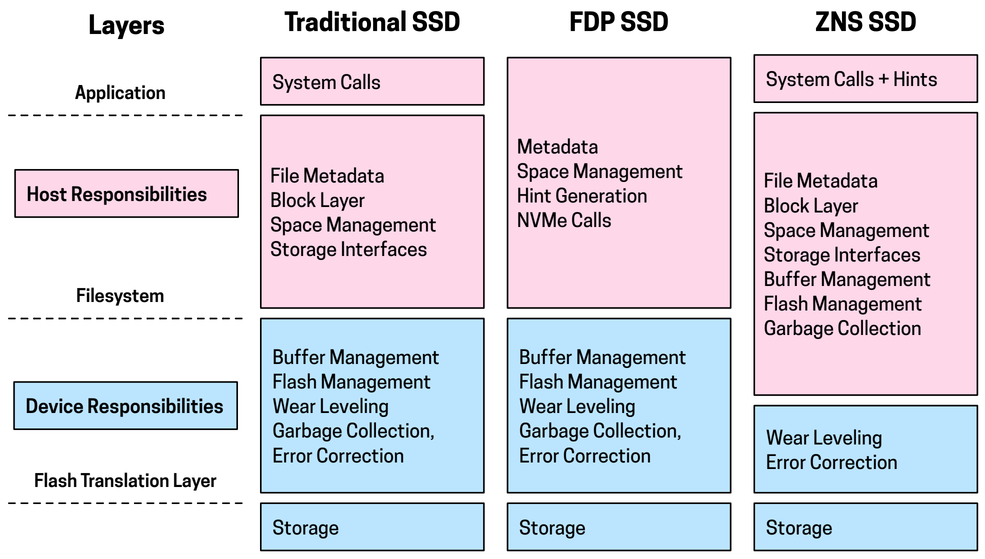

Figure 2.4: Updates to SSD interface in host-managed interfaces like Open-Channel and ZNS require the host to take over some tasks traditionally handled by the device, while hint-based interfaces largely maintain the traditional stack.

2.3.1 Hint-based interfaces

Multi-stream SSDs [31] were introduced in 2014 to allow data to be annotated with a stream ID, which the device could then use to partition and place data internally. Data in the same stream could be allocated to adjacent physical blocks regardless of logical block address. These devices supported up to 4 streams, relying on the host to indicate related data. The Linux kernel eventually added support for these hints through `fcntl()` with `F_SET_HINT` interface. Like the SSDs, these hints support four stream IDs: Short, Medium, Long, and Extreme lifetime. However, the complexity of using relative hints, especially in a shared system, along with limited device support, meant the protocol missed wide adoption. Ultimately, the Linux kernel maintainers removed these hints, only to bring them back as ZNS filesystems utilize them.

2.3.1.1 NVMe flexible data placement

Flexible Data Placement [51] is the most recent SSD standard, a joint effort from Google and Meta to standardize hints based on hyperscalar data center needs. These devices break down SSD superblocks into *reclaim units*, which can be written by indicating the *placement identifier* in the Non-Volatile Memory express (NVMe) write command. The device can erase these reclaim groups as a unit and reduce garbage collection overhead if they hold data with a similar lifetime.

The major advantage of the FDP interface is the introduction of placement directives without affecting compatibility with current applications. Broadly, FDP SSDs have abstractions for grouping (Reclaim Groups, Reclaim Units, and Reclaim Unit Handles) and abstractions for placement (Placement Identifier, Placement Handle). A placement handle maps to a reclaim unit handle that references the current reclaim unit within the current reclaim group. An integer value `DTYPE` in the NVMe write command indicates the placement identifier.

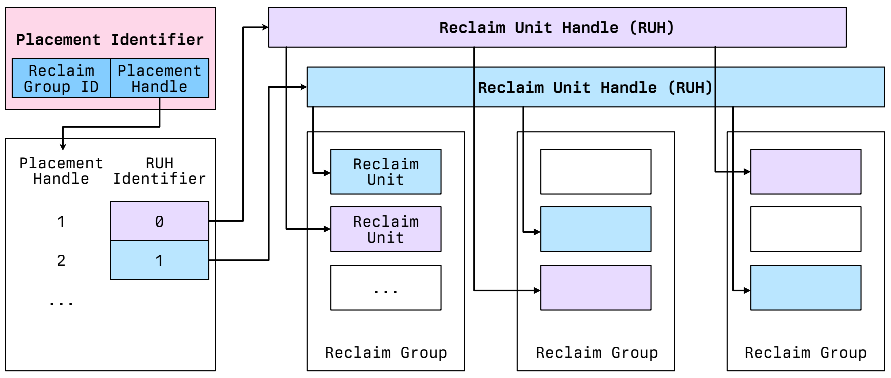

Figure 2.5: Hints in the FDP interface are given through the placement identifier, which indicates the reclaim group as well as the placement handler.

Currently, no Linux Kernel APIs are available for hints within the FDP protocol (by design). These hints are integrated into NVMe write commands, a much lower-level interface than the one used by applications. The current goal of this work is optimized data center architectures and not necessarily broad compatibility and use by consumer devices. FDP SSDs are more complex than traditional SSDs, maintaining the FTL and adding the placement logic on top, and hence do not offer similar cost savings to host-managed interfaces, which can reduce the DRAM and over-provisioning requirements. The work discussed in this dissertation adds preliminary support for injecting hints into applications that support FDP. Still, we focus mostly on host-managed architectures due to the lack of availability of APIs and devices.

2.3.2 Host-managed interfaces

Open Channel SSDs began as an effort to simplify devices, offloading the task of flash management to the host. Since the host is aware of the application and workload characteristics, it can make better placement and garbage collection decisions than storage, which has no access to the host state. LightNVM [15] allows a host-based FTL to manage flash. However, this approach breaks block-layer compatibility and assigns additional responsibilities to the host, such as wear leveling and garbage collection. Further, manufacturers cannot offer performance or lifetime warranties since they depend on host actions, which is a non-starter. Overall, open-channel SSDs were well-suited for research or specific deployment but were hard to use as general-purpose SSDs and never took off. The NVMe consortium refined the lessons from open-channel SSDs into the ZNS interface [12], which reduced the host responsibility to placement and garbage collection, returning wear management and error correction to the device as seen in Fig. 2.4.

ZNS SSDs [66] partition the storage device into a set of equally-sized zones supporting append-only writes. Unlike a traditional SSD, ZNS requires writes with a zone number, and offset, supporting writes only at the *write pointer* in each zone, which is the tail of the log maintained in the hardware. All other writes are rejected. Data becomes immutable whenever appended to the tail of a zone, and entire zones can be made immutable either by an explicit *finish* command or automatically when they fill up. The device does not perform any garbage collection, requiring zone reset commands from the host to reclaim space. For a ZNS SSD, reads can be performed similar to conventional SSDs; both sequential and random reads are supported. Further, reads can also be performed with a zone number and and offset. ZNS requires the host to perform resets at zone granularity, eliminating on-device garbage collection and device-side media over-provisioning.

While this represents a limitation over traditional block-based SSDs, it offers greater control to the host over internal SSD operations such as data placement and erase.

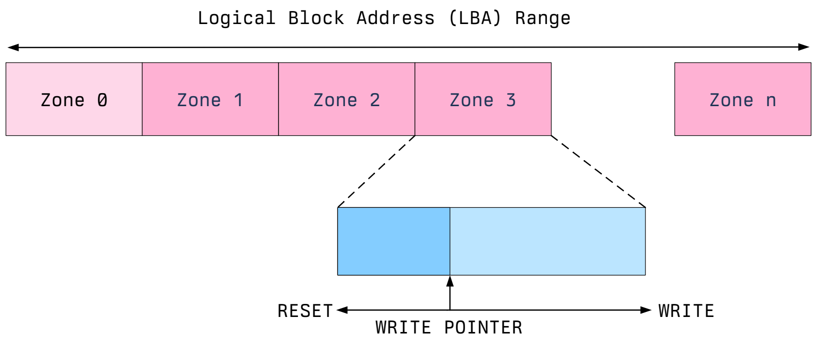

Figure 2.6: ZNS SSDs expose several equal sized append-only regions which maintain their tail through write pointers.

The general advantages of ZNS SSDs have been described at length in literature [14, 55]. We focus on advantages that are of particular relevance for systems architecture:

① **Data Placement:** ZNS SSDs provide greater control over data placement. For example, the device can align the host logs to zones and bridge garbage collection across levels reducing data movement.

② **Data Immutability:** Zones become immutable when full or closed, simplifying many operations such as version management, replication, and reorganization [27].

③ **Optimized Logs:** Applications can use the Zone Append primitive to simplify the bookkeeping needed to support log-structured applications [50].

With ZNS SSDs, we have an opportunity to align log-based systems with the log-based interface exposed by the hardware. Elimination of random writes and host-controlled garbage collection create new challenges and opportunities for data and metadata management. The ZNS append-only interface may not be appropriate for all aspects of data management. Additionally, ZNS SSDs have a resource limit on the number of zones that can be "active" (available for writes or appends), and managing these resource constraints is an additional responsibility of the host.

2.3.3 Specialized approaches

Over the years, besides block-based storage interfaces, manufacturers have introduced various types of *specialized* SSD interfaces designed for domain-specific purposes. These include memory-semantic load-store approaches using Compute eXpress Link (CXL) [69], the key-value SSD format [33], various types of SSDs which include compute resources [32, 65], proprietary approaches like Amazon's NitroSSD [7], and flash array solutions by companies [18]. These techniques unlock novel applications: for instance, offloading compute to storage or load-store operations on an SSD instead of block IO. Others allow vendors and cloud companies to achieve something similar to what ZNS and FDP do within their data centers. Key-value SSDs use an open standard that offloads operations related to key-value data management systems to storage, offering a `get()` and `put()` interface. These techniques offer a lot of promising improvements. However, this dissertation will not discuss these since they do not offer a general-purpose storage interface. They are designed to be application-specific, use-case-specific, or to eschew storage interface and replace it with a load-store interface. However, their special-purpose nature limits their usability outside of applications tuned for these interfaces; hence, we do not focus on them.

2.4 Kernel interfaces and hints

Of course, the devices are only one part of the puzzle; the major reason for the lack of wide deployment of modern SSDs is that kernel or programming interfaces are unavailable or frequently changing. Plus, applications are storage-unaware, even with a kernel hint interface, and are unlikely to be tuned for a small set of SSDs. As seen in Table 2.2, many of these SSDs allow support for placement in-

	Stream	**Open-Channel**	**ZNS**	**FDP**
Introduced	2016	2017	2021	2022
Linux Kernel	Deprecated	Deprecated	Available	–
Interface	`fcntl()`	`lightnvm`	`libzbd`	`libnvme`
Filesystems	–	–	F2FS, btrfs	–
Applications	AutoStream	RocksDB	RocksDB	Cachelib

Table 2.2: Support for novel SSD interfaces in applications and filesystems is typically limited to one or two applications due to the complexity of adopting these interfaces while maintaining backward compatibility.

dicators through the NVMe interface, which on Linux systems can be accessed through `libnvme` or `xNVMe`. Since applications do not use NVMe commands directly[1], without filesystem APIs, this approach forces placement and hint generation logic to be implemented in the driver or the filesystem. If used in traditional kernel hierarchy, filesystems and drivers are in the kernel and lack visibility into applications on top of them. Further, filesystems are hard to modify and update,

1 Some highly optimized applications like Cachelib can use NVMe commands directly.

requiring kernel upgrades and downtime, and adding complexity can introduce bugs and security vulnerabilities [56]. Systems like stream SSDs can utilize the `RWH_LIFETIME_HINT` interface through `fcntl()`, which can allow applications to specify hints. These hints were deprecated from the kernel but returned after systems like ZenFS and F2FS started using them on top of the ZNS interface [2]. The ZNS interface also supports `libzbc` (SCSi) and `libzbd` (NVMe) libraries to allow applications to support these devices better.

Besides `fcntl()`, `fadvise()` has been typically used to indicate file usage. Applications will typically use the `fadvise` system call to indicate the workload of the file to the filesystem to influence read-ahead and caches of the file. Hints such as `FADV_SEQUENTIAL`, `FADV_RANDOM`, and `FADV_NOREUSE` allows the filesystem to make the appropriate decisions. On the other hand, the lifetime hints present four lifetime levels: short, medium, long, and extreme. Neither of these interfaces currently captures the types of hints needed for these devices, presenting an abstraction that is too general and not transferable across applications. This dissertation will examine how these interfaces fail to capture rich hints and how we can improve upon them.

⁂

Summary

Efforts to increase flash density will come at the cost of performance and lifetime degradation, which are further exacerbated by the impact of garbage collection, a result that we get from not updating storage interfaces to flash. Various updates have been proposed: updates that are hint-based or host-managed. However, adopting these interfaces has not picked up due to the complexity of the additional tasks, storage abstractions, the cost of rewriting applications, changing interfaces, and the lack of kernel-level hint support.

We cannot effectively address the fundamental limitations in scaling flash without breakthroughs in material sciences. These limitations are from the physical properties of silicon and have been staved off so far due to *one-off tricks* and clever engineering[2]. To keep SSDs growing, we need to work with these limitations, addressing the harmful effects interfaces and systems have on flash as they are unaware of its limitations. It is time we stopped treating flash as a *faster* tape. The block layer and POSIX operations are outdated primitives to work effectively with flash storage. We can leverage architectures like ZNS and FDP to change how we use SSDs, using host insights to eliminate garbage collection overheads.

Chapter 3 explores the challenges introduced by adopting these interfaces and some of our solutions. We look at the design space and, more importantly, the trend of modern SSD architectures and their lack of adoption, analyze the causes, and propose a direction. Each of these technologies has been demonstrated with one or two examples, suffering the lack of widespread adoption until applications and operating systems stop supporting them. If we want to keep using flash storage and value open standards, we must address these limitations, make it easy to use these devices, and provide hints.

2 3D stacking of flash, for instance, allowed the density to keep growing, with up to 200 layers of flash cells laid on top of each other.

3
THE MODERN SSD DESIGN SPACE

"I still don't understand the play." "Doesn't matter. Just keep on telling the story."

—Wes Anderson • Asteroid City

Synopsis

Chapter 2 looked at the limitations of modern SSD and efforts made to update interfaces. In Chapter 3, we will explore the challenges that prevent widespread adoption and the overall design space. This chapter will explore what these interfaces mean for systems and applications, their tradeoffs, and the missing pieces. Since there is little guidance on how these technologies work, we discuss data placement strategies and various approaches that can allow us to utilize these interfaces. In this chapter, we hope to shed light on the modern SSD design space and current efforts, where they fall short, and where our work fits.

⁂

While hardware manufacturers keep innovating on SSD interfaces, these SSDs have not seen large-scale adoption, often being limited to 1-2 proof-of-concept examples as they require significant expertise in the interface, and updates can make the system less generalized and will require significant re-engineering on further interface updates. While it may seem surprising, the trend is that hardware interfaces change *faster* than the application or system running on top, driven by the urgency of required change or potential benefits offered by the new technology, requiring significant effort to keep applications running unmodified[1]. Conversely, applications continue working with POSIX and Virtual FileSystem (VFS) interfaces, largely unchanged from the 1990s [62].

History tells us that changes that break compatibility face an uphill task with adoption. Open-Channel and Multi-Stream SSDs were never widely adopted due to the effort required to use them effectively. Storage interfaces have traditionally been abstracted from applications, and for good reason: they provide us with portable software that is less complex and can be used with a variety of storage media. That leaves us with two choices: approaches like optional hints, which can be ignored if unsupported, or utilizing interfaces like POSIX to abstract the additional tasks. Since these hints depend on the lower layer to effectively utilize them, we focus on the POSIX layer, and this dissertation presents solutions in form of a filesystem and a per-application shim layer.

However, there is little guidance on choosing the right SSD for the job and issuing hints. While loosely grouping files by lifetimes is needed, that still leaves it up to the systems, which are complex and hard to modify, and the people working on them may not be familiar with the hint interfaces. We aim to distill our lessons from working on such devices and provide a framework that can ease adoptions and maximize the metrics each use case cares about: from device cost to throughput. This chapter presents the available interfaces, compares them, and discusses the state-of-the-art systems that utilize them. We will then talk about the limitations of these systems and how our approach fits in this design space.

1 Seen in Apple's transition from PowerPC $\rightarrow$ Intel $\rightarrow$ ARM architecture by Apple, significantly helped by the Rosetta binary translation layer.

3.1 Benefits of host-device coordination

On modern SSDs, placement hints can allow the device to partition data streams on different hardware regions and use separate write buffers to avoid interleaving unrelated data on logs. Such grouping can also reduce valid data relocation on erase.To demonstrate the benefits of host-device coordination we ran a simple experiment on a host-managed Western Digital ZN540 SSD. This device uses the ZNS protocol, allowing the host to pick which zones to write to and erase, while enforcing append-only writes. We made two 128 GiB partitions on the SSD formatting one with F2FS and the other with ZoneFS (filesystem layer projection of device zones). We then used the flexible IO tester (`fio`) [24] tool to sequentially write 1 GiB of data to a single file on F2FS and ZoneFS. We then scaled this benchmark to 1, 4, 8, and 14 (the maximum allowed open zones on ZN540) independent files on each system using the `psync` engine. To enable fair comparison, we used Direct IO, skipping the buffer cache, and for F2FS, we provided *sequential* and *extreme* lifetime hints using the `fadvise` and `fcntl`. For ZoneFS, we mapped writes to distinct zone files.

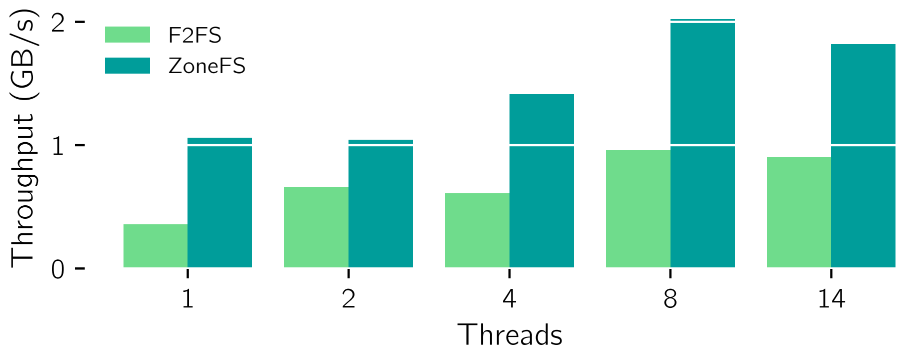

Figure 3.1: For the simplest case: sequential writes with no overwrites F2FS on the same hardware gets a fraction of throughput compared to ZoneFS.

As we see in the experiment in Figs. 3.1 and 3.2, a lightweight mapping layer with the right hints (map each file to a separate zone) can provide full device throughput, while F2FS is limited to 30–50% of the bandwidth due to the overhead of coordinating writers, and mapping writes to segments across its three temperature-separated logs. Coordination of separate write streams further incurs 2–3$\times$ higher latency. F2FS is further slowed down by its internal node and metadata updates as it needs to map larger contiguous regions to smaller segments. This limitation in adopting a zoned interface is fundamental to the design of F2FS as it cannot provision more hint levels than the number of logs it maintains. Modern log-structured systems have several writers with varying workloads: write-ahead logs, data logs, compaction, checkpoints, recovery, and replication. Even on log-structured filesystems, such streams get interleaved, increasing fragmentation and limiting performance isolation.

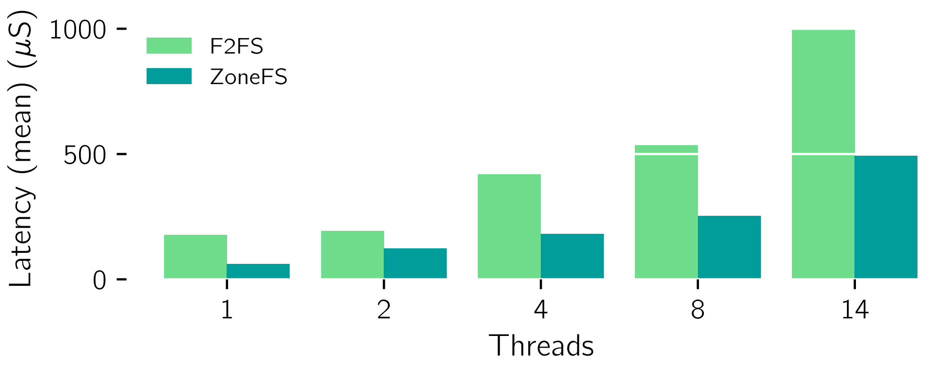

Figure 3.2: For the simplest case: sequential writes with no overwrites F2FS on the same hardware get worse latency than ZoneFS.

However, it is important to note that F2FS is a full-fledged filesystem while ZoneFS is closer to a raw block device, but it is also clear that short of a significant redesign, F2FS will find it difficult to maintain backward compatibility with traditional block devices while embracing modern interfaces. Fundamentally, hardware interfaces change faster than software, driven by changing media or necessities, while software interfaces like POSIX have largely remained the same over the last 30 years. Another problem with embracing such interfaces is that they may change over time, requiring frequent rewrites and redesign of approaches which is infeasible. As seen in Table 2.2, no application or filesystem currently supports Stream or Open-Channel SSDs. Four years after the introduction of ZNS SSDs, only RocksDB, btrfs, and F2FS can efficiently utilize it, with known issues in F2FS and btrfs support. We will examine why these SSDs are hard to accommodate in the filesystem or application layer and need a separate layer.

Overhead of kernel mechanisms

F2FS incurs significant overhead in the kernel, with several operations which consume time. In the `fio` benchmark above, it takes up 47% of the total CPU time of the benchmark for `f2fs_file_write_iter` of which buffered writes require 13% of the time as seen in Fig. 3.3. Despite direct IO, F2FS must buffer the parallel writes to map them across its six logs.

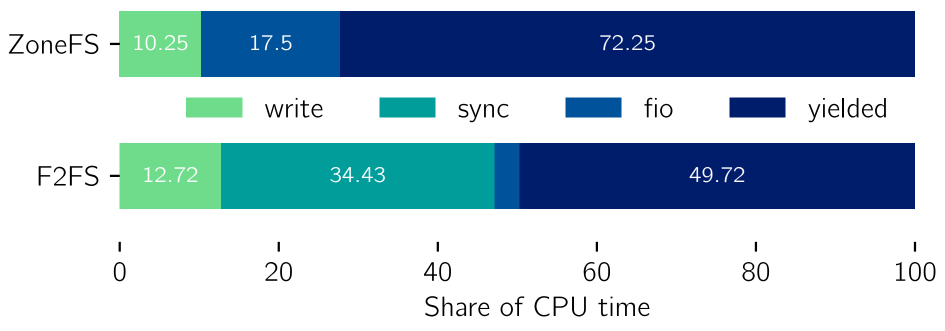

Figure 3.3: In the `fio` test, F2FS consumes 47% of all CPU time, adding significant overhead on top of write, while it makes up only 10% of ZoneFS time, due to reduced overhead of mapping files to a limited set of logs and metadata operation reduction.

ZoneFS, on the other hand, by skipping the page cache, minimal metadata, and no buffering, manages only to consume 10% of the `fio` cycles as seen in Fig. 3.3, offering a flat overhead without any slow or locking in-kernel mechanisms. The lack of coordination further helps improve latency and throughput; ZoneFS sees zero write operations that take more than 200 µs, compared to F2FS' 652.

3.2 Choosing the right technology

With a smattering of SSD types – Conventional SSDs, ZNS SSDs, FDP SSDs, CXL SSDs, Smart SSDs, and Key-Value SSDs, and other domain-specific SSDs, available hardware interfaces are more diverse than ever. Picking the right interface or combination of3 interfaces becomes essential. Adoption is further complicated because no systems can cohesively support a complex combination of such interfaces. With short lifetimes, it is no surprise that companies do not adopt these technologies immediately.

Fundamentally, these SSDs fit into different classes: (i) Storage: Conventional SSDs and FDP SSDs stick to the standard storage interface, while ZNS and Key-Value SSDs present a more optimized interface. (ii) Memory: CXL allows memory semantics on an SSD, presenting a load-store interface. Each storage and memory technology can support computation with FPGAs or Microprocessors making up the (iii) Compute SSDs. This dissertation limits itself to the discussion of the Storage class of SSDs, as persistent memory and computing on storage are separate research domains, however we discuss some preliminary designs to accommodate such devices in Chapter 9.

With the two diverging approaches taken by *hint based* SSDs and *host managed* SSDs it becomes challenging to pick the right SSD for a system. On one hand hint-based approaches like FDP are optional and can ensure that applications will keep working with the SSD despite not being adopted for the technology, but in many ways they maintain the high cost, memory requirement, and over-provisioning of traditional SSDs. Further, unlike host-managed approaches they cannot guarantee that background operations or on-device write amplification will not occur.

3.3 Limitations of the state of the art

Upgraded SSD interfaces have generally ended without any significant adoption. While newer approaches have tried simplifying these interfaces, adoption is still limited to a couple of applications or specially designed hyperscalar systems. In this section, we discuss such systems and why they limit the wide adoption of these interfaces. Approaches to effectively use these new interfaces have used general or special-purpose filesystems and modified applications. In this section, we will look at these approaches and why they are insufficient for wide adoption.

3.3.1 Filesystem design limitations

① *Filesystems are application-unaware:* Mainstream filesystems are designed to be independent of the applications running on top of them and find it difficult to generate useful hints unless communicated by the application. Currently, no standard interfaces exist: for example, F2FS can utilize hints from the StreamSSD interface and map it to zones, but with three hint levels across all applications, it is insufficient.

② *Filesystems are hard to modify:* As filesystems reside in the kernel, they are hard to modify and upgrade. Fixing ZNS-related bugs in F2FS, for instance, requires upgrading to a newer version of the Linux kernel, which is impractical for data centers as it requires migration and downtime and may cause issues with other applications. Adding complexity to the kernel can give rise to crashes and security vulnerabilities.

③ *Filesystems require broad compatibility:* Adapting a filesystem for ZNS, for instance, should not limit support for other types of SSDs. The increasing complexity of a filesystem can result in increased bugs in the kernel, reduced performance, and an increased attack surface.

Filesystem approaches generally introduce a new filesystem or adapt an existing general-purpose filesystem. The latter allows POSIX semantics, while the former generally involves application-specific backends. Overall both of these approaches have significant constraints that limit their success. POSIX filesystems need to ensure wide compatibility, and if popular filesystems such as ext4 [40], ZFS [16], XFS [57], F2FS [36], and btrfs [48] are modified, they need to combat design meant for a more traditional environment and concerns of backward compatibility. Often the changes do not make it through the Linux kernel code review process [22]. In a monolith like the Linux kernel, filesystems are bundled and updated with the Kernel requiring kernel upgrades to get newer versions which makes updating them averse to large-scale changes for limited utility. Further, as vulnerabilities can impact system security and bug fixes require kernel upgrades and down times, filesystems are not a preferred approach for many of the updates.

While some of these issues could be addressed, for example, with a FUSE [60] filesystem, it would still require a per-architecture per-application filesystem to utilize host-device hint mechanisms fully. Such usage of FUSE would be similar to our proposed shim layer but with the added complexity of managing per-application filesystems. Further, the overhead of repeated kernel crossings will negate any performance benefits from modern storage interfaces. Filesystem support for hints on stream SSDs was introduced [3] into the Linux kernel only to be dropped [5] due to lack of use, but subsequently reintroduced and repurposed to provide hints with the zone interface [59]. Open Channel SSDs never formally got filesystem support due to complexity of implementing the interface. With no plans for a kernel interface for FDP [52], ZNS is the only protocol which is supported by filesystems.

Special-purpose ZNS filesystems. ZenFS [13] and ZoneFS [44] are designed for the zoned interface. ZoneFS [44] exposes each zone as a single file. These files support writing data but do not support any changes to the filesystem layout, serving as a block-layer projection of the underlying device. ZenFS [13] is a plugin for RocksDB that exposes each zone as a writable set of extents that can be appended. However, ZenFS does not support the POSIX interface and systems that require in-place updates. Further, it does not perform hot-warm-cold separation (outside of what is achieved from the LSM-tree-based RocksDB engine) and does not support general primitives like directories and access control.

POSIX filesystems on ZNS. btrfs [48] is a Copy-on-Write filesystem with experimental support for Zoned Namespaces. It achieves an append-only design by enabling a rolling buffer of zones for the superblock, the only fixed structure, an approach that inspired our checkpoint design. We picked the multi-log approach of F2FS over btrfs as discussed in § 4.1.

F2FS [36] is well-suited for a zoned interface but has several limitations in its current version, namely requiring random-write areas for metadata and checkpoints, and excessive over-provisioning which is not designed for a ZNS interface. We use F2FS as the baseline for the log-structured interface it provides, along with temperature-specific logs. However, we updated the metadata management to not depend on in-place updates, improving performance and reducing write-amplification

Other filesystems like EXT4 or ZFS could be adapted to the zoned interface. However, they will require significant re-engineering effort and will not benefit from the flash-optimized decisions in F2FS. Systems such as RocksDB can work with ZNS but do not present a POSIX file interface or allow in-place updates, requiring rewrites of applications for a new storage interface.

Various SSD architectures have been proposed to enable host-device hint communication. NVMe Streams [10] (2016) reduced write amplification by allowing hosts to apply temperature hints on writes mapped to dedicated regions by the device controller. Open-Channel SSDs [15] (2017) allowed complete host control over data placement and garbage collection by allowing host-managed Flash Translation Layers. Open-Channel SSDs were refined into ZNS [12] (2021), with an aim to reduce write amplification and on-device garbage collection using host-managed append-only device partitions. FDP [51] (2023) improved streams by redirecting related writes based on host communication to the same device region without introducing a new SSD interface.

3.3.2 Limitations of modified applications

① *Rewriting applications is expensive:* Rewriting applications for a specific architecture requires significant engineering effort. For instance, the ZenFS project, which optimizes RocksDB for ZNS, is a multi-year effort with thousands of lines of code [13].

② *Applications are storage-unaware:* Applications have traditionally used the file interface and are abstracted from the nature of storage. They are unaware of system usage or other applications, resulting in any efforts of resource acquisition and hinting being ineffective.

③ *Modifying applications limits layering and portability:* Even if an application is customized end-to-end to use custom interfaces, it cannot then effectively address multiple types of devices. Breaking away from the file abstraction can hurt other entities operating on the data, like backup programs and other maintenance scripts.

Prior work in using these interfaces involved modifying existing applications [63, 72] or using custom file systems like ZenFS and FStream [45, 47]. Applications such as RocksDB, Percona server, and MyRocks have been adopted to the ZNS interface through the ZenFS [13] plugin. WALTZ [37] optimizes further over ZenFS, reducing the tail latency with the help of the zone append command.

Custom file systems are both application and device specific and are unsuitable for use in cloud environments comprising heterogeneous applications and storage architectures.

Interposition approaches outside the file system have used either eBPF [73] or SPDK [70] as kernel-bypass mechanisms. `syscall_intercept` [21] can overload system calls by disassembling and patching binaries. The approach we discuss in Chapter 5 is similar to `syscall_intercept` but does not have the overhead of disassembling and patching binaries. A few `LD_PRELOAD`-based filesystem prototypes exist, like PlasticFS [42] and AVFS [28], which allows peeking into compressed files. Goanna [23] implemented a filesystem through ptrace extensions, similar to the `LD_PRELOAD` technique in spirit. More recently, zIO [54] used user-space libraries using `LD_PRELOAD` to eliminate unnecessary copies of data. Several FUSE [60] filesystems have been implemented for optimized data placement. For instance, PLFS [8] optimizes for parallel checkpoints to map it optimally to underlying filesystems.

Our approach uses interposition to inject hints and remap data if needed, while no other systems interpose between the application and the filesystem. Similar techniques have been implemented in different layers of the stack. Cloud Storage Acceleration Layer [71] enables the adoption of ZNS with clusters of varied storage and a host-based flash translation layer. However, such an approach is intended to be a solution with no application input and uses various types of storage to balance out random vs. sequential accesses.

3.3.3 Missing hint APIs

As we discussed in § 2.4, one major factor that impedes the adoption of modern SSD interfaces is the lack of application programming interfaces for providing hints. Hints in the Linux Kernel are still limited to 4 temperature levels shared across applications, a legacy from the stream SSD interface. It is impossible to provide directives for FDP from any program that does not directly issue NVMe commands- something only provided by highly optimized programs like Cachelib with limited support [25]. Even with these hint interfaces, the hints produced rely on temperature rather than lifetime. We propose a new hint API in Chapter 6 that accounts for temperature and lifetime and is generalizable across different interfaces. It removes arbitrary limitations on the hint interface, allowing simple but rich per-application hints.

3.4 Our approach

To address the fundamental issues discussed in this chapter we present three systems that we worked on:

3.4.1 Updating filesystems

In Chapter 4, we introduce Persimmon a filesystem based on F2FS with append-only metadata which requires no in-place updates, no fixed block addresses, and is the first zoned filesystem that works efficiently even on devices with no random writes. We show that not only is it possible to have such a system, but it can keep up with flash-optimized filesystems while significantly cutting down on garbage collection overhead.

3.4.2 Enabling rich hint interfaces

Hints on SSDs are helpful for two main purposes: isolation of write streams improves performance and grouping related data reduces device wear. For a *good* hint, a system must separate independent write streams (such as data logs, write-ahead logs, checkpoints, and manifests) into separate groups. Such grouping will eliminate the interleaving of streams on device buffers and flash, improving performance due to device-level isolation and parallelism. Additionally, the hints must help the system group data by its *expected* lifetime, reducing the garbage collection overhead. Kernel interfaces such as `RWH_HINT` and `fadvise` offer limited levels of hinting, so for our hint generation technique (see Chapter 6), we implemented a more flexible hint format which we can convert into these formats, but also captures more information for future, more sophisticated APIs.

In this work, we focused on log-structured systems for two main reasons: (i) they are already flash-optimized and can be easily translated to append-only interfaces like ZNS, and (ii) they achieve a natural temporal grouping: incoming writes are written to the log which is compacted with files in the same level deleted together. Further, users can predict the file size and lifetime with the right compaction strategy, making data placement and grouping easier.

We propose a new hint interface in Chapter 6, demonstrating how application-tuned hints with heuristics and machine learning. Parakeet captures lifetime and data temperature, allowing richer hints designed to improve performance by isolating independent data streams on separate regions and lifetime by grouping data related by lifetime.

3.4.3 Shim layers to ease transition

While new hardware interfaces enable better performance with host-device communication, they increase the complexity of applications and storage stack. Applications or filesystems must be rewritten to implement host-side hints for a heterogeneous set of ever-changing interfaces. We can avoid application rewrites with filesystems that absorb hint generation, but filesystems are often harder to modify as they typically reside in the kernel. It is no surprise, therefore, that the various SSD architectures have seen limited adoption outside of one or two applications used to demonstrate their potential. For instance, as seen in Table 2.2, no application or filesystem currently supports Stream or Open-Channel SSDs. Four years after the introduction of ZNS SSDs, only RocksDB, btrfs, and F2FS can efficiently utilize it, with known issues in F2FS and btrfs support.

Considering the limitations of filesystems (discussed in § 3.3.1) and applications (see § 3.3.2), we believe that a separate *shim* layer is the best place to *isolate* the complexity of adopting new interfaces rather than applications or filesystems. Such a layer would abstract interface changes from the applications and filesystems while enabling low-overhead reconfiguration to get most of the benefits of modern SSDs. In this architecture, simplicity is maintained in the application and the filesystem, while the added complexity of hinting is isolated in a configurable layer, enabling low-cost adoption.

In Chapter 5, we demonstrate Shimmer, a shim layer that eases the adoption of new interfaces by operating on these interfaces with no change to the application or the operating system, leveraging hints provided by Parakeet to outperform existing filesystem-based and application-specific solutions. Shimmer can also be used with Persimmon to support hints for a wider set of applications, allowing all of these systems to work together as an end-to-end system, which unlocks all benefits of modern interfaces with no change in application logic.

⁂

Summary

In this chapter, we examined how to pick the right device for applications and how support for modern interfaces is implemented in the state of the art. This support leaves much to be desired and has been the reason adoption has suffered for these devices. We see that filesystems only support one type of interface, often not implementing it to maximize performance, and application-specific backends are hard to adopt and enable one application after a significant engineering effort. There is also a lack of a good hint interface, impeding hint generation and provision. Our systems are designed to overcome these issues and can work in tandem to unlock the full capabilities of modern interfaces at no cost.

PART II

SYSTEMS FOR MODERN FLASH

“I have learned—but again and again I forget—that abstraction is a bad thing, innumerable and infinitesimal and tiresome; worse than any amount of petty fact. …It is like a useless, fruitless vegetation, spreading and twining and fading and corrupting; even the ego disappears under it…”

— Glenway Wescott • The Pilgrim Hawk

4
PERSIMMON: APPEND IS HERE

Accountants don't use erasers; otherwise they may go to jail. All entries in a ledger remain in the ledger. Corrections can be made but only by making new entries in the ledger. When a company's quarterly results are published, they include small corrections to the previous quarter. Small fixes are OK. They are append-only, too.

—Pat Helland • Immutability Changes Everything [27].

Synopsis

Host–managed devices enforce device constraints such as append-only writes, which rule out in-place updates, random writes, and mechanisms that may result in unordered writes like `mmap()`. Most filesystems still depend on unsupported paradigms like fixed addresses and in-place updates. So, while filesystems present a good way to abstract hardware changes from the application layer, they require require random-write regions for their metadata, requiring complex setups or compromises on filesystem size to support interfaces like ZNS.

In this chapter, we present Persimmon, a fork of the F2FS filesystem built with append-only metadata structures to avoid any in-place updates or need for known locations. Persimmon updates F2FS with new in-memory structures for better management of zones, improves checkpoint logic and uses append-only metadata management. In addition to reducing tail latency spikes and unlocking more usable space, Persimmon presents the first filesystem that does not depend on any in-place updates or fixed super block locations. Persimmon achieves this by embracing hardware-software coordination, using log tails maintained by the SSD to store its metadata.

⁂

The standard way to add support for new types of storage is to enable general purpose, POSIX-friendly filesystems. However, the complexity of adapting current systems to an append-only interface has hurt the adoption of these SSDs. Even log-structured append-only systems such as F2FS, btrfs, and RocksDB still depend on known locations and in-place updates for various metadata operations. These systems typically employ auxiliary drives which support in-place updates while maintaining the logs in append-only regions. These random-write drives still need to perform garbage collection, which can cause slowdowns, and complex multi-namespace or multi-device setups can impact the performance and stability of such systems. Further, the limited compatibility updates can mean odd characteristics in practical use: for instance, F2FS has as low as 36% of the total size as usable (depending on the configuration) with ZNS [53], and btrfs uses less than half the capacity of each zone.

Requirements for a random write friendly space scale with system size and make adopting new standards harder. Larger device sizes require gigabytes of random-write space. In the case of F2FS, metadata structures and the checkpoint region grow with the size of the disk. We observed that at 1 TB, the size of these structures goes beyond 4 GB, the amount of random-write space on the 8 TB Western Digital ZN540 drive, necessitating additional drives to support total capacity of the drive. Such cross-device mappings can lead to performance and stability issues. To unlock the full potential of zoned SSDs in reducing wear and tail latency, we need an append-only filesystem that does not depend on any fixed addresses (which need in-place updates) or random updates. Such a filesystem should partition data by lifetime, performing garbage collection as required while maintaining POSIX compatibility to avoid application rewrites.

4.1 Design tradeoffs: copy-on-write vs. multi-log append

To achieve our goal of a ZNS-first filesystem, we first looked at desirable properties in a filesystem from a ZNS perspective. An append-only log-structured filesystem is well suited for such an interface [46]. We looked at two append-only update design families: log-structured merge trees and a copy-on-write design. To explore these, we chose F2FS and btrfs as they are flash-optimized, append-only, open source, and POSIX compliant. btrfs is a Copy-on-Write filesystem, organized as a B-Tree, while F2FS uses up to six separate append-only logs.

We then ran a simple sequential insert workload to measure the performance and utilization of each of these approaches. We expected btrfs, with its Copy-on-Write nature, to generate large amounts of data invalidated with each RocksDB compaction. To test out this theory, we set up a 32 GB emulated SSD with btrfs and F2FS. We then wrote 100 million records of 10 Bs, each with compression writing 6 GiB to each filesystem. The same workload yielded the following utilization results:

Table 4.1: Comparison of btrfs and F2FS on a 32 GB SSD.

	Zones		Capacity		
	Used	Full	Used	Invalid	Total Writes
F2FS	10	5	6.92 GB	0.80 GB	27.87 GB
btrfs	25	12	18.35 GB	12.15 GB	23.00 GB

As seen in Table 4.1, F2FS performs more writes than btrfs, mainly due to the relocation of invalid data on garbage collection. On the other hand, btrfs has valid data scattered across 25 zones and cannot garbage collect them due to the limited availability of free zones. With its 6 logs of varying temperatures, F2FS can better identify temporary files and garbage collect them together to free up zones. btrfs, on the other hand, has a single log with all the writes, making it harder to collect garbage.

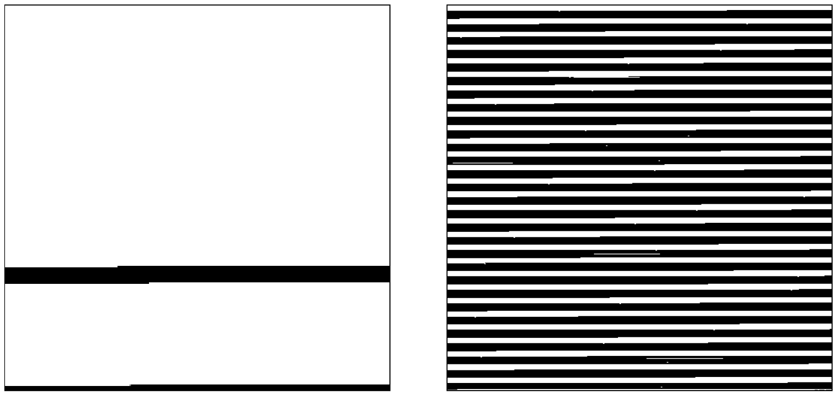
(a) btrfs (Conventional) (b) btrfs (ZNS)

Figure 4.1: We compare "full" block groups on btrfs on a conventional SSD with ZNS; white: valid data, black: unused space. On ZNS btrfs sees half-filled extents and poor utilization. An optimized filesystem will have a block that has no unused space— no black bands.

Further, btrfs' B-Tree structure with Copy-on-Write generates new data on every write, writing it to statically sized pre-allocated chunks, causing a large number of partially filled chunks that require garbage collection to reclaim. Unlike a traditional SSD, these empty regions cannot be written to on an append-only device, causing write and space amplification. For instance, in Fig. 4.1, we analyze a full block group from this experiment to a full block group on a traditional SSD and see that 60% of it does not contain valid data, resulting in poor utilization. Further, the balancing process is expensive in terms of writing and hurts the performance and the lifetime of the drive. In addition, as we see in Fig. 4.8, F2FS performs better than btrfs as its multi-log and multi-append point design enables greater parallelism and simpler grouping on a zoned device. The performance and layout benefits inspired Persimmon to inherit F2FS' data segment design.

However, F2FS metadata still expects a traditional in-place update-friendly interface and fixed locations. F2FS optimizes itself for SSDs by maintaining six logs for appending writes. Three logs are used for node data in the 'main area' while the other three keep filesystem data. Node data includes any block of information required to address other blocks, such as inodes or indirect blocks. Filesystem data makes up files or directories. These groups maintain separate logs for hot, warm, and cold data. Separating this data can extend the lifetime of a device by reducing write amplification.

F2FS supports ZNS for the data region [36], but not for checkpoint and metadata region which require fixed addresses and an in-place updates. This can be achieved either by:

① Dedicated *conventional* zones with in-place updates.

② Multi-device setup

③ In-place updates virtualized by a device-mapper

④ A device mapper (`dm-zoned`) which can emulate in-place updates

These options are not particularly desirable as they maintain all the limitations of conventional SSDs (although at a smaller scale) and add complexity to the architecture, and impact performance. Further they have their own limitations, `dm-zoned` for instance, requires zone capacity to match zone size, which is not the case on *available Zoned SSDs* as they use a power-of-two zone size to simplify addressing, even if the usable capacity is smaller.

4.2 Limitations of F2FS on zoned storage

Before we change F2FS, it is important to understand F2FS architecture to understand which parts use in-place updates and fixed addresses.

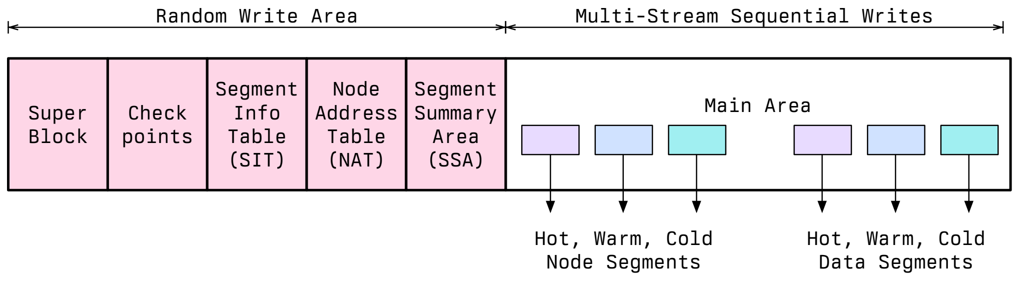
Random Write Area
Multi-Stream Sequential Writes
Super Block
Check points
Segment Info Table (SIT)
Node Address Table (NAT)
Segment Summary Area (SSA)
Main Area
Hot, Warm, Cold Node Segments
Hot, Warm, Cold Data Segments

Figure 4.2: F2FS consists of a random-write area for metadata and six temperature-separated logs for node and data.

Random Write Requirements. As we see in Fig. 4.2, F2FS metadata that requires fixed LBAs and hence in-place updates for maintaining the following structures:

① The *Segment Information Table (SIT)* stores the number of valid blocks in the log and a bitmap for their validity.

② The *Node Address Table (NAT)* provides a mapping of node ids and associated logical block addresses in F2FS.

③ The *Segment Summary Area (SSA)* stores owner information about blocks present in the logs.

While these requirements are relatively small compared to the data region, on large SSDs they require several gigabytes of space and see significant write activity. Since F2FS allows multiple devices, one of the proposed solutions is to use a dedicated random-write-friendly device, however this requires another storage device with matching block and sector sizes which might be difficult especially as F2FS is geared towards smartphone workloads.

Checkpoints generally store all these structures, using a segment (typically 2 MiB), with the exception of the SSA, these structures can grow up to several megabytes. The SSA on the other hand can grow up to gigabytes in size as it maintains metadata on every single 2 MiB segment. These structures see writes with every change in metadata and without media that can support in-place updates, they can quickly outgrow the limited random-write area available on ZNS SSDs. For instance, on the 8 TB ZN540 SSD, the maximum size of F2FS supported is 1 TB.

Checkpoint Design. Checkpoints in F2FS rely on a known address for the head of the pointer, and work in two packs, writing to them one at a time. The checkpoints themselves are append-only and we can easily modify the design not to rely on specific addresses as discussed in § 4.4.

Space Utilization. Over-provisioning in F2FS to allow garbage collection does not take into account the size of the zone, relying on capacity. As these can vary dramatically, it allocates much higher space for garbage collection. For instance, on the ZN540 with 2 GB capacity and 1.2 GB usable space, F2FS reserved more than half the usable space for garbage collection. If these calculations were tuned for ZNS, it can have unexpected impact on conventional SSD space reservation.

While F2FS is a good demonstration for the benefits of the ZNS interface, it falls short by requiring a rather large random-write area, as well as by inefficiently utilizing the available space. We decided to base persimmon on F2FS, redesigning the parts that require in-place updates to be append-only. We started with the data section of F2FS with its helpful temperature-separated multi-log design and replaced the metadata design to be append-only. In Persimmon, we eliminate the random writes and in-place updates associated with metadata and checkpoints. We introduce a new logic for over-provisioning and new in-memory structures to aid zone allocation and cleanup, reducing garbage collection overhead.

4.3 Persimmon design

While the random-write structures discussed in § 4.2 are valuable, they do not necessarily need in-place updates. Persimmon maps the SIT, NAT, and SSA block addresses to physical addresses on the drive, which allows updates virtualizing an in-place update interface. These are stored in dedicated zones as they see frequent updates and can be garbage collected with a low data movement cost. We implemented this mapping with Persimmon's page cache implementation.

Persimmon's page cache operation modifies the `address_space_operations` function table to minimize the change required in the F2FS code base. The Linux kernel allows these function tables as an interface for adding modified behavior for pages in a filesystem context. When grabbing pages, we switch out the `address_space_operations` object without affecting other filesystem operations. However, simply remapping the metadata is insufficient as it can add performance overhead to operations in Persimmon. To address the slowdown in access and cleanup, we introduce new in-memory structures which offer speedups for zone and metadata operations.

4.3.1 Eliminating in-place updates

We introduce three additional data structures, as seen in Fig. 4.3 to manage the metadata mapping, optimize lookup, and to aid in allocation and garbage collection:

Metadata Address Table (MAT). The MAT is an array that associates metadata block indices with block addresses. The F2FS metadata exists in a contiguous address range, with the last SIT address adjacent to the first NAT address. We offset indices by the first block address (the first SIT address) to reduce memory overhead before queries. The MAT size is the difference between the largest and the smallest block addresses.

Zone Information Table (ZIT). The ZIT is an array that maps a zone index to a count of the number of invalid blocks within that zone. The ZIT allows for speeding up garbage collection. Calculating the number of invalid blocks for a given zone requires iterating through each zone. This lookup table is updated if the address for a block does not match the one under iteration, incrementing the count.

Zone Bitmap (ZB). ZB maintains important zone data in memory to speed up new zone allocation. Persimmon reads the associated entry from the ZB instead of issuing a zone management command for the zones, avoiding the penalty of a zone management command to the device.

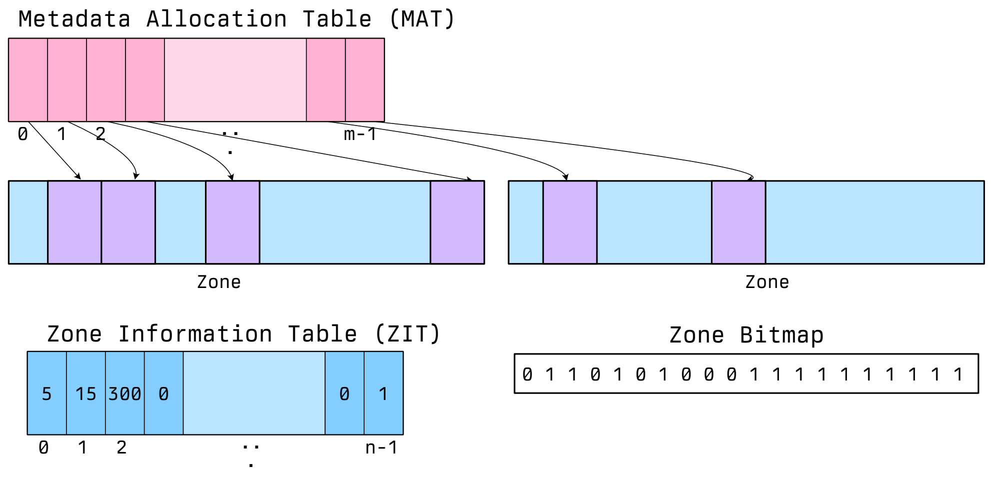

Figure 4.3: The three structures that reside in memory: the MAT , the ZIT, and the ZB. Here m is the number of pages in metadata and n is the number of zones.

Persisting in-memory Structures. To repopulate these in-memory structures on `mount` and for crash recovery, we must persist them to the drive. We split the metadata into page-sized chunks, which we append to the dedicated mapped metadata zones. In particular, we divide the MAT into chunks, and its in-memory handle maintains the addresses of each chunk. The MAT chunks provide direct addressing to blocks. Alongside the MAT chunks, we update the number of invalid blocks for a previous zone based on block updates. The full ZB is stored at the end of each chunk because its size is relatively small compared to the other structures. In systems with a large number of zones, we may need to split up the ZB as well.

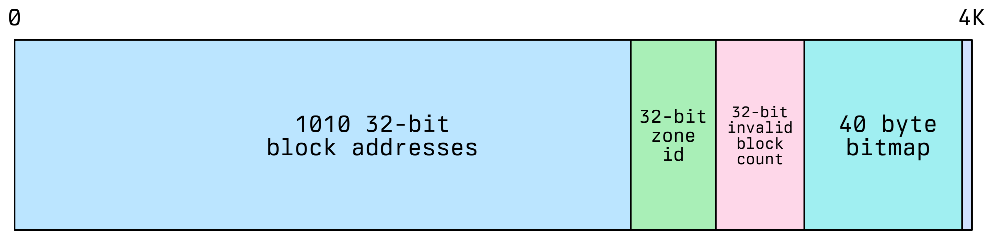

Figure 4.4: The page-aligned chunk design.

A chunk comprises 1010 32-bit MAT entries (the changes since the last chunk), a 32-bit zone id, a 32-bit invalid count, and a 40-byte bitmap as seen in Fig. 4.4. These data structures get appended as a whole to the checkpoints because of how infrequently the system generates checkpoints and the small checkpoint size (no more than a few blocks). We store the chunks adjacent to the Persimmon metadata and update the metadata and checkpoints to track them. This design is similar to the inode updates in LFS[49]. We utilize the Linux page cache for metadata and chunk blocks. To reduce the amount of data we need to write, we batch metadata updates in a `bio struct`, then write all dirty chunk pages in the same batch.

The NAT, SIT, and SSA are at least a few megabytes in size, with the SSA approaching tens of gigabytes at F2FS's maximum supported capacity. These structures are frequently updated, with an updated NAT with every data change. Unfortunately, this increases the size of the on-drive footprint for the metadata and consequently the MAT. However, since these structures are in dedicated zones and are frequently updated, space can be reclaimed at relatively low cost.

4.3.2 Allocating new zones

When a metadata zone fills up, Persimmon chooses another to continue accepting writes. The ZB is responsible for providing this information. The filesystem iterates through the list of zones until it finds the next free zone. We iterate through zones numerically, consulting the ZB until it finds the first free zone. We chose this approach over defining a queue of zones, assuming there will be few zones to search and NVMe commands can report empty zones. We maintain F2FS' maximum supported size: 16 TB. However, we run our tests on smaller drives, so we define the bitmap in our tests as a fixed size of 40 bytes since this is enough to address 320 zones, which would be sufficient for metadata. We allow users to expand ZB size based on the number of zones in the `makefs` tools.

4.3.3 Garbage collection

We added a cleaning procedure to release blocks of data that were made invalid by updates. This procedure uses the ZIT to inform its decision when choosing a zone for garbage collection to minimize the number of writes incurred from migrating valid data. We trigger garbage collection whenever the number of free zones is equal to 2, allowing for space to write to another zone where necessary. This trigger is configurable and can be decided by the performance and capacity requirements of the system. Once a zone with the most invalid blocks is selected, the validity of blocks is determined by querying for the associated mapped logical block address. The block is copied to a new open zone if a match exists.

4.3.4 Storage geometry

For device geometry compatibility, Persimmon defines a few different data size categories for managing blocks. Blocks in F2FS are all 4096 bytes. Segments are groups of blocks, and sections are groups of segments. 'Zones' are groups of sections, not to be confused with the ZNS zones. These options allow the host to align data structure size with the erasure block units of the underlying device and avoid the write amplification penalty.

To accommodate the sequentially written checkpoints, the size of the space allocated to the checkpoint is now two full zones, which allows for two individual checkpoints to be written alternately. The size originally allocated for non-checkpoint metadata is increased by 20% to provide additional space for persisting additional in-memory structures and so that enough space is available for garbage collection. Beyond that, the layout of Persimmon remains similar to F2FS. Additional fields were added to the super block in order to consider mapped addresses and the underlying block address that are mapped to. The drive layout uses two checkpoint zones, and allocates metadata and data zones as needed.

4.3.5 Repopulating in-memory structures on boot

A strategy similar to the *fysnc* read-ahead utilized in F2FS also inspires the model for persistence. Upon mount, like F2FS, the most recently written zone is read up to the write pointer, updating the in-memory structures where necessary. This involves reading every other block because the chunked data is in adjacent blocks. When we read a chunk block, we update the block address for the respective chunk in the in-memory MAT, the invalid count for the associated zone in the ZIT, and the ZB. The data stored in the chunks represent states idempotent as a whole, and not incremental updates that require previous chunks to be applied first, nor will the system's state become invalid due to repeated applications of the chunk data. This design ensures a valid state in the event of a power outage.

4.4 Append-only checkpoints

Much like metadata, F2FS checkpoints depend on a known location, and are updated in place. Since in-place updates are not supported in an append-only design, we redesigned the checkpoints to be append-only, utilizing a pair of zones that are garbage collected once they are full. For resetting zones, we adopted a policy similar to btrfs [48], where Persimmon resets zones after writing the most recent data to another zone. Persimmon's checkpoints include additional fields for persisting metadata and in-memory structures to the checkpoint. However, F2FS checkpoints still depend on an on-device known location for checkpoint headers which can get erased with a zone reset. Since the zones maintain a write pointer pointing to the log head, we use the write pointer in the checkpoint zones as a known location for the checkpoint. We ensure that the footer maintains mappings to the rest of the checkpoint, using the write pointer to read the footer, giving us access to the checkpoint. Since the footer is a fixed size directly preceding the write pointer address, this simplifies keeping track of checkpoints.

Reclaiming space for a checkpoint can occur before writing a checkpoint since the previous valid state is located in the previously-written zone. This approach differs from btrfs; checkpoint writes do not spill over into the other zone, maintaining their separation and eliminating the need for data movement on garbage collection.

4.5 Limitations

Since Persimmon is a fork of F2FS, we inherit some of the limitations already present in F2FS, namely, a maximum filesystem size of 16TB due to 32-bit logical block addresses. Additionally, Persimmon only works on zoned SSDs. We add a small amount of write and memory overhead with the new metadata structures;

however, even with these additions, we reduce the overall amount of writes with better garbage collection and simplified device structure. Further, as Persimmon works on zone granularity, it allocates more on-device real-estate toward checkpoints and metadata relative to the size of these tables, especially on systems with larger zones. We plan to explore techniques to reduce this space overhead.

4.6 Evaluation

We worked on Persimmon with the goal to create a filesystem with append-only structures that is able to utilize the zoned interface to its full potential, reduce garbage collection overhead, improving tail latency, device utilization, and reducing write amplification. We demonstrate the small overhead of Persimmon in terms of memory usage and mount times.

For evaluation, we use a 64 Core AMD EPYC server with 256 GB of DRAM. We use dedicated Western Digital ZN540 zoned drives for each filesystem to ensure strict performance isolation and use conventional zones on the same drive in case of F2FS to ensure that cross-device coordination does not impact its performance. We write to empty regions of flash to ensure that no on-device garbage collection occurs unrelated to the benchmark. We capture statistics both at the device level and the filesystem level to measure writes. We use Yahoo Cloud Serving Benchmark [19] with RocksDB and RocksDB's built-in tool `db_bench`.

4.6.1 Improvement in tail latency

We tests the impact on latency with YCSB's[19] 6 workloads: namely: A. Update-heavy workload: 50% reads, 50% updates, B. Read-heavy: 95% reads, 5% updates, C. Read-only, D. Read-Insert: 5% records are inserted, and latest records are read, E. Short ranges: 5% records are inserted and scanned over short ranges, F. Read-Modify-Write: 50% Reads and 50% Read-Modify-Writes

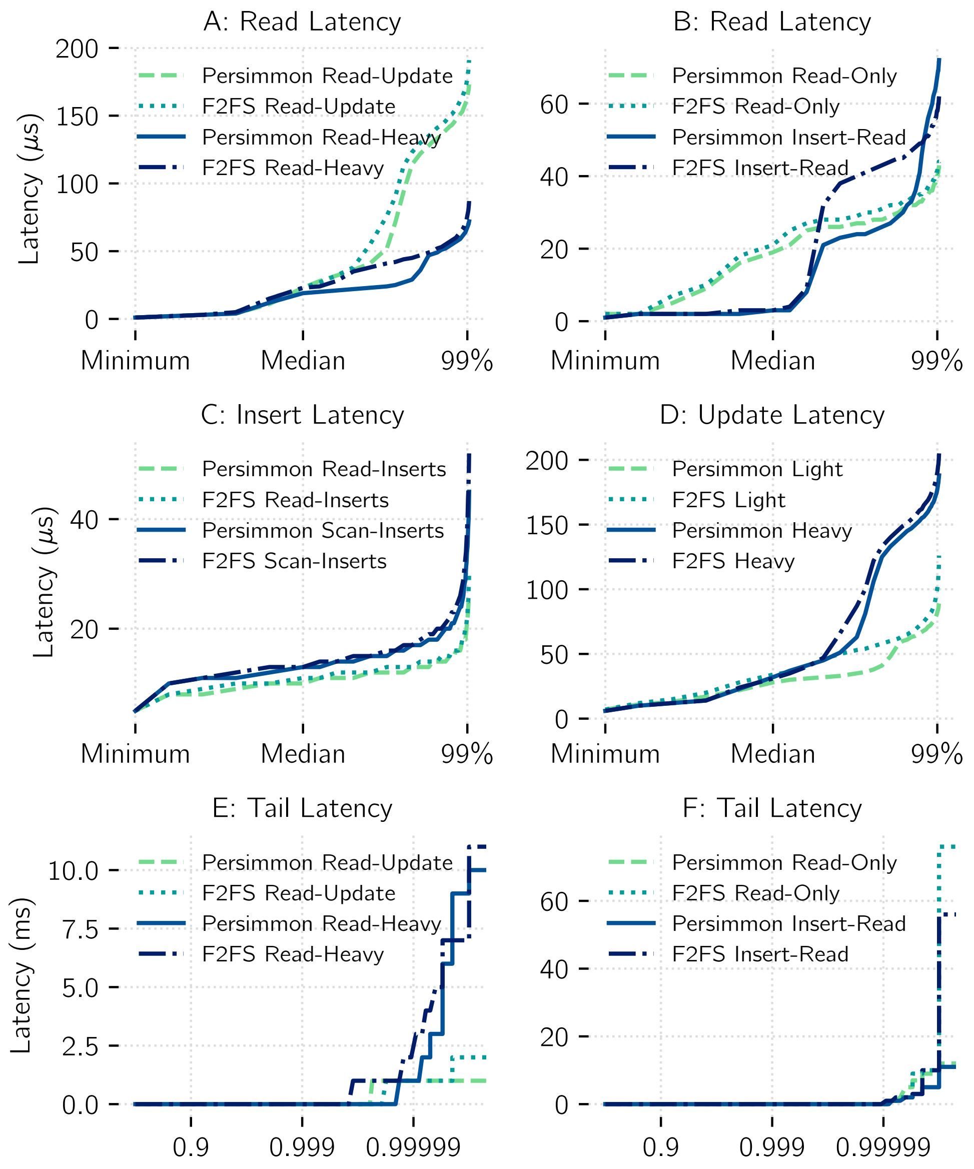

Figure 4.5: For Read-Latency, Persimmon performs similar to F2FS offering better tail latency.

For read latency as seen in Fig. 4.5 Persimmon performs uniformly better, minimizing latency for every workload, especially tail latency (E and F). We get this advantage due to the faster lookup operations enabled by the in-memory MAT and minimizing tail latency spikes usually caused by background operations. For the Read-Insert workload, the persistence of recent inserts in the metadata structures hurts Persimmon at the tail. Zoned F2FS, on the other hand, see millisecond-to-second spikes in tail latency, especially during read-heavy workloads as the device tries to free up space and garbage collection. As the filesystem fills up, efficiently collecting garbage becomes essential, and Persimmon cuts down on garbage collection in the filesystem.

4.6.2 Efficient garbage collection

We first ensured that each device had the same size to measure garbage collection efficiency. Then we ran a multi-threaded benchmark, which filled the drive with 55 GB sequential inserts. This is followed by random updates, deletes and overwrites, and finally by random reads. This is the same benchmark that we used to measure throughput and write amplification. Effectively these actions yielded more than 200 GB of writes on the 128 GB drives ensuring garbage collection.

We measured the number of pages moved by the device in the two garbage collection modes:

① **Background:** F2FS and Persimmon's age-sorted background mode performs lazy garbage collection.

② **Foreground:** When space is running out the filesystems can aggressively free up zones.

As seen in Fig. 4.6, during the benchmark, Persimmon does not need to move large amounts of pages to perform background garbage collection since Persimmon can reset metadata and checkpoint zones without data movement as the data

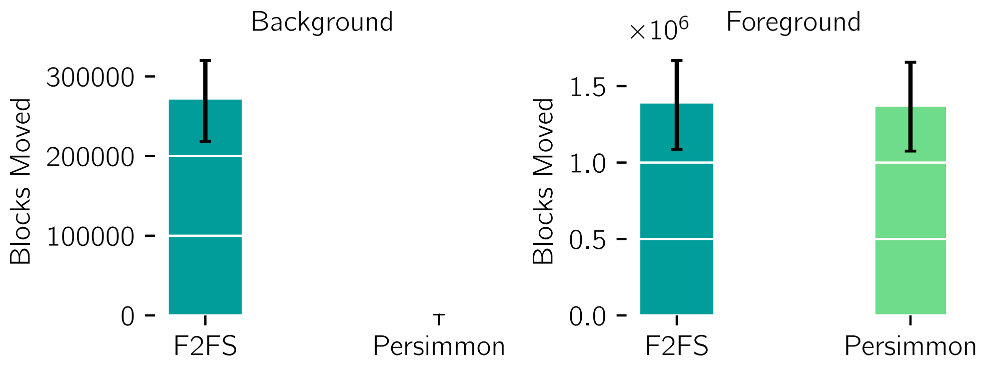

Figure 4.6: Persimmon performs virtually no garbage collection in the background as opposed to F2FS.

has a short lifetime. Persimmon's greater availability of free space further helps it wait longer to collect garbage, waiting for more data to be deleted over F2FS, resulting in lower data movement. F2FS needs to perform more background garbage collection to ensure free space. On the other hand, both systems perform similar amounts of foreground garbage collection, as this is workload dependent and in the data region.

Table 4.2: Persimmon can free up space without overhead due to better grouping of data

	Calls	**Data Moved**	**Read Tail Latency**
F2FS	12 ± 5	1050 ± 229 MiB	76.73ms
Persimmon	1 ± 0	0 ± 0 MiB	12.41ms

For garbage collection efficiency in Table 4.2, we look at the statistics maintained by the filesystem. Persimmon manages to garbage collect more efficiently, needing just one call for regular workloads as opposed to F2FS's 12, and reduced number of foreground calls. As each call negatively impacts throughput, write amplification and latency, we see modest improvements due to this reduction in overhead. In our workloads Persimmon moved no data in the background while maintaining more free space compared to F2FS moving almost 1 GiB.

4.6.3 Write and space amplification

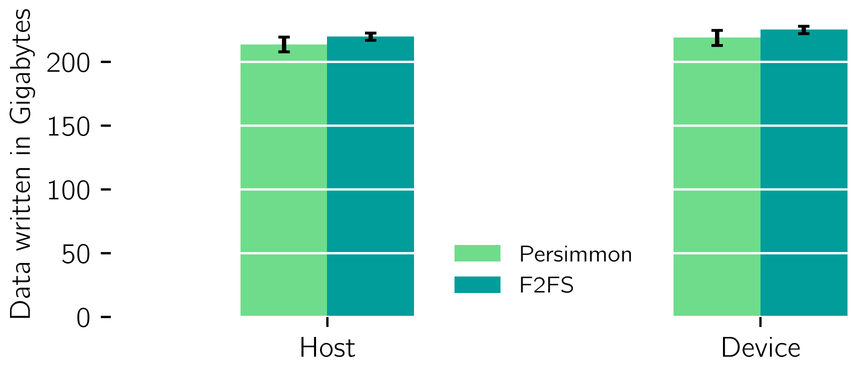

Figure 4.7: Persimmon reduces writes both to the device and the filesystem

Table 4.3: Usable space with 10% over-provisioning and 2 GB zones

	F2FS (GB)		Persimmon (GB)	
Size	Usable	Reserved	Usable	Reserved
64 GB	33	31	47	17
128 GB	93	33	107	21
256 GB	215	41	229	27

Persimmon reduces the total writes from the benchmark by about 4 GB, plus the writes saved in avoiding the garbage collection, which the metadata drive for F2FS will have to perform. As seen in Fig. 4.7, we measure both the writes reported by the filesystem and those reported by SMART data on the device. In our tests, on an average Persimmon writes 214.5 GB to the filesystem, resulting in 219.7 GB on the device, where F2FS 218.4 GB of writes on an average rising up to 223.6 GB on the SSD. While it is not a large difference, all of these writes are due to the extra garbage collection of metadata and F2FS needs to perform. Further, Persimmon offers a larger usable space, especially at smaller size, over F2FS as seen in Table 4.3, with metadata overhead shrinking with the filesystem size.

4.6.4 Throughput

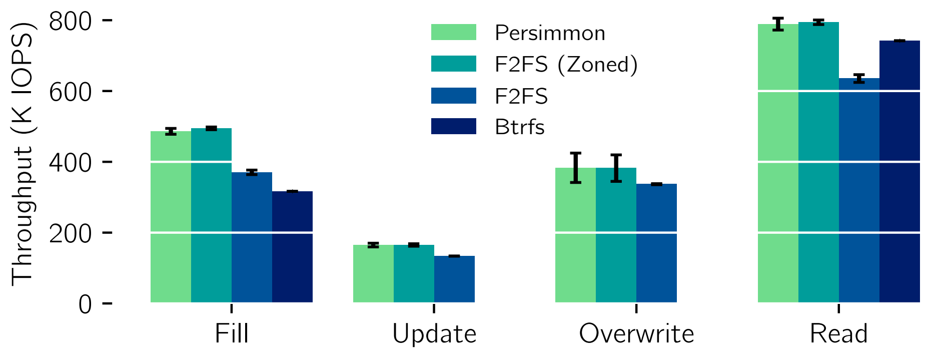

Figure 4.8: Persimmon sees a slight slowdown in sequential insert, but is otherwise similar to F2FS.

While we do not expect significant improvements in the flash-optimized F2FS, as seen in Fig. 4.8, we see similar performance to F2FS in Persimmon, with a slight degradation in the read workload, due to additional lookup overhead. Performance is better than non-zoned F2FS and btrfs, however, we were unable to test btrfs's update or overwrite performance due to stability issues in zoned mode.

4.6.5 Overhead

Mounting and unmounting. As Persimmon introduces new in-memory structures we expect to observe some overhead in terms of mount times and memory usage. However, our approach is faster at mounting than F2FS, as the single-device single-namespace layout is faster to read. The time to mount grows linearly with size as seen in Table 4.4. For unmount, Persimmon takes a small performance hit, due to the need to persist extra structures, but the overhead is insignificant.

Memory overhead. With the addition of new in-memory structures, we see a modest overhead in the memory utilization, particularly under heavy workloads, however this overhead is about 20 MB per each 100 GB added to the filesystem size

Table 4.4: Mount and unmount times (Seconds)

	F2FS		Persimmon	
Size	mount (s)	unmount (s)	mount (s)	unmount (s)
256 GB	0.2185	0.016	0.149	0.02
512 GB	0.327	0.018	0.2545	0.022
2048 GB	–	–	0.8784	0.026

as seen Fig. 4.9. Since this could be an issue at larger SSD sizes we provide an option to limit this overhead with an `ioctl`, however, lowering the memory use may impact performance.

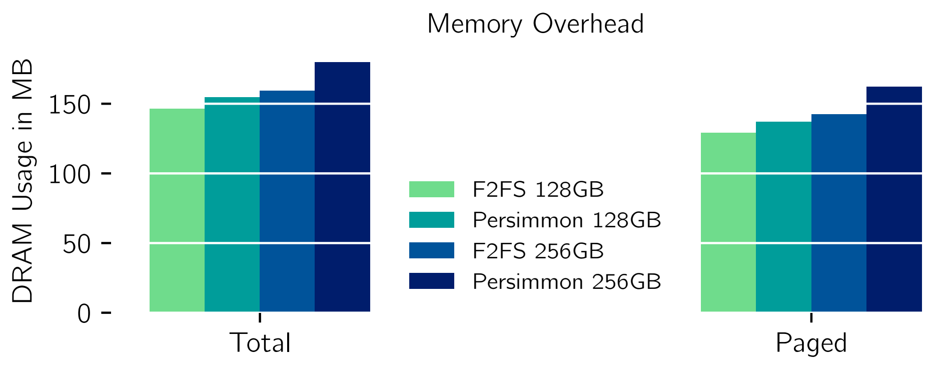

Figure 4.9: We observe a slight uptick in memory use depending on the workload, however the overhead is fairly small.

4.7 Reflections on Persimmon's approach

4.7.1 When compatibility is not enough

Adopting new hardware interfaces like ZNS requires changes in fundamental abstractions that applications rely on. Rewriting applications is expensive, and it is easier to modify the filesystem to accommodate the benefits offered by these new interfaces. However, changing existing filesystems without breaking backward compatibility is a complex task that results in compromises, as we see with F2FS, requiring a random-write area for metadata and excess over-provisioning to accommodate the new interface as seen in Table 4.3.

4.7.2 Complexity inherent to kernel filesystems

While future versions of these filesystems may address these issues, updates depend on Linux kernel releases which may take years for widespread adoption. Upgrading the filesystem structure requires major changes to on-drive data and metadata layout and can cause compatibility or stability issues. These issues hurt the adoption of modern storage protocols like ZNS and FDP since rewriting applications is expensive, and filesystems do not support the new standards effectively. However, the interface has much potential beyond the benefits we demonstrate with Persimmon. For instance, these filesystems' hint and grouping mechanisms are rudimentary and cannot generate workload or architecture-based hints.

4.7.3 Hint generation

An essential property of ZNS that we exploit is the ability to group data with related lifetimes, allowing us low-cost zone resets. The F2FS temperature separation could be tuned further for file lifetimes rather than workloads, grouping files with related lifetimes. The naive hints can be improved with approaches that use active learning systems and tune according to topology and workload. Persimmon uses a `fadvise()` system-call-based hint mechanism for deciding which files end up on which log. An external service or the application could provide better directives to Persimmon than F2FS' heuristic-based approach.

4.7.4 The need for user space approaches

Since filesystems are in the kernel, changes require updating the kernel module, migrating the old structures to the new logic, unmount, reformat, and remounts. Errors can cause a kernel panics, data loss, and system crashes, further complicating the adoption of novel filesystems. An easier way could be to use a simple filesystem, like ZoneFS, which maps the block layer to a file interface. POSIX compatibility and placement logic could then be implemented as a shim layer.

4.7.5 Kernel bypass

Another way could be to use frameworks like xNVMe [39] or SPDK [70] to bypass the kernel, implementing all logic in userspace. However, this approach adds complexity to the storage logic.

4.7.6 Other NVMe protocols

NVMe FDP has been proposed to address the issues with the append-only nature of ZNS, allowing grouping based on hints in a random-write friendly interface. With key-value SSDs, computational storage, and memory-semantic SSDs, interfaces to SSDs are getting more diverse, and the adoption of heterogeneous hardware becomes even more challenging. A filesystem cannot reasonably support different types of devices and tune each application according to the workload. We must rethink our approach to these SSD protocols and filesystems to build data centers with diverse storage types.

⁂

Summary

With NAND-flash-based SSDs becoming ubiquitous, we need to scale them without severely impacting the lifetime or performance of the drives. ZNS is an effort to enable that. However, it has not seen its full potential explored due to the complexity of modern filesystems, manufacturer-specific extensions, and the need to preserve backward compatibility. With even log-structured systems requiring random writes and in-place updates, traditional systems see various usability issues on append-only SSDs. We present Persimmon, an append-only filesystem for ZNS SSDs that improves on F2FS in metadata management, garbage collection, device utilization, and tail latency. But perhaps the biggest benefit of Persimmon is that it needs no auxiliary devices, and can be used on an append-only interface. We compare multiple approaches and demonstrate the utility of embracing an append-only interface, lowering tail latency, improving performance, reducing the amount of data written, and simplifying garbage collection. We plan to open source Persimmon to get community feedback and contributions.

5
SHIMMER

Thirty spokes, meet in the hub. Where the wheel isn't is where it's useful Hollowed out, clay makes a pot. Where the pot's not is where it's useful. Cut doors and windows to make a room. Where the room isn't, there's room for you. So the profit in what is is in the use of what isn't.

—Tao Te Ching (Ursula K LeGuin Translation)

Synopsis

We present Shimmer, an efficient shim layer that enables host-device hint communication with *no modifications* to the system or the application. Shimmer allows applications to adapt to changing hardware interfaces with low effort and generates hints to improve performance. We demonstrate the benefits of Shimmer by writing a lightweight shim layer for two widely used log-structured data management systems: RocksDB, Cachelib, and WiredTiger. Shimmer can enable quick adoption of different hint interfaces without rewriting applications. In this chapter, we will discuss the design on Shimmer along with implementation details of *shimmer-vfs*: a lightweight persistence layer. In Chapter 7, we will evaluate the performance of Shimmer.

⁂

Changing interfaces enforce constraints on device usage. ZNS' append-only interface means we must rewrite applications to avoid in-place updates. Similarly, FDP requires writes to be annotated with a placement group to be effective. While these approaches offer significant performance improvement, rewriting applications to utilize them is expensive, limiting their adoption.

Traditionally, the added complexity of hints and placement has been absorbed in the filesystem layer, which presents its own set of limitations as we saw in Chapter 3. Filesystems are unaware of application-level data characteristics unless specialized hint mechanisms are used. Even flash-optimized filesystems like F2FS see a 33% write throughput degradation over application-specific solutions like ZenFS [13] despite being provided appropriate write hints. Furthermore, changing hint interfaces and device interfaces adds complexity to the kernel and gives rise to bugs and security vulnerabilities.

Shimmer isolates the added complexity of hint generation in a separate *shim* layer, enabling the benefits of hint communication without the need for application or filesystem changes. Shimmer allows customized hint generation and data placement *without the cost of additional kernel crossings* by interposing on read-write calls, modifying them for the specific interface before sending them to the device through the Linux storage stack. Shimmer can provide hints to a hint-capable filesystem or with *shimmer-vfs*, a lightweight append-only filesystem implemented with Shimmer. With *shimmer-vfs*, we show that significant performance gains with a few lines of application-specific configuration.

5.1 Requirements for a good shim layer

For a shim layer to replace filesystems or application rewrites, we believe it should have the following properties:

① *No application change:* Application rewrites are expensive and can lead to bugs and performance regression.

② *No host change:* Installation of kernel modules can lead to security and stability issues.

③ *Easy to adopt:* Such a shim layer should be easier to deploy than an application or a filesystem.

④ *Efficient:* A shim layer should not add significant CPU or memory overhead.

⑤ *Multi-application:* Data centers often use other programs for performance monitoring, replication, and backups; such applications should have the same visibility.

⑥ *Effective:* Unlocks the performance benefits new storage architectures offer.

⑦ *Extensible:* The hint generation should be configurable, adding the ability to add custom logic, including systems that learn dynamically.

We built our systems to stay true to these principles, and we demonstrate that such a layer is not only feasible; it can outperform custom solutions.

5.1.1 Picking the fastest shim technique

While using a FUSE file system or eBPF-based approach is *flexible*, since hints can be provided by simply observing and intercepting system calls made by the target application, it suffers from performance degradation due to the increased utilization of the kernel, which also adds configuration and debugging complexity. Using these techniques, a hint generation engine deciding the hint for a new file may require several costly kernel crossings before it is able to modify the call to create the file with the right hint. On top of this, a file system or eBPF-based approach is that it cannot easily take advantage of application knowledge.

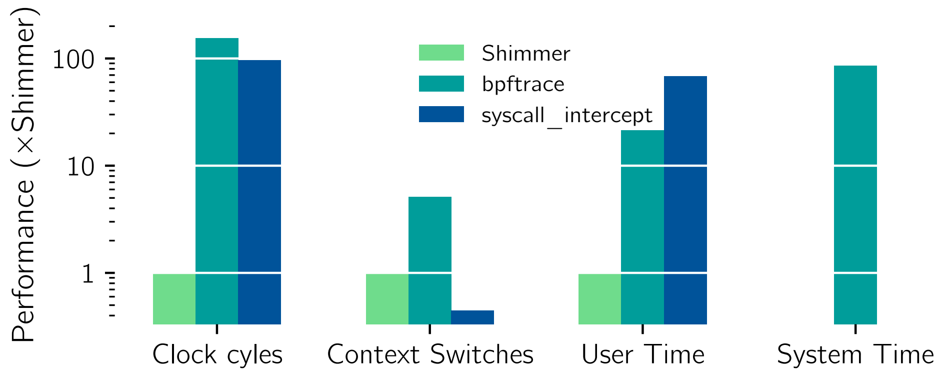

Figure 5.1: The overhead of intercepting write calls with Shimmer (described in Chapter 5), `syscall_intercept`, and `bpftrace`, normalized by the lowest value. We normalize `bpftrace`'s system time by Shimmer's total time

`LD_PRELOAD` is a mechanism employing the technique of *late-binding* that allows users to preload in shared objects to selectively override functions defined in other shared objects through dynamic linking. Simply put, `LD_PRELOAD` enables developers to override function calls made in an application without needing to modify the application itself. It also works entirely in user space, so it does not incur the performance overhead caused by unnecessary kernel utilization. We measured the performance of these different interposition techniques with a simple program that issues writes to multiple files. As seen in Figure 5.1, our approach uses a fraction of the CPU cycles of both `bpftrace` and `syscall_intercept` to perform the same write call interception. `syscall_intercept` needs to disassemble and patch the binary, and thus requires significantly more processing power, but saves on context switches over other approaches. `bpftrace` involves execution in the kernel, which performs significantly worse than userspace approaches. Although the `LD_PRELOAD` approach works out of the box for stores like RocksDB which call `libc` directly despite being statically linked internally, an approach like `syscall_intercept` may be needed for applications which use indirect `libc` functions or statically link `libc`, both of which cannot be intercepted using this technique.

```
unsafe fn write(&mut self,
fd: c_int,
buf: *const c_char,
nbytes: size_t) -> c_int {
...
 if filename.contains(".log") {
     WALFILES.insert(fd);
     // 0x40c is F_SET_RW_HINT
     let _ret = libc::fcntl(fd, 0x40c, &3);
 }
 if filename.contains(".sst") &&
flags as i32 & libc::O_ACCMODE != libc::O_RDONLY {
     SSTFILES.insert(fd);
     let _ret = libc::fcntl(fd, 0x40c, &5);

 }
 ...
}
```

Figure 5.2: Overloading a `write()` call with `LD_PRELOAD`

5.2 Shimmer design

Shimmer is a dynamic library that allows interposition on file read-write calls, modifying them, if necessary, to redirect them to different regions on an SSD. Shimmer allows the configuration of rules to generate hints for an application, leveraging user insight to generate hints and place data. As Shimmer has insight into the application and the storage architecture, it can generate more effective hints than an application or a filesystem. Further, users can modify the hint generation easily without having to rewrite, reconfigure, or recompile the application or the filesystem.

Work on shim layers has had a renaissance recently, with fast-changing hardware and modern hierarchies like CXL requiring transparent porting of applications. Techniques include systems such as eBPF [11], FUSE [43], WASM [1], virtual machines, and dynamic libraries [21] can be used to implement shim layers. We focused on simplicity and performance, so we chose a dynamic library approach using `LD_PRELOAD` as it involves no additional kernel crossings and is all userspace. FUSE and eBPF require additional kernel crossings, limiting the performance gains, while WASM adds overhead to the runtime.

While kernel hint interfaces are unavailable for ZNS and FDP, systems like ZenFS and F2FS support the `RWH_LIFETIME_HINT` from the Stream SSD protocol on ZNS SSDs. Shimmer introduces its own hint protocol, as discussed in Fig. 6.1, which it can map to such hint formats even if the application does not implement them. While currently hint support is limited to F2FS and ZenFS, we implement a lightweight virtual filesystem layer (*shimmer-vfs*) to demonstrate what kind of performance a filesystem designed for hints could achieve. For flexibility, Shimmer offers two modes:

① *Hint-only Mode:* Shimmer provides an approximation of its grouping in the form of stream kernel hints – short, medium, long, or extreme.

② *Shimmer-vfs Mode:* Shimmer modifies the data path to remap files onto a ZoneFS [44] device acting as a user-space filesystem.

The *hint-only* approach offers a lower overhead of adoption and could be tuned if filesystems offer a better interface for hints in the future. It acts without changing the data path, ensuring that applications not linked with Shimmer can access the data. The shimmer-vfs mode unlocks the full potential of an interface like ZNS, offering improved throughput and lower latency. In this mode, Shimmer acts analogous to a filesystem and remaps files to specific zones. In this mode, applications must be pre-loaded at runtime with Shimmer to access remapped data.

To demonstrate Shimmer, we implemented it for the ZNS protocol and stream SSDs in the hint-only mode. While FDP SSDs are available, they lack kernel API support, routing directives through NVMe commands, which Shimmer could implement, but mapping system calls to NVMe commands would significantly in-

	Shimmer	**eBPF**	**`syscall_intercept`**	**FUSE**
Adds Kernel crossings	–	yes	–	yes
Modifies the Application	–	–	yes	–
Re-configures the kernel	–	yes	–	–

Table 5.1: We chose the `LD_PRELOAD` approach for Shimmer as it requires no change to the application, kernel, or special privileges, ensuring that data security and system stability is not affected.

crease the complexity of a shim layer. Further, as FDP SSDs are not yet widely available, we could not evaluate the effectiveness of such an approach. We plan to explore this in future work.

5.3 The Shimmer virtual filesystem (*shimmer-vfs*)

ZNS partitions the device into equal-sized append-only regions called zones. The host is responsible for garbage collection, which is done by performing the erase operation on a zone. ZNS SSDs require active management of write buffers. We demonstrate how a filesystem can effectively use such an interface by implementing a lightweight mapping layer *shimmer-vfs*, which maps files to zones.

For ZNS SSDs, we implement *shimmer-vfs* on top of ZoneFS, a block-layer representation of zones, a part of the Linux kernel. ZoneFS exposes each zone as an append-only file and does not implement write buffering, as it could violate append-only semantics. Reads are unchanged. We chose ZoneFS over raw device as it offers a simple 1–1 mapping for system calls, and adds almost no overhead over writing to raw hardware. *shimmer-vfs* maps application files to these zone files, implementing its own write buffering, and uses hints to decide grouping of files. Since ZoneFS is bundled with the Linux kernel, adopting Shimmer does not require large-scale system reconfiguration or downtime.

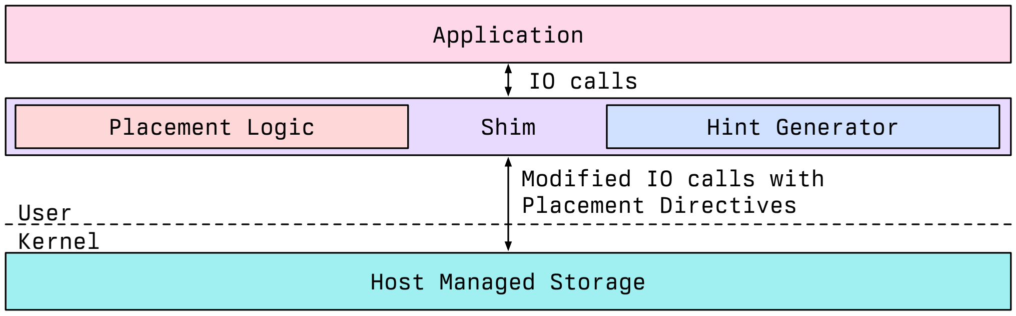

Figure 5.3: Simplified Shimmer architecture.

As seen in Fig. 5.3, Shimmer has 3 major components:

① **Interposition:** Interposes on I/O calls to modify them.

② **Mapping (*shimmer-vfs*):** Maps files to device groups.

③ **Hint Generation:** Decides which zone to map a file to.

5.3.1 Data layout in *shimmer-vfs*

Much like a filesystem, *shimmer-vfs*' main job is to map files to different regions on the SSD. To perform this, *shimmer-vfs* uses user-configurable fixed-sized extents, breaking each file down into extent-sized chunks and using stream information from the hint generation logic to map them to zones. Shimmer's buffer cache maintains a circular linked list of extent-sized buffers (one per writable file). We copy incoming writes to these buffers, and once they fill up, Shimmer notifies a background writer thread, which persists them to a specific zone based on the hint. Once persisted, the buffer is marked empty and reused.

Flash-optimized systems are log-structured, so we can significantly reduce the metadata overhead to locate these files. *shimmer-vfs* ensures files are stored in contiguous locations and hence needs to maintain only the zone ID, file size, and offset to locate a file. *shimmer-vfs* achieves this by maintaining two data structures: it maps filenames to unique identities in a lock-free hashmap called *NameSpace* to speed up lookup, and the identifiers are organized in another map, which

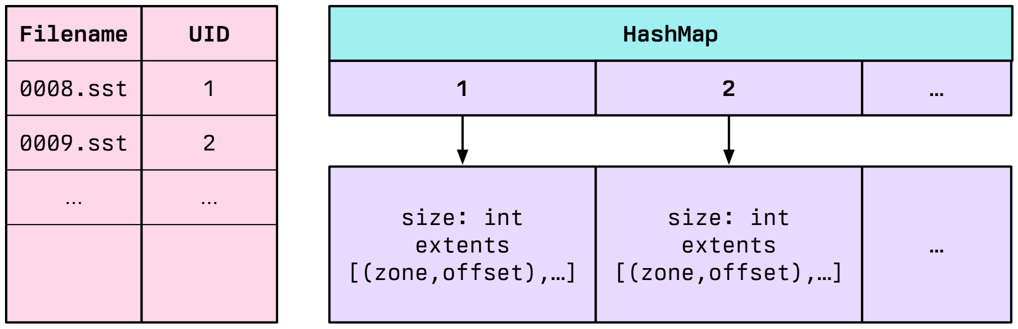

Figure 5.4: The mapping layer contains two data structures, one lock-free hashmap to speed up lookup translating paths into unique identifiers and another for maintaining the metadata of each file.

maps IDs to file offset and size as seen in Fig. 5.4. Shimmer uses `dashmap` [64] a lock-free hashmap based on associative arrays for most of its data structures.

Shimmer maintains an additional structure, the *ZoneMap*, to speed up zone allocation and cleanup, building a map of all available zones. For each zone, Shimmer maintains the capacity, current write offset, and invalid data size, packing these into a 128 bit structure per zone, indexed by zone number. This *ZoneMap* is maintained in a lock-free hashmap and persisted. Many of the performance gains for *shimmer-vfs* are from the lock-free data structures it uses internally, enabled by assumptions about the files, such as the append-only nature and immutability once written. These assumptions are valid for many log-structured systems, but if not followed, *shimmer-vfs* will need to pass through such files to the random-write filesystem.

5.3.2 A simplified buffer cache

The Linux kernel buffer cache does not guarantee the preservation of the order of writes and may reorder data before writing, which makes supporting data buffering complicated. Systems like ZenFS and F2FS utilize the `mq-deadline` scheduler to get around reordering partially; however, `mq-deadline` is not a particularly efficient scheduler, and the results are high lock contention [6]. Since *shimmer-vfs* can make certain assumptions about the nature of the workload that can simplify the design of such a cache, we decided to implement a user-space buffer cache.

To support breaking down larger files, *shimmer-vfs* implements *extents*, which breaks down files into equal-sized chunks. Users can pick an extent size based on the expected file size, but we restrict it to a power of 2 for addressing simplicity. Unlike filesystems, *shimmer-vfs* does not allow sharing extents, and buffers are padded to an extent size if not full. Since our target workloads make files immutable once written, we can afford to use up some extra space in this way to make metadata management easier.

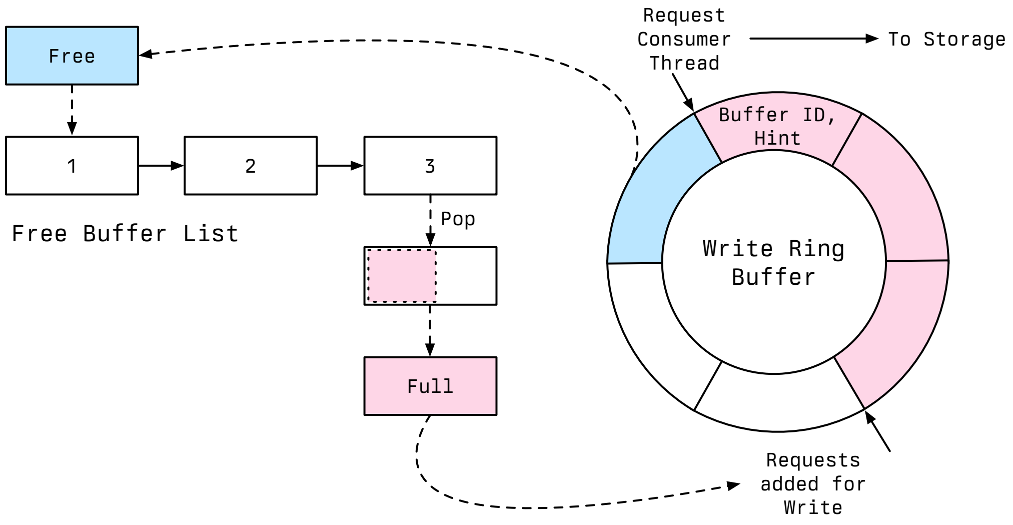

Figure 5.5: *shimmer-vfs* maintains extent-sized buffers in a linked list, a buffer extent is allocated to each writable file, and once it fills up, the information is sent to a background writer thread which consumes and frees it.

Free Buffer List. *shimmer-vfs* maintains a list of extent-sized memory regions that are assigned to writable files opened by the application. Each write is copied into the buffer, on `fsync()` or a full buffer, and the buffer is added as a write request (see Fig. 5.6) to a ring buffer that maintains write requests.

As seen in Fig. 5.5, a background thread goes through each write request, using the stream to place data. *shimmer-vfs* uses *allocate-on-flush* to simplify metadata

```
struct WriteRequest {
id: usize, file: usize, buffer: usize, stream: Stream,
}
```

Figure 5.6: A write request carries unique identifiers for the file, the buffer, and the stream.

management and pins the writer thread to a single core to minimize the performance impact of context switches. This task can also take care of necessary background actions, like closing zones once they are filled.

The buffer cache implements several optimizations for efficiency, including the use of `jemalloc` for thread-affinity, thread-per-core architecture to minimize context switches, and allocate-on-flush to avoid pre-allocating space in an append-only context.

5.3.3 Supporting tasks

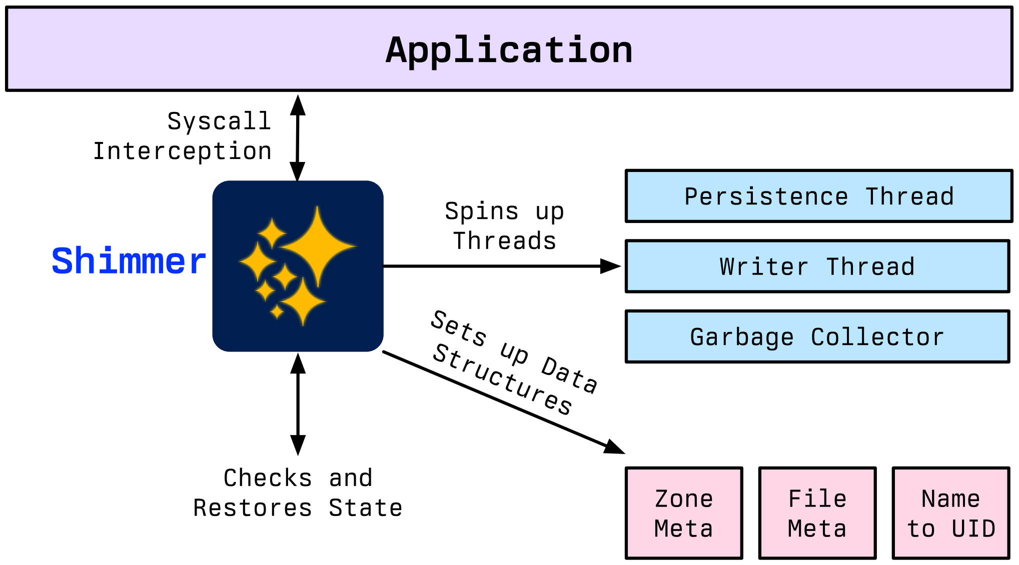

Figure 5.7: *shimmer-vfs* performs a number of tasks on the first run, this includes spinning up threads for maintenance tasks and populating data structures.

shimmer-vfs needs to perform certain tasks in the background such as persistence of metadata to make sure it can recover from crashes, and garbage collection.

As seen in Fig. 5.7, *shimmer-vfs* checks for persisted state, and spins up maintenance threads on first run.

Garbage Collection. *shimmer-vfs* implements lazy garbage collection, using the ZoneMap to track zones with the most invalid data, issuing an `ftruncate()` call when available zones are below a certain threshold (currently 4). Since log data becomes invalid relatively quickly *shimmer-vfs* always has zones it can free with minimal cost. If relocation is needed Shimmer can relocate the data to a new zone by issuing a Zone Copy command, eliminating the overhead of fetching the data into memory and writing it back on the device.

Open Zone Limit Management. On ZNS devices, since write buffer allocation is not dynamic, we need to actively track used resources and free them by finishing or closing a zone. Shimmer in hint-mode does not need to worry about this as it is the responsibility of the underlying filesystem; *shimmer-vfs*, on the other hand, maintains a count of open zones and closes them as they fill up. Since there can be a dozen or more open zones at a time, this limit is rarely reached and typically happens when running multiple applications simultaneously.

Coordination with other Shimmer applications. *shimmer-vfs* implements a lazy coordination technique with other instances, allocating completely free zones and checking the open-zone capacity offered by ZoneFS on `sysfs` to regulate its usage. More complex setups will require a special daemon to coordinate instances.

5.4 Interposition

`LD_PRELOAD` allows `libshimmer` to be loaded before other objects while executing a binary. Shimmer uses this functionality to selectively override various `libc` calls, using Shimmer calls instead for IO. In *shimmer-vfs* mode, Shimmer captures the content of calls the application issues, copying the data to per-file buffers and redirecting the call to the zone indicated by the hint generation. Such interposition makes up a bulk of Shimmer code, along with the logic to allow buffering and alignment to device block boundaries. This logic does not need to be modified per application unless the application behaves in some unexpected manner. For instance, truncating a file is not supported on an append-only log, and Shimmer will lie about a successful truncation if the application expects it.

As Shimmer can intercept any `libc` calls, *shimmer-vfs* can maintain the guarantees provided by any filesystem, such as data persistence on a successful `fsync()`. Since Shimmer does not have a runtime independent from the application, it leverages specific calls to maintain state, spin up background tasks for garbage collection, and persist its metadata. While mapped files are invisible to applications not pre-loaded with `libshimmer`, utilities such as backup or copy can also be preloaded with Shimmer, offering a filesystem-like alternative. On ZNS, *shimmer-vfs* only supports append-only files, which comprise a bulk of log-structured data storage, and any other randomly-written files, such as `LOCK` files, can be transparently passed on to the filesystem without interception by Shimmer.

5.4.1 Overloading `libc` calls

`open()`. Intercepting the file open call allows Shimmer to perform many setup actions that are helpful later; this includes assigning or fetching the right `uuid` for a file, allocating buffers for a writable file, opening the suitable zones for a read-only file among other tasks. `open()` is also where Shimmer checks for any persisted data about a previous session, populating its internal data structures. These extra actions add an overhead on the first open call but only need to be done once. If the file needs random-writes, Shimmer forwards the unmodified call to the random-write path.

`close()`. Shimmer's close call closes the file descriptor, flushing remaining data in the buffer down to the device.

`unlink()`. On unlink, Shimmer marks all extents of a file as deleted in per-zone Metadata, if all extents on a zone are deleted, it erases the zone and adds it to the list of free zones.

Algorithm 1: The `int open(...);` call

```
int open(const char *path, int oflag, ...) {
if First Invocation then
    check for state
    load/initialize state
    start background threads
end
if $path$ not in Files then
    Assign $uuid$
    Add $name, uuid$ to Files
end
if $path$ matches a rule in $get\_hint()$ then
    if mode is ReadOnly then
        locate zone for $uuid$
        open zone as read
        return file descriptor
    else
        get zone from hint function
        allocate write buffer
        return file descriptor
    end
else
    Perform unmodified call
end
}
```

5.4.2 Writes and Reads

`pwrite() / write()`. The `write` and `pwrite` calls copy data from application buffers to the Shimmer buffer cache. Since `open()` handles allocation of buffers to append-only files, we check if such a buffer exists; if not, *shimmer-vfs* forwards the write to the random-write file system without any change. While the `write()` calls do not persist data on disk, once writes to a single file build up extent-sized amount of data a background thread flushes this data updating the file metadata. The write logic supports asynchronous engines such as `io_uring` [4].

`pread() / read()`. The read and pread calls need to be routed to where the file is mapped on the zones. From the maintained metadata *shimmer-vfs* gets the zone and an offset for a particular file and redirects to call to a file descriptor pointing to the zone with this new offset. If the file is not stored by Shimmer we do not modify the call. In some cases, especially when extent size is small, a single read could span multiple extents, possibly on separate zones, *shimmer-vfs* incurs the cost of additional reads in this case. We leverage the kernel buffer cache on top of ZenFS to support fast random reads.

5.4.3 Persistence

`fsync() / fdatasync()`. *shimmer-vfs* uses `fsync()` and `fdatasync()` along with other calls such a `sync_file_range()` to ensure that both Shimmer and the filesystem state is up to date when the application expects it. This involves syncing file buffers, and in case of fsync, persistence of the changes in Namespace and ZoneMap to dedicated regions on the SSD. These are stored in the bincode format as a binary file and can be restored on the next run as a part of the first call to `open()` function.

5.4.3.1 *Other modified calls*

In addition to the previously discussed calls Shimmer needs to modify other calls to guarantee persistence, prevent extra allocation, and maintain a consistent state. Shimmer performs the following actions on each of these calls:

`fallocate() / ftruncate()`. If files are hinted, Shimmer returns 0 without performing these tasks as they can end up allocating space on the original path.

`readahead()`. Performs a readahead on the mapped file rather than original path.

`rename()`. Changes the stored filename.

`mmap()`. Supports mmap read access only, mmap writes are not supported due to the append-only nature of ZNS.

`__pread_chk()`. Used by WiredTiger over `pread()`.

Shimmer allows specific files not to be intercepted using the hint mechanism. *shimmer-vfs* uses this for random-write files, LOCK files, and other accesses such as `procfs` reads that and application needs. This ensures that applications work with minimal modifications and not get any unexpected errors, utilizing the default path for anything not implemented by Shimmer. Since these files are kilobytes to megabytes in size Shimmer puts all the log-structured data, which makes up most of the data on log-structured systems, on ZNS, relying on relatively small random-write space.

5.4.4 Limitations

Shimmer's implementation has certain limitations due to the focus on simplicity and minimizing performance overhead trading off some of the flexibility possible with other approaches. `LD_PRELOAD` cannot work on statically linked applications. However, this is not a big issue in practice as the applications we tested (RocksDB, CacheLib, and WiredTiger) dynamically link with the system `libc` even when they are internally statically linked. Shimmer cannot be used with golang applications as they use system versions of `syscalls` rather than linking with `libc` wrappers. These applications would need an approach using ptrace, eBPF, WASM runtimes, or modified Golang Libraries. It is not always easy to preload the library for complex client-server applications, as a number of processes perform different activities. A filesystem approach or a custom C library can address this.

Currently, *shimmer-vfs* is designed for log-structured append-only applications. Since SSD-optimized applications generally follow this pattern, we can adapt several applications to this interface, but applications with data structures that use in-place updates or mmap writes (like BTrees) cannot utilize *shimmer-vfs* and will be passed through to the filesystem on conventional zones. Shimmer can still issue hints for these writes if the underlying filesystem supports them. Any other utilities that need to access the same files will need to be preloaded with *shimmer-vfs* and will need log-structured writes as well. In the future, we will explore limited support for in-place updates or filesystems that can achieve this to be tuned for Shimmer. We put together *shimmer-vfs* to demonstrate the possible performance gains, and in its current form, it is not a filesystem replacement and does implement filesystem operations like access control.

We could address these limitations by implementing Shimmer using other techniques: user-space filesystems, layered filesystems, interception using eBPF, custom versions of `libc`, and other approaches. As we will see in Chapter 7, our focus was simplicity and performance.

5.5 Integrating hints into Shimmer

Since Shimmer hints capture both temperature and lifetime, translating them to device-specific hints that support one of those parameters can limit their effectiveness. We hope that Linux offers a richer hint interface with four generalized temperatures across all applications in the future. *shimmer-vfs* on top of ZNS devices can demonstrate the full potential of a powerful hint system. Shimmer achieves this by maintaining a set of zones for each stream, co-locating extents with the same lifetime on the same zone. Zones are allocated to each stream from a queue of free zones and returned to the queue on erase or garbage collection.

Shimmer's hints are file specific and will not be effective on non file interfaces such as NVMe commands or object-based storage. On devices that support flexible data placement, if a kernel API is made available, Shimmer could forward the lifetime-group along with the write request indicating the hint for the device. Shimmer's approach is optimized for log-structured applications and assumes creation of several files. This is insufficient for applications that write to a single file (SQLite for instance). Currently there is no way to indicate hints for each write as there is for files or objects. We plan to explore this in the future.

⁂

Summary

As each density increase impacts NAND flash performance and lifetime, it is essential to rethink the random-write-friendly interface SSDs present, as it can exacerbate these effects. However, adopting new interfaces requires large rewrites and multi-year efforts with the loss of optimizations for current interfaces. Shimmer shows that you can significantly reduce the effort needed to enable applications to be protected from storage interface changes, yet at the same time benefit from the new interface. Shimmer decouples hints and placement from filesystems and applications as they have a limited view of the entire system and are not optimized for specific workloads or architectures that a data center might have. Shimmer gives the administrators of these systems the power to make adjustments according to their use case without modifying the application or the filesystem. With no system or application modification, Shimmer can use modern storage interfaces and achieve better performance than application-specific solutions.

6

DIRECTIVES FROM PARAKEET

"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to," said the Cat.

—Lewis Carroll • Alice's Adventures in Wonderland

Synopsis

A major issue with modern SSDs is the nebulous guidance on directives. Placement directives require host and application input, but manufacturers cannot provide specific guidance on generation of such directives as they are application and workload dependent. As a result, the industry focus is more on end-to-end systems rather than guidance to developers or users of the application, limiting the adoption. This chapter focuses on generating directives based on application patterns and workloads and translating them to interface-specific placement directives. We then present a guide on using these directives to place data for underlying systems.

⁂

While we have explored the limitations of directives and the guidance, the term *hint* remains unclear; after all, what is a hint? So far, we have intentionally used the broadly applicable phrase *hint*, which refers to different things across protocols. Broadly, a placement directive is a host-issued tag on data that the device can use. A directive could indicate data temperature: hot-warm-cold, the nature of the workload: random-sequential, or the deletion sets: sets of files deleted together.

The chief reason for the lack of directive usage is the unavailability or limitations of the Application Programming Interfaces (APIs) available to provide directives. While `fadvise()` is used to indicate sequential/random, `fcntl()` allows four temperature levels shared by the system. These directives do not take into account the deletion of the files, which has the potential to significantly reduce garbage collection if it can co-locate files related by lifetime.

Directives across protocols vary further— ZNS expects a zone number to append to but requires host-managed zones, while FDP requires a *placement directive* at the NVMe command level. Neither of these techniques matches with what is available in the kernel, which is four levels of temperature hints. With Parakeet, we propose an abstract representation of directives, which we can convert to these formats and any future formats. directives in Parakeet can capture more information than system directives and perform lossy conversion to other formats. We demonstrate such conversion with the F2FS tests, while *shimmer-vfs* can use the full potential of the richer directives.

6.1 Directives in Parakeet

Inspired by different types of SSD directive designs, Parakeet organizes *lifetime-groups* as sets of integers in an arbitrary number of *streams*. The *stream* represents writers, while the *lifetime-group* maintains a group of files expected to have a similar lifetime. The streams are application-specific, while lifetime-groups are stream-specific. For example, an application could have WAL and DATA streams, each of which maintains lifetime-group for groups of files based on creation time or other parameters. Another application may choose to use HOT, WARM, and

COLD streams with similar lifetime-group grouping. We designed Parakeet's internal directive representation to be simple and extensible, allowing easy translation into the available hint formats and any future changes.

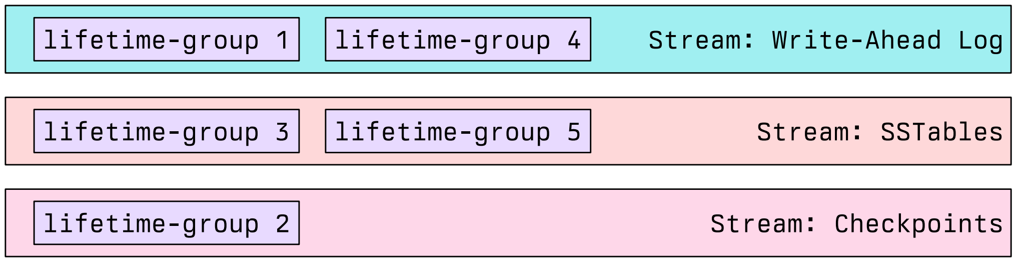

Figure 6.1: Parakeet directives are organized into streams of writers with multiple groups of files based on lifetime.

Parakeet's `get_hint()` API maintains a simple, extensible interface returning an integer, a lifetime-group, or a stream depending on the interface. Streams can be added by users familiar with application semantics and workload or generated automatically based on observation.

Internally, Parakeet organizes each file in a dedicated *lifetime group*, which are mapped to data streams which group related data. For instance, for RocksDB as we see in Fig. 6.2, the data streams contain different types of files, and while writing, the stream is used to group files on the same zone (which makes up the lifetime group).

```
enum Stream {
    SSTable,
    LOG,
    #[default]
    Undefined,
}
static FD_TO_STREAM: LazyLock<DashMap<usize, Stream>> =
LazyLock::new(DashMap::new);
```

Figure 6.2: Parakeet with RocksDB creates streams for different types of files, maintaining directive for each file in the dedicated map of file descriptors to streams.

6.1.1 Translating Parakeet directives into device directives

Since Parakeet directives capture both temperature and lifetime, translating them to device-specific directives that support one of those parameters can limit their effectiveness. For instance, as we discuss in the evaluation, even with Parakeet provided stream directives we see no impact on performance. For the current Linux directive interface, achieved through `fcntl()`, Parakeet needs user configuration for indicating which of its streams to map to the `RWH_LIFETIME` directives.

shimmer-vfs on top of ZNS devices can demonstrate the full potential of Parakeet's directive capability. *shimmer-vfs* maintains a pool of zones for each Parakeet stream, ensuring that files with the same lifetime-groups are co-located on the same zone. Zones are removed from these pools when they are erased and added as needed from a big pool of available zones. For performance isolation, *shimmer-vfs* uses different sets of zones for streams along with different sets of zones depending on the lifetime-group. On devices such as ZNS SSDs, since write buffer allocation is done by *shimmer-vfs*, we can leverage these directives to provide performance gains for multi-application workloads.

On devices that support flexible data placement, if a kernel API is made available, Parakeet could forward the lifetime-group along with the write request indicating the directive for the device. Parakeet's directive approach can work with object-based storage, although there is no support for directives for such an interface through the kernel. Parakeet's approach is optimized for log-structured applications and assumes generation of sets of files. This is insufficient for applications that write to a single file (SQLite for instance). Currently there is no way to indicate directives for each write as there is for files or objects. We plan to explore this in the future.

6.2 Heuristics for log-structured systems

To demonstrate Parakeet we focus on log-structured IO-intensive applications on top of flash. Write-intensive applications include key-value stores, databases, and caching systems. To address these, we focus on storage backends of such systems, including RocksDB and WiredTiger. As seen in Table 6.1, only small configuration changes are needed to to enable applications with similar characteristics.

Configuring Parakeet

To use Parakeet the administrator only needs to configure the number of streams and the rules to categorize files into those streams. This can be as easy as indicating a file extension tied to a stream.

Table 6.1: Lines of code

Parakeet (base)	**RocksDB**	**WiredTiger**	**CacheLib**
551	3	5	2

Only a few lines of changes are needed to support log-structured applications with Parakeet. In some cases like CacheLib, application configuration is sufficient for Parakeet to provide data placement.

6.2.1 RocksDB

RocksDB presents two major IO-intensive logs: the Write-Ahead Log (WAL) represented by the `.wal` files and the Sorted String Table (SST) represented by the `.sst` files. We use this to allocate a WAL stream and a DATA stream. With time new `.wal` and `.sst` files are created from inserts and compaction. The WAL files have a very short lifetime and are deleted as soon as a new one is created, while depending on the compaction, the SST may be valid for a long time. Parakeet can be configured to work with any compaction, but for *shimmer-vfs*, universal or FIFO compaction schemes are better than level-based compaction as they can offer predictable file sizes and lifetimes. RocksDB generates many other files, including `LOCK`, which *shimmer-vfs* cannot support; hence, it doesn't interpose on those calls.

6.2.2 WiredTiger

For WiredTiger in Log-Structured Merge tree (LSM) mode, *shimmer-vfs* only supports remapping the `.lsm` files, as WiredTiger uses `mmap` writes for its WAL. These `.lsm` files are put in a single stream and mapped to zones based on the lifetime. WiredTiger performs several operations that complicate its use with *shimmer-vfs*. For instance, it periodically writes a single page as a root page to test write ability, using a stat call and a read to ensure it persists. It uses internal `libc` functions for some tasks (for instance `__pread_chk()` over `pread()`), so Parakeet needed to add mapping from these functions to their standard wrappers. Even with `O_DIRECT`, multi-threaded writers can contend on a single `lsm` file, so *shimmer-vfs* currently only supports single-threaded updates. This is not a limitation of the approach, and we are exploring solutions to serialize these writes.

6.2.3 CacheLib

For CacheLib, the backing files require custom names, we configure CacheLib's backend, Navy to create multiple block files for various types of cache and indicate these files with directives for Parakeet.

6.3 Automating directives in Parakeet

The heuristic directives discussed in § 6.2 require configuration per-application, and possibly per-workload. The major advantage of decoupling hints from the application and the filesystem is that we can plug custom hint generation functions into the `get_hint()` interface, which can generate directive based on a trained model on deletion of files.

6.3.1 Capturing data

We use the same techniques that we use to inject directives to capture logs of application usage information. As seen in Fig. 6.3 we use Shimmer to log updates to a database along with timestamps, which can then be used to train a clustering algorithm on. As the technique uses statistical machine learning rather than deep learning, we decided to use `scikit-learn`

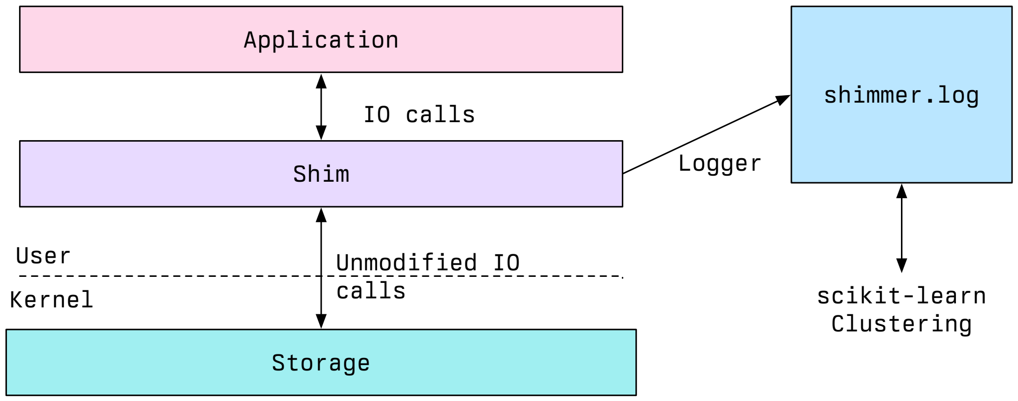

Figure 6.3: Shimmer can help log actions which can then be used to train data.

We use logic as seen in Fig. 6.4 to capture call state, any flags, timestamp and other details by running an application linked with a specialize Shimmer layer. This is logged to a file which is converted into our training data.

6.3.2 Clustering

Various clustering techniques can then be used with this data, allowing dynamic or static directive generation based on the technique. Techniques like Density Based Spatial Clustering of Applications with Noise (DBSCAN) [38] or k-Nearest-Neighbors can be used. In the example above, we only use data available at open time, but this can further be expanded in read, write, close, and unlink informa-

```
unsafe fn open(pathname: *const libc::c_char,
flags: libc::c_int,
mode: libc::mode_t) -> libc::c_int => my_open {
let fd: libc::c_int = redhook::real!(open)(pathname, flags, mode);
let path = CStr::from_ptr(pathname).to_str().unwrap().to_string();
let filename = match std::path::Path::new(&path).file_name() {
      Some(file) => file.to_string_lossy(),
      None => return fd,
  };
  let entry =format!("{},{:b},{:b}\n",filename, flags, mode);
  Logger.write_all(entry.as_ref()).unwrap()
}
```

Figure 6.4: Shimmer allows us to capture trace-level analytics on calls like open, close, and unlike, allowing grouping related files.

```
$ cat /tmp/shimmer.log
 1    | filename,flags,mode
 2    | LOG,10000000001001000001,110100100
 3    | LOCK,100000000000001000010,110100100
 4    | LOG,10000000001001000001,110100100
 5    | LOCK,100000000000001000010,110100100
 6    | 000000.dbtmp,10000000001001000001,110100100
 ...
24    | 000008.log,10000000001001000001,110100100
25    | 000009.sst,10000100001001000001,110100100
26    | 000010.log,10000000001001000001,110100100
27    | 000011.sst,10000100001001000001,110100100
28    | 000012.log,10000000001001000001,110100100
```

Figure 6.5: We use such captured log data to train a clustering model.

tion captured by Shimmer. With DBSCAN (see Fig. 6.6), even with just the flags and mode of the file, the algorithm can achieve distinction similar to our heuristic, grouping SST files and short-lived files separately.

```
clusters = DBSCAN(eps=4.54, min_samples=4).fit(data[['flags','mode']])
data['stream'] = pd.Series(clusters.labels_, index=data.index)
print(data[['filename','stream']])
         filename stream
0             LOG      0
1            LOCK      1
2             LOG      0
3            LOCK      1
4    000000.dbtmp      0

451    000437.sst      2
452    000438.log      0
453    000439.sst      2
454    000440.log      0
455    000441.sst      2
```

Figure 6.6: In its simplest iteration, DBSCAN is able to group files based on expected workload into 3 streams, separating lock files from log, and SST.

Shimmer can then use these streams to group files, using timestamps to break them into placement groups. Currently, the system uses a trained model, but we plan to integrate more dynamic hint generation in the future.

⁂

Summary

In this chapter, we looked at directives determined by heuristics and machine learning. More significantly, we motivate the need to separate hint generation from the application and the storage system, as it enables a richer view of the interaction of these components. Further, an intermediate representation allows flexible hints that we can translate to changing standards. Armed with rich hints, let's see how filesystems and Shimmer can benefit from them in Chapter 7.

7

EVALUATION

"And now that you don't have to be perfect, you can be good."

—John Steinbeck • East of Eden

To demonstrate the benefits of Shimmer, we focus on four types of tests:

① We compare throughput and tail latency of *shimmer-vfs* with the application-specific approach of ZenFS and the filesystem approach of F2FS, the two systems that make up the state of the art.

② We showcase the versatility of Shimmer with additional applications.

③ For multi-tenancy, we test several concurrent writers on each system.

④ And finally we measure overheads such as write amplification and memory consumption compared to F2FS.

For the tests not involving RocksDB, we only compare *shimmer-vfs* with F2FS as it is the only system that supports those applications on ZNS SSDs.

System Setup

To evaluate Shimmer, we set up a test server with two ZNS SSDs with the following specifications:

System Specifications

CPU:	2 x AMD EPYC 7542
Cores:	32 each
DRAM:	128 MB DDR4 @ 3200 MHz
Ubuntu:	22.04
Linux Kernel:	6.2
Storage:	2 x Western Digital DC ZN540
Capacity:	4 TB (formatted to 512 GB)

We used latest versions and the benchmarking tools bundled with each of the systems: `db_bench` for RocksDB, `wt_perf` for MongoDB, and `cachebench` for Cachelib. Between runs of each benchmark, we issued Zone erase and format commands to ensure that each experiment was not affected by unexpected device operations. We set up the random write area required by Shimmer and F2FS on the 4 GB of random write area available on each ZNS SSD so writes would go to the same device; limiting our device capacity to 512 GB as F2FS needs a larger random write region beyond that capacity.

7.1 Shimmer vs F2FS vs ZenFS

We started our evaluation with RocksDB as it has an application-specific backend ZenFS for ZNS SSDs, and several filesystem optimizations such as the ability to provide F2FS with write hints. As ZenFS leverages application insights to intelligently place data, we wanted to compare our approach to such a tuned system.

As seen in Fig. 7.1, for inserts and updates, Shimmer offers almost $2\times$ improvement over F2FS and can match tailored approaches like ZenFS. Such performance gains are possible because of Shimmer's efficient design. Shimmer provides intelligent hints discriminating between the frequent small writes of the Write Ahead Log, and the large writes of Sorted String Tables (SSTables), utilizing separate device buffers to provide each stream isolation. Shimmer sees a small overhead on reads, and particularly readwhilewriting, due to the additional overhead of ex-

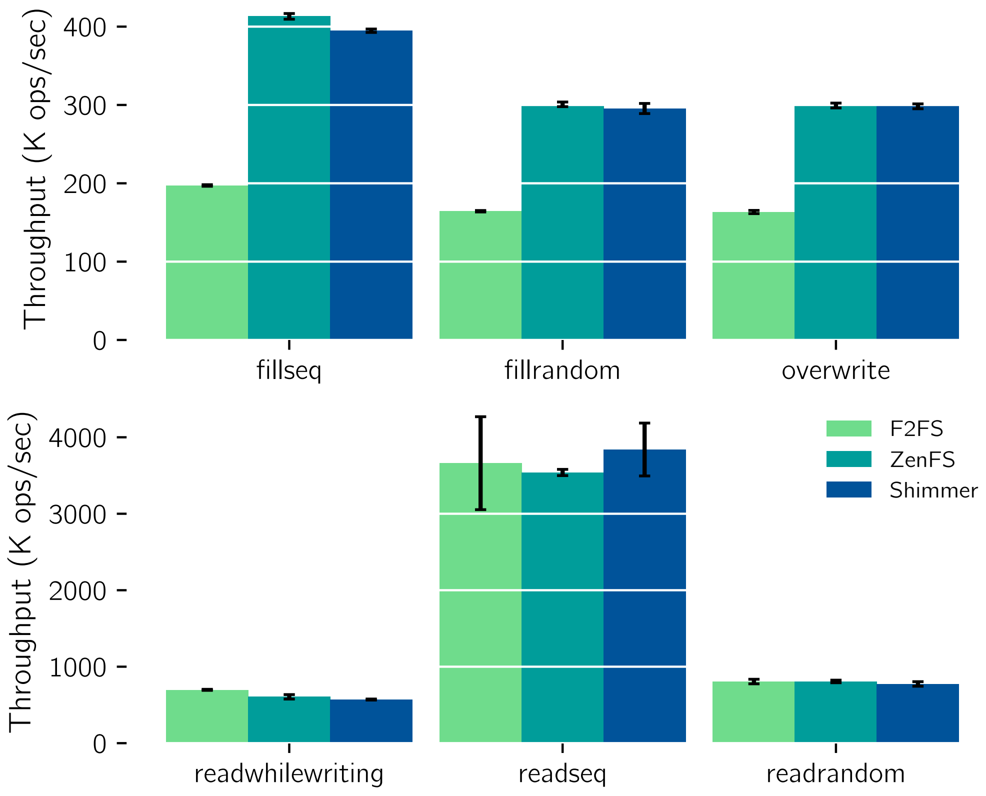

Figure 7.1: Throughput for fillrandom, overwrite, readwhilewriting and readrandom workloads of RocksDB's `db_bench` benchmarking tool.

tent and address resolution in userspace. Even with hints, we observe that F2FS is unable to utilize the full device bandwidth.

RocksDB Evaluation

```
1. Operations:  10 Million
2. Key Size:    20 B
3. Value Size: 400 B
4. Compaction: FIFO
5. Benchmarks:
    a. fillseq - Sequential Insert
    b. readseq - Sequential Read
    c. fillrandom - Random Insert
    d. readrandom - Random Write
    e. overwrite - Update keys and values
    f. readwhilewriting - Reads and Writes
```

One of the main benefits of write isolation is the improvement in tail latency. *Shimmer's user-space nature ensures that the latency impact of the additional operations remains minimal compared to the high impact of kernel-crossings and persistence.* As seen in Fig. 7.2, Shimmer offers better tail latency over both F2FS and ZenFS for random inserts and random reads. This is particularly evident at p99.99, where Shimmer benefits from the filesystem optimizations in RocksDB by intercepting `readahead()` and pre-buffering data, a filesystem optimization unavailable to ZenFS.

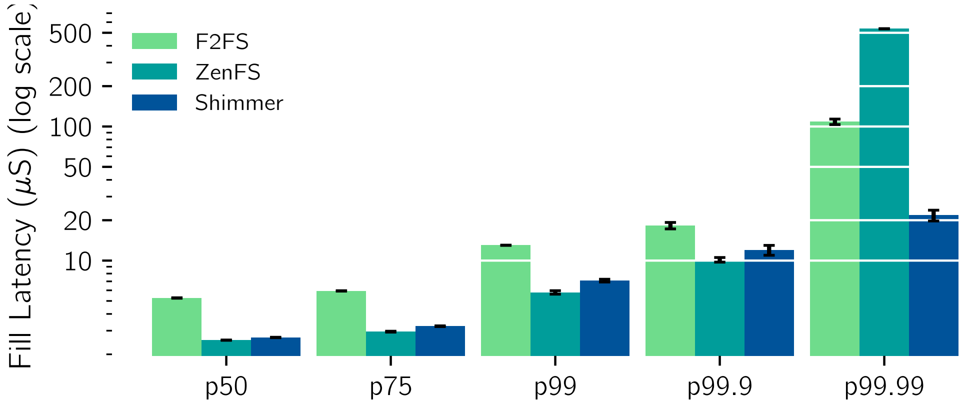

Figure 7.2: Shimmer offers the lowest tail latency among all approaches.

Why is Shimmer faster?

Digging deeper into the performance metrics as seen in Fig. 7.3 we see that when we dive deeper into the call stacks of these individual systems, for F2FS, the benchmark spends a large time in the `f2fs_file_write_iter` function, called by the write-ahead log, which calls various metadata and data functions compacting incoming streams across its six logs, performing metadata updates, and allocating and freeing memory in the cache to support these operations. As seen in the slices below this function, F2FS triggers several kernel mechanisms for cache management, writes, and garbage collection on writes.

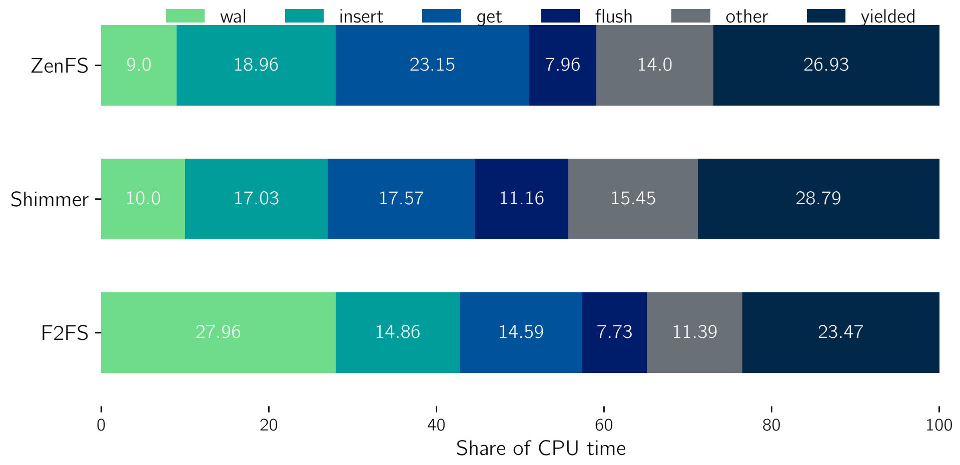

Figure 7.3: Breakdown of performance shows Shimmer and ZenFS with comparable write performance as opposed to the additional overhead of F2FS.

On the other hand, Shimmer write-ahead log, and insert make up a relatively small chunk of CPU time, each calling a single stack of operations, typically persist and saving state. These functions cause bounded activity in the kernel, as the response is simpler and more akin to a get/put interface than a filesystem. The userspace overhead is also limited, with these functions accounting for less than 27% of total program time, similar to ZenFS as opposed to the 44% of F2FS. Reads (get operation) in Shimmer are more efficient than ZenFS, but add slight overhead over F2FS, accounting for 17.5% of the time as opposed to 23% on ZenFS, which explains the latency spikes seen in Fig. 7.2.

In Shimmer the close operation can cause high execution time as it truncates any full zones that are completely invalidated, while this adds overhead on close, it reduces the overhead on write that is seen in F2FS as it needs to free up space while handling a write call.

7.2 WiredTiger and Cachelib

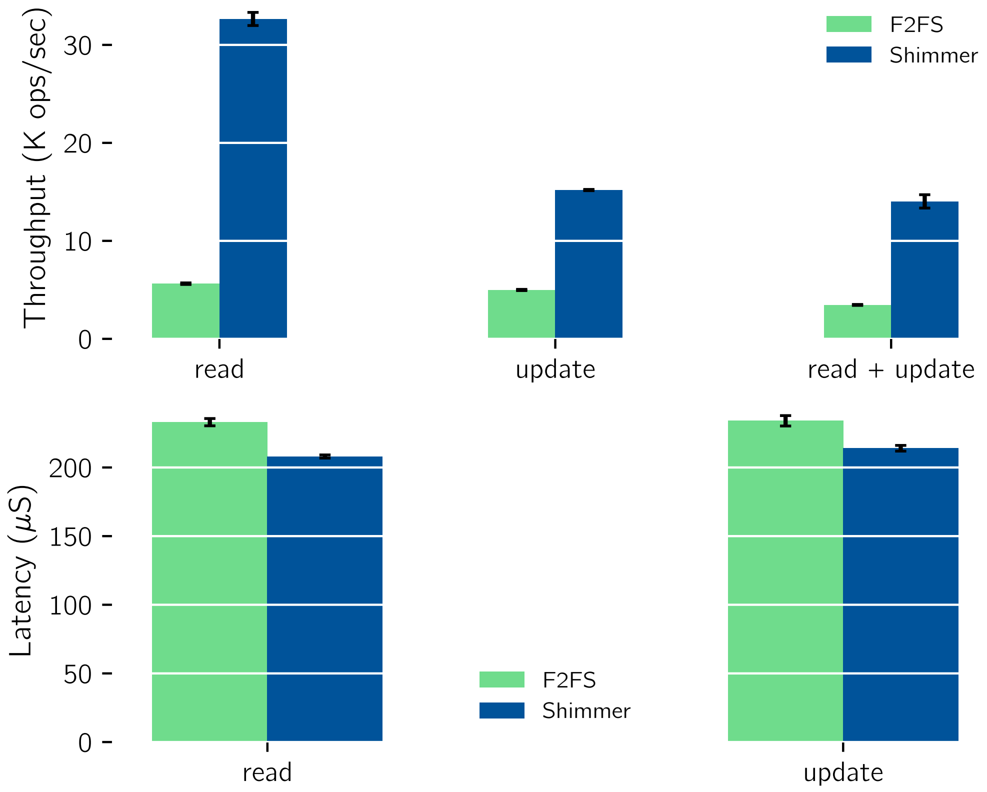

Figure 7.4: Due to physical separation of log and lsm files to dedicated zones, Shimmer can allow dramatically faster writes and updates in WiredTiger.

While RocksDB's compose-able nature makes it a great target for testing new interfaces, one of the main benefits of Shimmer's approach is generalizability, where it can be deployed to different applications, and to show that we made simple *shimmer-vfs* layers for WiredTiger's LSM Tree Mode and to test a non-storage backend workload Cachelib: a caching library.

We tested WiredTiger's performance by modifying the medium-multi-lsm test in `wt_perf`, WiredTiger's benchmarking tool to insert 10 Million entries in an LSM tree and then perform reads and updates in separate threads. For WiredTiger, MongoDB's storage backend, we observe a significantly improvement in updates and reads in the LSM mode. As we see in Fig. 7.4 Shimmer with multi-threaded readers can performs more than $6\times$ faster, with write-heavy workloads up to $3\times$ faster. These are enabled by the ability to separate logs that go to the random write area and streams of LSM files that stay on zones. Physical separation of mmap-writes in the dedicated random write area, while log-structured writes are appended to dedicated zone help improve the performance.

We adapted Meta's caching library Cachelib [9] to *shimmer-vfs*, which required no change in Shimmer's logic. The only required change was configuration of Cachelib to use separate files for different types of caches. As we see in Figs. 7.5 and 7.6, Shimmer offers lower latency for reads and writes (except maximum latency, a single spike required for our setup tasks), and improved throughput especially with increase in files.

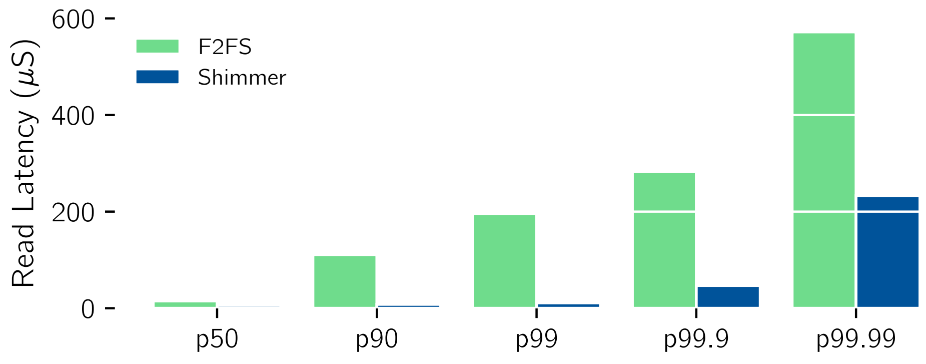

Figure 7.5: With Cachelib's `cachebench` tool, Shimmer outperforms F2FS on throughput and read latency

To perform this test we set up cachelib with two files as the spillover area of 128 MiB cache in memory. We named cache files to have `.sst` and `.log` extensions depending on the type (Cache of large or small objects) and used unchanged Shimmer from RocksDB to map them to different zones. We then performed 4 million get and set ops on the cache, with a majority of them getting to storage to measure storage performance

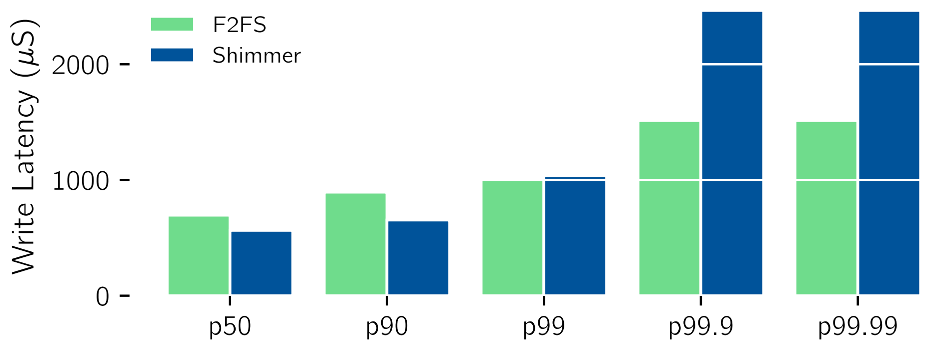

Figure 7.6: With Cachelib's `cachebench` tool, Shimmer outperforms F2FS except the maximum write latency induced by initial setup

7.2.1 Multi-tenancy

ZNS SSDs can eliminate degradation caused by write interference from multiple streams of incoming writes. With Shimmer, multiple applications can be run in parallel on the same SSD with a significantly lower degradation in performance as their writes are isolated to separate write resources. To evaluate degradation, we ran fio sequential insert and RocksDB random insert workloads simultaneously on the same device. This results in 3 concurrent writers, each multi-threaded: RocksDB, RocksDB-WAL, and WiredTiger. We measured the throughput and latency on both WiredTiger and RocksDB, focusing on tail latency, which is typically adversely affected by such workloads.

Table 7.1: Average tail latency

	Shimmer (μs)	F2FS (zoned) (μs)	F2FS (conv) (μs)
p99.99 (iso)	19.02	21.11	27.35
p99.99 (int)	20.47	26.85	32.65
pmax (iso)	3518.00	3368.00	9442.00
pmax (int)	4523.00	4386.00	10129.00

Workloads with interference (int) see tail latency spikes over isolated workloads (iso) due to multiple writers contending for the same resources. On a ZNS SSD, Shimmer can utilize the control over write buffers to prevent contention reducing the impact on p99.99 and pmax tail latency. When we look at the tail latency in Table 7.1, we see the adverse impact of write interference on a shared SSD. Shimmer sees a 22% increase in pmax and virtually no increase in in p99.99 latency, which is smaller than F2FS's 21% increase in p99.99 and 23% increase in pmax latency. Compared to F2FS on conventional SSDs, the advantage of host-device hint communication becomes apparent, with F2FS having high spikes in the tail caused by write interference and garbage collection.

7.3 Write amplification

Reduced garbage collection and improved grouping help reduce write amplification as data can be deleted together, avoiding the amplification caused by moving data. As there is no on-device garbage collection, *shimmer-vfs* sees a device WAF of 1, similar to F2FS (zoned) and ZenFS. In the benchmarks we discussed, erasing Zones did not involve moving any data, as either a log or an sstable zone was completely invalidated at the time of reclaim. The compaction policy of the LSM tree will largely dictate the need for valid data relocation. FIFO and Universal compaction generally result in periodical deletion of old data, which can free up older groups as new data is written. For a leveled approach, the largest levels will not be compacted frequently enough to enable zone erase and may need relocation.

As we see in Table 7.2, in our write-intensive experiments, Shimmer frees up more space without moving any data as WAL blocks are freed up quickly, allowing shimmer to erase without the need for relocating data.

	Shimmer	F2FS
Garbage Collection Calls	8	4
Data Moved	0	1059 MiB
Free Space at the End	48.2 GiB	33.2 GiB

Table 7.2: We ran an insert-heavy benchmark with updates, inserts, sequential, and random writes to compare F2FS and Shimmer's garbage collection performance. Shimmer benefits from co-location of related data to have zones that are completely invalid, ready to be freed. ZenFS currently does not implement garbage collection so it is not included in this table.

7.4 Memory overhead

Memory in Shimmer

The buffer cache (Extent Size $\times$ Write-able Files) +
ZoneMap (32 bits $\times$ Addressable Zones) +
Stream Hints (64 bits $\times$ Hint Streams) +
File Data (64 bits $\times$ Extents per File $\times$ Open Files)

Shimmer in hint-only mode does not maintain any file data and uses no extra memory. However, *shimmer-vfs* needs to perform write buffering, which significantly requires allocating some amount of memory per writable file. This can be controlled by limiting the extent sizes. To speed up access, Shimmer maintains a few more maps in memory, including the ZoneMap which adds 32 bits per addressable zones, NameSpace which maintains extents of all active files, and a hint-to-zone map which is a fixed size structure with its size depending on the number of active streams. In our experiments we used 32 MiB buffers with 2 write streams, totaling 64 MiB write buffers which made up most of the actively used memory. This size is smaller than the buffers maintained by filesystems in the kernel. Profiling the memory using KDE heaptrack [34], *shimmer-vfs* used 73 MiB at its peak in the evaluation experiments.

PART III

EPILOGUE

“Painting is special, separate, a matter of meditation and contemplation, for me, no physical action or social sport. As much consciousness as possible. Clarity, completeness, quintessence, quiet. No noise, no schmutz, no schmerz, no fauve schwärmerei. Perfection, passiveness, consonance, consummateness. No palpitations, no gesticulation, no grotesquerie. Spirituality, serenity, absoluteness, coherence. No automatism, no accident, no anxiety, no catharsis, no chance. Detachment, disinterestedness, thoughtfulness, transcendence. No humbugging, no buttonholing, no exploitation, no mixing things up.”

—Ad Reinhardt, statement for the catalogue of the exhibition, “The New Decade: 35 American Painters and Sculptors,” Whitney Museum of American Art, New York, 1955.

8 DISCUSSION

"Everything changes once we identify with being the witness to the story, instead of the actor in it."

—Ram Dass

The journey from Persimmon to Shimmer explores several aspects of systems on top of modern SSDs, from the benefits of host-device coordination to the role of kernel and userspace in enabling these benefits. As we add additional responsibility for placement and performance hints in our systems, the increase in complexity is unavoidable. However, where we add the complexity determines the benefits that we can get. Increasing the complexity of the application can unlock full benefits but can limit the generalizability of the approach. Changes to the filesystem can ease the transition but limit unlocking the full performance and possibly introduce a greater attack surface.

8.1 Main takeaways

A Cosmic View. Applications are built to be storage-unaware and filesystems to be application-unaware. Any modification to this paradigm will impact a system's portability, compatibility, and complexity. To maintain these properties and yet to incorporate a global view of a system, adding a system-specific layer like Shimmer is the easy solution. When hinting is *decoupled from the application and the filesystem,* it is easy to introduce custom hint frameworks like Parakeet, which can be tuned per application and workload. Putting the hint capabilities in the other places would result in kernel or application upgrades and won't be hot-swappable.

Break layering abstractions by adding a layer. While new interfaces offer performance and cost benefits, rewriting existing systems is complex. As seen in Fig. 8.1 Shimmer can shield data management systems from complex hardware hierarchies to simplify their management. Administrators can easily configure applications for specific hardware and workloads with a shim layer, unlocking benefits that applications or filesystems cannot offer without rewrites. Importantly, modern systems result from decades of optimization for interfaces presented by the kernel. While it might be tempting to rewrite everything in modern paradigms, without strong primitives, it will simply lead to increased fragmentation. Adding layers in between allows us to ease such transition until modern standards take the de-facto role, allowing us to enjoy the benefits at a lower cost.

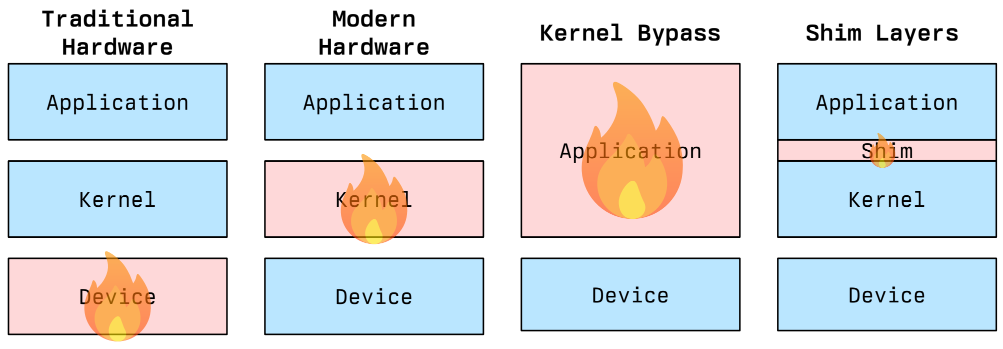

Figure 8.1: The complexity of managing flash cannot be eliminated, just moved across layers. Rather than adding such complexity to the kernel, or the application, we can isolate it in a small configurable layer and allow easy adoption of modern interfaces.

Abstraction boundaries present shim opportunities. Filesystems simplify application interfaces to the standard VFS API. They further simplify sharing data, maintenance, replication, and backups. Even with data-intensive workloads like large language models, sharing, loading, and processing data is on top of the file abstractions [17]. Cloud storage backends simulate filesystems, as that is what interface applications expect. With a standard interface like the VFS, shim layers can maintain compatibility while opening up new opportunities. Porting systems to Shimmer required relatively small code changes as opposed to dedicated storage backends due to the universal nature of the filesystem interface.

Abstractions offer consistent interfaces: from the user-kernel boundary to the kernel-device boundary. As hardware sees radical changes, shim layers can enable adoption while minimizing disruption across the stack.

Reducing the kernel attack surface. As seen in Fig. 8.2, as filesystems serve diverse and changing hardware, they increase in complexity over time. This complexity can increase the attack surface in the kernel, as seen in critical vulnerabilities discovered in F2FS (Fig. 8.3). ZoneFS has a footprint in the kernel that is smaller than F2FS by a factor of 40, presenting a simpler design with smaller attack surface. Shimmer's all userspace design on top of such a system presents a streamlined stack that can be hardened much more easily to reduce bugs and exploits.

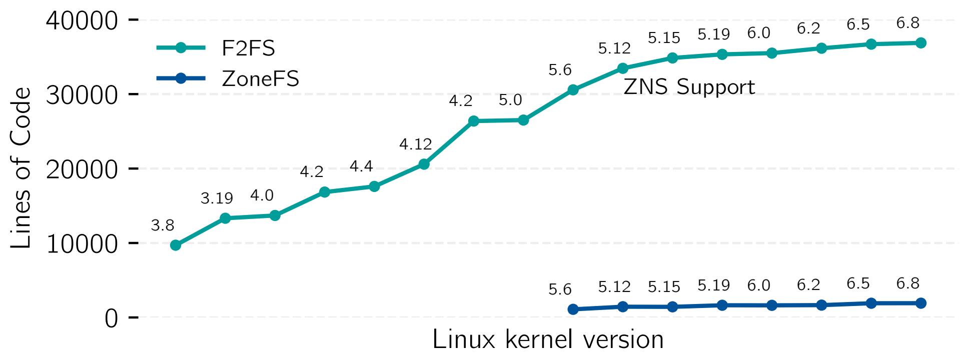

Figure 8.2: With diversifying hardware, the complexity of filesystems keeps increasing, increasing the attack surface in the Linux Kernel. Here, Shimmer's approach of using ZoneFS can minimize the complexity, and hence the potential for bugs by using a significantly simpler filesystem.

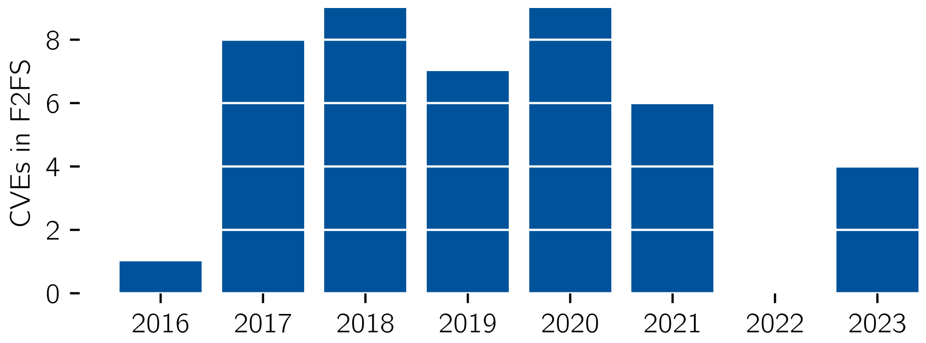

Figure 8.3: Critical vulnerabilities and exploits (CVEs) in F2FS over time.

Other approaches may avoid such complexity by bypassing kernel altogether, with approaches such as SPDK [70], or with newer kernel approaches like I/O Passthru [30], however such approaches move security guarantees to the application layer and disable kernel safeguards. Shared user-kernel space approaches like eBPF or `io_uring` open up escape hatches that have been widely exploited [35].

Static and dynamic hint communication. We assert that hints at compile time, like those present in RocksDB for F2FS, are insufficient to unlock the full benefits, as they may not reflect the device characteristics or the workload, which may differ in architecture from compile-time assumptions. At the same time, plugins like ZenFS and shim layers like Shimmer demonstrate that the full benefits can be unlocked when configured at run time. Even with naive hints, Shimmer can unlock the full performance of a ZNS SSD as Shimmer leverages control over device buffers to dedicate regions of SSDs to groups of data and minimizes contention over buffering and placement.

Companies using modern data management systems have greater insight into the specific storage architecture and workload than the developers of such systems. It is challenging to enable such optimizations in the design of applications or filesystems as they need to cater to a diverse set of workloads. Our vision is that Shimmer can be utilized to prevent custom application forks or filesystems, enabling easy adoption and hint generation with local expertise.

Helping modern SSDs succeed. As we saw in Table 5.1, new interfaces are introduced and demonstrated with standard applications but struggle to be widely adopted due to the costs and risks associated with novel hardware. We already have examples of hint communication efforts (Open-Channel SSDs, Stream SSDs) that showcased what could be achieved but were unused and hence deprecated. If we do not enable an easy transition to NVMe standards, *proprietary interfaces will win*. Storage systems will become boxes with input and output at the mercy of hyperscalars. With ZNS and FDP, we have an opportunity to change this pattern.

In resource-constrained devices like smartphones, interfaces like ZNS are essential as the SRAM availability for caching the flash translation layer is limited [29]. With a Shimmer-like approach, we can leverage the similarities and differences between applications to generate valuable hints; for instance, each application may have an instance of SQLite but may see vastly different workloads. A Shimmer-enabled ZNS backend could reduce device costs while improving performance.

Picking efficient shim techniques. Modern times have seen a resurgence of shim layer techniques, from virtualization to layered filesystems, eBPF, Web Assembly, dynamic libraries, and binary patching techniques. Post-Moore architectures drove this urgent need as specialized hardware can eliminate the bottlenecks of the kernel and single host systems. On the other hand, the pace of change in software interfaces remains glacial, with filesystem interfaces still reigning supreme. We demonstrated (see Fig. 5.1) that any performance-oriented shim layer should *avoid patching binaries and repeated kernel crossings* to get the best performance.

Beyond SSDs. Changing hardware interfaces allow us to rethink the role of traditional systems like filesystems. Currently, filesystems have many limitations as they take on responsibilities ranging from encryption, compression, data management, storage management, placement, de-duplication, and host-device coordination. Such complexity is unnecessary and inefficient in the kernel. It is time to reconsider the storage stack, where layering would allow easy, configurable, drop-in replacements for each role, composing data management from tunable parts rather than a monolith gated beyond the kernel boundary.

8.2 Future work

We are actively working on designs for the following challenges:

① **Configuration format:** Currently each implementation of Shimmer needs to be written as an overloaded function with similar signature. We plan to automate this process with inbuilt primitives which can be configured with simple toml configurations.

② **Fast-path optimization:** We can override functions to use kernel and block-layer bypass mechanisms such as xNVMe, SPDK, and eBPF trace points like XRP [73] to channel NVMe commands directly into the driver.

③ **Shimmer daemon:** Currently, the lifetime of a Shimmer shim layer is tied to the program it is dynamically linked to, and separate Shimmer invocations along with different applications cannot communicate or coordinate. We plan to implement a daemon that can enables multi-tenancy of Shimmer-overloaded applications.

④ **Programming languages support:** Currently Shimmer and Twilight are implemented in Rust, and have bindings for Rust, C, and C++ code. Overloading functions in other languages is complex, but we plan on exploring how bindings to other languages might be achieved using C as a *lingua franca*.

⑤ **Additional architectures:** We plan to explore how Shimmer can be used with computational storage, KV-SSDs, memory-semantic SSDs, CXL, and other non-flash systems.

9 CONCLUSION

"The drama's done. Why then here does any one step forth? — Because one did survive the wreck."

— Herman Melville • Moby-Dick; or, The Whale

As we enter the post-Moore era, the bulk of improvements will be from efficient architectures rather than shrinking feature sizes. Improving efficiency necessitates re-evaluating software and hardware interfaces that are the staples of modern systems. Across systems, traditional abstractions present limitations we need to bypass to get the most out of the hardware. Therefore, it is no surprise that technologies such as Web Assembly (WASM) and extended Berkeley Packet Filter (eBPF) have seen a surge in popularity. Modern SSDs present one such challenge, requiring host-device coordination in systems designed to abstract data access from operation logic. Providing directives becomes difficult in the cloud, where storage layers are isolated across containers, performing repeated tasks at each layer.

We propose several ways to address such issues, combining rich hint generation, easy interposition, and flash-optimized filesystems. In this dissertation, we identify the issues— hint generation, hint provisioning, and hint utilization, and present techniques to perform each task.

Persimmon. With Persimmon, we explored different design choices for filesystems on SSDs and showed the benefits of a multi-log approach. While several log-structured filesystems still depend on *in-place* updates for various internal mechanisms, we demonstrated that this is not a fundamental requirement. Persimmon shows how host-device coordination can be bidirectional, using device-maintained metadata to keep track of its data. Persimmon improves space utilization, reduces tail latency, and embraces append-only writes for *both data and metadata*, all while maintaining the flash-optimized performance of F2FS.

Parakeet. With Parakeet, we look at the benefits of decoupling hint generation from the filesystem and the application, allowing dynamic hints that Parakeet can translate across interfaces. We propose a modern hint API, which takes into account *lifetime and temperature* of data and demonstrates the benefits of doing so. We showed how log-structured applications can be made more efficient by exploiting the temporal nature of generated data.

Shimmer. With Shimmer, we showed lightweight interception of applications, allowing hint injection and data placement without modifying the application or the underlying filesystem. Connecting Shimmer to Parakeet can allow applications to match specially tuned approaches like ZenFS with a fraction of the effort. Shimmer can further unlock better isolation between applications on the same system and improve write performance with lower write amplification. We demonstrate several applications with Shimmer, from data stores like RocksDB and MongoDB to caching libraries like Cachelib, improving performance for writes and reads.

The major advantage of leveraging traditional abstractions like POSIX is that our systems: Shimmer, Persimmon, and Parakeet, which were designed independently over the last few years, can work in tandem. We can place data with hints from Parakeet, remapping append-only files with Shimmer, and remaining files with Persimmon. While we applied these techniques to a narrow domain of modern SSDs, this approach is also useful in other domains. For instance, systems with CXL could allow transparent use of extended memory, and systems with specialized accelerators could allow hints about data streams to the accelerator through traditional layers of abstraction. With these systems, we demonstrate how new interfaces could be introduced with low overhead while shielding applications and operating systems from hardware changes.

BIBLIOGRAPHY

[1] Bytecode Alliance. *WASI: WebAssembly System Interface*. https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md. 2022.

[2] Bart Van Assche. *[PATCH v5 00/17] Pass data lifetime information to SCSI disk devices*. https://lwn.net/ml/linux-fsdevel/20231130013322.175290-1-bvanassche@acm.org/. 2023.

[3] Jens Axboe. "[PATCH v2] Support for write stream IDs." In: *Linux Kernel Mailing List* (2015). URL: https://lore.kernel.org/all/1430856181-19568-5-git-send-email-axboe@fb.com/T/.

[4] Jens Axboe. *Efficient IO with io_uring*. https://kernel.dk/io_uring.pdf. 2019.

[5] Jens Axboe. "Re: [PATCH 1/2] nvme: remove support for stream-based temperature hint." In: *Linux Patch Mailing List* (2022). URL: https://lore.kernel.org/all/164642520857.251407.3488369238595834309.b4-ty@kernel.dk/.

[6] Jens Axboe. *Linux Kernel Mailing List: [PATCHSET RFC 0/2] mq-deadline scalability improvements*. https://lwn.net/ml/linux-block/20240118180541.930783-1-axboe@kernel.dk/. 2024.

[7] Jeff Barr. "AWS Nitro SSD – High Performance Storage for your I/O-Intensive Applications." In: *AWS News* (2021). URL: https://aws.

amazon.com/blogs/aws/aws-nitro-ssd-high-performance-storage-for-your-i-o-intensive-applications/.

[8] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. "PLFS: a checkpoint filesystem for parallel applications." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 2009, pp. 1–12. DOI: 10.1145/1654059.1654081.

[9] Benjamin Berg et al. "The CacheLib Caching Engine: Design and Experiences at Scale." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020, pp. 753–768. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/berg.

[10] Janki Bhimani, Jingpei Yang, Zhengyu Yang, Ningfang Mi, N. H. V. Krishna Giri, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. "Enhancing SSDs with multi-stream: What? why? how?" In: *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. ISSN: 2374-9628. 2017, pp. 1–2. DOI: 10.1109/PCCC.2017.8280493.

[11] Ashish Bijlani and Umakishore Ramachandran. "Extension framework for file systems in user space." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 121–134.

[12] Matias Bjørling. "From Open-Channel SSDs to Zoned Namespaces." In: *Vault '19*. Boston, MA: USENIX Association, 2019.

[13] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs." In: *2021*

USENIX Annual Technical Conference (USENIX ATC 21). 2021, pp. 689–703.

[14] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs." In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 2021, pp. 689–703. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/bjorling.

[15] 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)*. Santa Clara, CA: USENIX Association, 2017, pp. 359–374. ISBN: 978-1-931971-36-2. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling.

[16] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. "The zettabyte file system." In: *Proc. of the 2nd Usenix Conference on File and Storage Technologies*. Vol. 215. 2003.

[17] Karolína Netolická Christoph Stanger. *Introducing Cloud Run volume mounts: connect your app to Cloud Storage or NFS*. https://cloud.google.com/blog/products/serverless/introducing-cloud-run-volume-mounts. 2024.

[18] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. "Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1683–1694.

ISBN: 9781450327589. DOI: 10.1145/2723372.2742798. URL: https://doi.org/10.1145/2723372.2742798.

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

[20] Intel Corp. *Intel® Optane™ Memory H10 with Solid State Storage*. https://ark.intel.com/content/www/us/en/ark/products/189613/intel-optane-memory-h10-with-solid-state-storage-intel-optane-memory-32gb-intel-qlc-3d-nand-ssd-512gb-m-2-80mm-pcie-3-0.html. 2019.

[21] Intel Corp. *syscall_intercept*. https://github.com/pmem/syscall_intercept. Retreived: 2017-03-20. 2023. (Visited on 06/01/2023).

[22] Viacheslav A. Dubeyko. *Linux Kernel Mailing List: [RFC PATCH 00/76] SSDFS: flash-friendly LFS file system for ZNS SSD*. https://lore.kernel.org/linux-fsdevel/20230225010927.813929-1-slava@dubeyko.com. 2023.

[23] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. "Enabling transactional file access via lightweight kernel extensions." In: *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*. San Francisco, CA: USENIX Association, 2009, pp. 29–42.

[24] *Fio - flexible I/O tester rev. 3.30*. URL: https://fio.readthedocs.io/en/latest/fio%5C_doc.html.

[25] Arun George. *CacheLib Pull Request #277: Adds support for Flexible Data Placement (FDP) over NVMe*. Accessed: November 15, 2023. 2023. URL: https://github.com/facebook/CacheLib/pull/277.

[26] Laura M. Grupp, John D. Davis, and Steven Swanson. “The Bleak Future of NAND Flash Memory.” In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. FAST’12. San Jose, CA: USENIX Association, 2012, p. 2.

[27] Pat Helland. “Immutability Changes Everything: We need it, we can afford it, and the time is now.” In: *Queue* 13.9 (2015), pp. 101–125.

[28] Ralf Hoffmann and Miklos Szeredi. *AVFS: A Virtual Filesystem*. https://avf.sourceforge.net/. Last accessed: August 29, 2023. 2001.

[29] Wookhan Jeong, Hyunsoo Cho, Yongmyung Lee, Jaegyu Lee, Songho Yoon, Jooyoung Hwang, and Donggi Lee. “Improving Flash Storage Performance by Caching Address Mapping Table in Host Memory.” In: *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, 2017. URL: https://www.usenix.org/conference/hotstorage17/program/presentation/jeong.

[30] Kanchan Joshi, Anuj Gupta, Javier Gonzalez, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Lund, and Jens Axboe. “I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux.” In: *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. Santa Clara, CA: USENIX Association, Feb. 2024, pp. 107–121. ISBN: 978-1-939133-38-0. URL: https://www.usenix.org/conference/fast24/presentation/joshi.

[31] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. “The Multi-streamed Solid-State Drive.” In: *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. Philadelphia, PA: USENIX Association, 2014. URL: https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang.

[32] Yangwook Kang, Yang-suk Kee, Ethan L. Miller, and Chanik Park. "Enabling cost-effective data processing with smart SSD." In: *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. 2013, pp. 1–12. DOI: 10.1109/MSST.2013.6558444.

[33] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. "Towards Building a High-Performance, Scale-in Key-Value Storage System." In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. SYSTOR '19. Haifa, Israel: Association for Computing Machinery, 2019, pp. 144–154. ISBN: 9781450367493. DOI: 10.1145/3319647.3325831. URL: https://doi.org/10.1145/3319647.3325831.

[34] KDE. *KDE heaptrack*. https://github.com/KDE/heaptrack. Last Accessed: January 15 2024. 2024.

[35] Tamás Koczka. *Learnings from kCTF VRP's 42 Linux kernel exploits submissions*. https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html. 2023.

[36] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. "F2FS: A New File System for Flash Storage." In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 2015, pp. 273–286.

[37] Jongsung Lee, Donguk Kim, and Jae W. Lee. "WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD." In: *Proc. VLDB Endow.* 16.11 (2023), pp. 2884–2896. ISSN: 2150-8097. DOI: 10.14778/3611479.3611495. URL: https://doi.org/10.14778/3611479.3611495.

[38] R. F. Ling. "On the theory and construction of k-clusters." In: *The Computer Journal* 15.4 (Jan. 1972), pp. 326–332. ISSN: 0010-4620. DOI: 10.1093/comjnl/15.4.326. eprint: https://academic.oup.com/comjnl/

article-pdf/15/4/326/1005965/150326.pdf. URL: https://doi.org/10.1093/comjnl/15.4.326.

[39] Simon AF Lund, Philippe Bonnet, Klaus BA Jensen, and Javier Gonzalez. "I/O interface independence with xNVMe." In: *Proceedings of the 15th ACM International Conference on Systems and Storage*. 2022, pp. 108–119.

[40] 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*. Vol. 2. Citeseer. 2007, pp. 21–33.

[41] Rino Micheloni, Alessia Marelli, and Kam Eshghi, eds. *Inside Solid State Drives (SSDs)*. en. Vol. 37. Springer Series in Advanced Microelectronics. Springer Singapore, 2018. DOI: 10.1007/978-981-13-0599-3. URL: http://link.springer.com/10.1007/978-981-13-0599-3 (visited on 04/06/2023).

[42] Peter Miller. *PlasticFS: A GNU Project*. https://plasticfs.sourceforge.net/. Last updated: 2012. 2012.

[43] Naoki Mizusawa, Kenji Nakazima, and Saneyasu Yamaguchi. "Performance evaluation of file operations on OverlayFS." In: *2017 Fifth International Symposium on Computing and Networking (CANDAR)*. IEEE. 2017, pp. 597–599.

[44] Damien Le Moal and Ting Yao. "Zonefs: Mapping POSIX File System Interface to Raw Zoned Block Device Accesses." In: *Vault '20*. Santa Clara, CA: USENIX Association, 2020.

[45] Myounghoon Oh, Seehwan Yoo, Jongmoo Choi, Jeongsu Park, and Chang-Eun Choi. "ZenFS+: Nurturing Performance and Isolation to ZenFS." en. In: *IEEE Access* 11 (2023), pp. 26344–26357. ISSN: 2169-3536. DOI: 10.

1109/ACCESS.2023.3257354. URL: https://ieeexplore.ieee.org/document/10070767/ (visited on 05/31/2023).

[46] Devashish Purandare, Pete Wilcox, Heiner Litz, and Shel Finkelstein. "Append is Near: Log-based Data Management on ZNS SSDs." In: *Conference on Innovative Data Systems Research 2022 (CIDR '22)*. 2022. URL: https://www.cidrdb.org/cidr2022/papers/p93-purandare.pdf.

[47] Eunhee Rho, Kanchan Joshi, Seung-Uk Shin, Nitesh Jagadeesh Shetty, Jooyoung Hwang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. "FStream: Managing Flash Streams in the File System." In: *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, 2018, pp. 257–264. ISBN: 978-1-931971-42-3. URL: https://www.usenix.org/conference/fast18/presentation/rho.

[48] Ohad Rodeh, Josef Bacik, and Chris Mason. "BTRFS: The Linux B-tree filesystem." In: *ACM Transactions on Storage (TOS)* 9.3 (2013), pp. 1–32.

[49] Mendel Rosenblum and John K Ousterhout. "The LFS storage manager." In: *Proceedings of the 1990 Summer Usenix*. 1990.

[50] 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*. 1991, pp. 1–15.

[51] Chris Sabol and Ross Stenfort. *Hyperscale Innovation: Flexible Data Placement Mode (FDP)*. 2022. URL: https://nvmexpress.org/wp-content/uploads/Hyperscale-Innovation-Flexible-Data-Placement-Mode-FDP.pdf.

[52] Chris Sabol, Ross Stenfort, and Mike Allison. "Flexible Data Placement: State of the Union." In: *Flash Memory Summit (FMS) 2023*. Google, Meta, Samsung. 2023.

[53] Dongjoo Seo, Ping-Xiang Chen, Huaicheng Li, Matias Bjorling, and Nikil Dutt. "Is Garbage Collection Overhead Gone? Case study of F2FS on ZNS SSDs." In: *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 2023.

[54] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. "zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO." In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, 2022, pp. 431–445. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/stamler.

[55] Theano Stavrinos, Daniel S. Berger, Ethan Katz-Bassett, and Wyatt Lloyd. "Don't be a blockhead: zoned namespaces make work on conventional SSDs obsolete." In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, 2021, pp. 144–151. ISBN: 978-1-4503-8438-4. URL: https://doi.org/10.1145/3458336.3465300 (visited on 08/26/2021).

[56] Jinghan Sun, Shaobo Li, Jun Xu, and Jian Huang. "The Security War in File Systems: An Empirical Study from A Vulnerability-centric Perspective." In: *ACM Transactions on Storage* 19.4 (2023), pp. 1–26.

[57] Adam Sweeney. "Scalability in the XFS File System." In: *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*. San Diego, CA: USENIX Association, 1996. URL: https://www.usenix.org/conference/usenix-1996-annual-technical-conference/scalability-xfs-file-system.

[58] Billy Tallis. *The Intel SSD 660p SSD Review: QLC NAND Arrives For Consumer SSDs*. https://www.anandtech.com/show/13078/the-intel-ssd-660p-ssd-review-qlc-nand-arrives. 2018.

[59] Bart Van Assche. "[PATCH v2 00/15] Pass data temperature information to UFS devices." In: *Linux Patch Mailing List* (2023). URL: https://lore.kernel.org/linux-block/20231005194129.1882245-1-bvanassche@acm.org/.

[60] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. "To FUSE or not to FUSE: Performance of User-Space file systems." In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 2017, pp. 59–72.

[61] Robin Verschoren and Benny Van Houdt. "On the Impact of Garbage Collection on Flash-Based SSD Endurance." In: *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 16)*. Savannah, GA: USENIX Association, Nov. 2016. URL: https://www.usenix.org/conference/inflow16/workshop-program/presentation/verschoren.

[62] Stephen R Walli. "The POSIX family of standards." In: *StandardView* 3.1 (1995), pp. 11–17.

[63] 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*. 2014, pp. 1–14.

[64] Joel Wejdenstal. *dashmap: A High-Performance Concurrent Hash Map for Rust*. https://github.com/xacrimon/dashmap. Accessed: September 21, 2023. 2023.

[65] Louis Woods, Zsolt István, and Gustavo Alonso. "Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading." In: *Proc. VLDB Endow.* 7.11 (2014), pp. 963–974. ISSN: 2150-8097. DOI: 10.14778/2732967.2732972. URL: https://doi.org/10.14778/2732967.2732972.

[66] NVM Express Workgroup. *NVM Express Zoned Namespaces Command Set 1.1*. 2021. URL: https://nvmexpress.org/developers/nvme-command-set-specifications/.

[67] 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." en. In: *ACM Transactions on Storage* 13.3 (2017), pp. 1–26. ISSN: 1553-3077, 1553-3093. DOI: 10.1145/3121133. URL: https://dl.acm.org/doi/10.1145/3121133 (visited on 05/31/2023).

[68] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. "Don't Stack Your Log On My Log." In: *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*. Broomfield, CO: USENIX Association, 2014. URL: https://www.usenix.org/conference/inflow14/workshop-program/presentation/yang.

[69] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S. Kim. "Overcoming the Memory Wall with CXL-Enabled SSDs." In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, 2023, pp. 601–617. ISBN: 978-1-939133-35-9. URL: https://www.usenix.org/conference/atc23/presentation/yang-shao-peng.

[70] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. "SPDK: A development kit to build high performance storage applications." In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2017, pp. 154–161.

[71] Qinghua Ye and Kapil Karkra. *Cloud Storage Acceleration Layer (CSAL): An Open-Source, Host-Based Flash Translation Layer (FTL)*. https://www.snia.org/educational-library/cloud-storage-acceleration-layer-csal-enabling-unprecedented-performance-and. Last accessed: August 29, 2023. 2023.

[72] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. “FlashKV: Accelerating KV performance with open-channel SSDs.” In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (2017), pp. 1–19.

[73] Yuhong Zhong et al. “XRP: In-Kernel Storage Functions with eBPF.” In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, 2022, pp. 375–393. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/zhong.