msF2FS:

Design and Implementation of an NVMe ZNS SSD Optimized F2FS File System

Nick Tehrany

msF2FS:
Design and Implementation of an NVMe ZNS SSD Optimized F2FS File System

by

Nick Tehrany

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday, March 23rd, 2023, at 12:00 PM.

Student number:	5411726	
Thesis committee:	Prof. dr. ir. Alexandru Iosup,	TU Delft and VU Amsterdam, chair and daily supervisor
	Prof. dr. ir. Fernando A. Kuipers	TU Delft, committee member
	Dr. ir. Animesh Trivedi	VU Amsterdam, daily co-supervisor

TU Delft

Abstract

The ongoing digitalization of the world, estimated to reach a yearly data generation of 200 Zettabytes by 2025, is putting increasing pressure on system developers to provide systems capable of scaling with future needs. Of particular importance are the data storage systems, providing the means of storing and retrieving the vast amounts of data. One widely adopted storage technology, predicted to become the leading media for future data storage, is flash-based solid state drives (SSD). The complex architecture of flash SSD however introduces several challenges, such as necessary garbage collection, for managing the flash storage. To readily integrate flash SSD into storage systems, the flash management idiosyncrasies are hidden inside the storage device. The hiding of the flash management idiosyncrasies has however been identified to have significant performance implications. As a result, numerous efforts have pushed towards more open flash storage interfaces, with the most recent addition of Zoned Namespace (ZNS) flash SSD. ZNS SSD presents a unique opportunity for storage software and the flash SSD to coordinate the flash management responsibilities.

While the open flash storage interface of ZNS presents a plethora of opportunity in software optimization, current software support is in its early stages, leaving much of its potentials yet to be explored. In this work, we present msF2FS (multi-streamed F2FS), a file system with optimized ZNS integration, based on the de facto standard flash file system F2FS. msF2FS enhances the ZNS integration by leveraging the parallelism capabilities of ZNS SSD, and increasing the coordination between the file system and applications for data placement decision-making. The data placement coordination between file system and application reduces the suboptimal data placement decisions made by the file system. Evaluations of msF2FS show the benefit of the application and file system coordination, with the RocksDB application achieving up to 23.19% higher throughput as a result of optimized data placement. We make all developed code of msF2FS publicly available at `https://github.com/nicktehrany/msF2FS`.

Preface

This thesis work concludes the years of my master's degree at the Delft University of Technology. During the recent years of studying at the TUDelft, I have been able to significantly deepen my knowledge and understanding of systems, and particularly storage systems. In addition to theoretical knowledge gained, I had the opportunity to be part of the AtLarge Research group, conducting storage systems research, speaking at national conferences, and submitting research articles at prestigious venues in collaborations with leading storage technology experts. Beyond graduating from a master's degree at a distinguished university, the experiences of the past years have shaped my future academic interests and prospects, paving the beginning of my academic career.

As a result this thesis work has not been solely my independent work, but has included guidance, feedback, and comments from numerous people, whom I like to express my gratitude towards. I am particularly grateful to Animesh Trivedi, who has not only supervisor this thesis work, but who has originally introduced me to storage systems and computer science research. Similarly, I would like to thank Alexandru Iosup, who has supervised my work alongside Animesh Trivedi over the past years, and who have shaped my experiences in computer science research. I am very grateful for all the lessons I have learned, opportunities I have had as a result, and the guidance, support, and inspiration from Animesh Trivedi and Alexandru Iosup. Furthermore, I would like to extend my gratitude towards Fernando Kuipers, who agreed to be part of the graduation committee for this thesis work.

I would also like to thank the ZNS team at Western Digital (WD), who made this work possible with donations. In particular, Matias Bjørling has provided continued support, feedback, and comments throughout the evaluation of ZNS devices. Lastly, I would like to express my gratitude towards the entire team at the AtLarge Research group, which have provided a welcoming and engaging experience of computer systems research. Frequent discussions and continuous feedback on my progress have lead to the work of this thesis, and have provided a joyful and educational time over the past years of collaboration. I especially want to thank Krijn Doekemeijer for the continued cooperation in storage research, the extensive feedback, comments, suggestions provided throughout this work, and for aiding in the carbon footprint calculations used in this thesis work.

Nick Tehrany
Delft, March 15, 2023

Contents

1 Introduction

The ongoing digitalization of the world, generating vast amounts of data is estimated to reach a yearly 200 Zettabytes [211] of data being generated, stored, and processed. Large contributing factors for these vast amounts of data are social media [68, 83], *Internet of Things (IoT)* devices and edge computing [10, 272], for which predictions estimate a total of 75 Billion connected devices by 2025 [23], and the increasing adoption of big-data [9, 97, 178, 316] and cloud services [6, 247]. Putting the importance of *Information and Communications Technology (ICT)* into perspective, its presence in the Netherlands is responsible for generating a total of 3.3 million jobs, and contributes to 60% of the total GDP [111].

One indispensable component of the ICT infrastructure is provided by storage systems, enabling the storage and retrieval of all the generated data. Generated data is stored on storage devices through various interfaces, all relying on *read* operations to retrieve prior stored data, and *write* operations to write new data to the storage device. Figure 1.1 illustrates the interaction of big-data with storage systems through read and write operations, going to different types of storage devices. The most widely adopted storage device is the *Hard Disk Drive (HDD)* [7, 56] (commonly also referred to as *disk*), which due to its low cost and high storage capacity has presented an appealing solution to storing large amounts of data. However, HDD suffers from low performance due to physical limitations of the drives themselves, relying on mechanical movement on the device in order to read and write data. As a result, HDD suffers from increased latency for random accesses [62, 122] and high power consumption [34, 89].

An alternative storage technology that is gaining increasing adoption is the *Solid State Drive (SSD)*, projected to deliver 30% of the created storage media from 2017-2025 [245]. SSD is most commonly constructed with *flash cells*, resulting in such SSDs commonly being referred to as *flash SSD*. Outperforming HDD, flash SSD has the capability of providing μ-second access latency, resulting in several GB/s bandwidth with *I/O Operations per Second (IOPS)* (read and write operations) in the order of millions [109, 252]. However, flash SSD remains more expensive than HDD, and has slightly lower storage capacity. Increasing demand on faster storage devices is pushing SSD innovation to provide increasing capacities and lower device cost. Predictions therefore estimate that the largest growth in storage devices being used in the future will be flash-based SSD [245].

1.1. Challenges of Flash SSD

While flash SSD provides significant performance gains over conventional HDD storage, it introduces its own challenges. Particularly, the architecture of flash SSD lacks support for in-place updates, requiring to explicitly erase existing data prior to being able to overwrite it. A common approach to dealing with the flash-specific constraints is to hide these in a part of the device firmware, called the *Flash Translation Layer (FTL)*, to provide a flash SSD that seemingly supports in-place updates. The utilization of a FTL simplifies the integration of flash SSD, however the internal management idiosyncrasies introduce several challenges. Primarily, the lack of in-place updates for flash cells introduces the process of *Garbage Collection (GC)*. Over time, as more data is being overwritten, old data is simply marked as invalid, with the new data being written in a free space. This results in invalid and valid data being interleaved across the storage space. When the flash SSD runs low on free space, it must take the still valid data that is interleaved with invalid data, move it to a new space, followed by erasing all old data and reclaiming the space. This process is referred to as *Garbage*

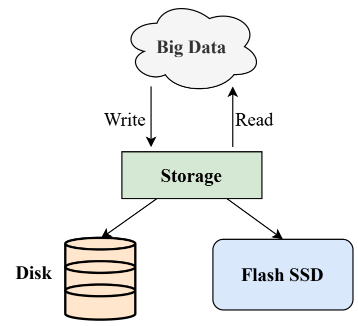


Figure 1.1: Big data storing and retrieving data from various storage devices through *read* and *write* operations.

Collection (GC).

The process of GC therefore introduces several performance overheads. Firstly, the performance becomes unpredictable [151, 310], due to the FTL deciding when to run GC. Secondly, the operation latency can increase significantly [55], as there is no indication into how much valid data must be moved during the GC operation, constituting its high latency. Lastly, the lifetime of flash cells is also limited, causing the flash to wear down over time, as it is being written and erased. Therefore, the introduced GC continuously moving around valid data decreases the lifetime of the flash cells, and therefore the flash SSD lifetime [98].

1.2. File Systems For Flash SSD

In order to integrate flash-based storage devices into systems, developers have turned to flash-friendly data structures, such as log-based designs relying on append-only data writing, matching the characteristics of flash storage. With file systems remaining a ubiquitous method for storing large amounts of data by managing the storage media and exposing a file addressable abstraction, file system design has commonly focused on log-based data structures, referred to as *Log-Structured File System (LFS)*. The de facto standard LFS for flash-based storage devices is *Flash-Friendly File System (F2FS)*, which is widely adopted and present in the Linux kernel.

Inherent to log-based data structures, F2FS must run GC, much like the FTL on the flash SSD, in order to erase invalid data and free space. Therefore, design choices within F2FS particularly aim at reducing the GC overheads for which particular mechanisms are introduced. Specifically, F2FS utilizes data grouping based on the data lifetime. By grouping data with similar lifetime together, GC can be reduced as the data is erased in close proximity. Reducing the interleaving of valid and invalid data decreases the need for GC to move the valid data. Such grouping is implemented in F2FS by separating data into three lifetime classifications (hot/warm/cold), which are mapped to three different logs, one log for each lifetime classification. The utilization of multiple logs furthermore allows to decrease the *fragmentation*, where data belonging to a single file is broken into non-consecutive fragments that are spread out across the storage space. Fragmentation is a major contributing factor to not only GC overheads, but additionally affecting the read performance [144].

The utilization of different lifetime classifications for data similarly introduces coordination between storage stack layers. Within a conventional Linux storage stack, illustrated in Figure 1.2a, applications interact with the file system by reading/writing to files, where the file system, running in kernel space, uses the Linux block I/O layer to interact with the storage device. With applications having the highest degree of knowledge on the access patterns it generates for its files, passing data lifetime information to the file system through *hints*, allows the file system to enhance the data placement decision with the knowledge of the application, benefiting in the management of GC overheads.

1.3. Zoned Namespace SSD

While LFS aim to provide enhanced integration with flash GC by aligning its data structures to the flash characteristics, such approaches fail to properly integrate flash storage by duplicating GC, with device- and file system-level GC, and unaligned data structures that do not match all flash devices [310]. The issues arising of flash storage hiding management idiosyncrasies is commonly referred to as the *semantic gap* between the storage device and storage software [321], where interfaces to access and manage the storage device fail to represent and align with the flash-specific characteristics. By decreasing FTL responsibility and increasing control for storage stack software, efforts such as *Software-Defined Flash (SDF)* [218] and *Open-Channel SSD*

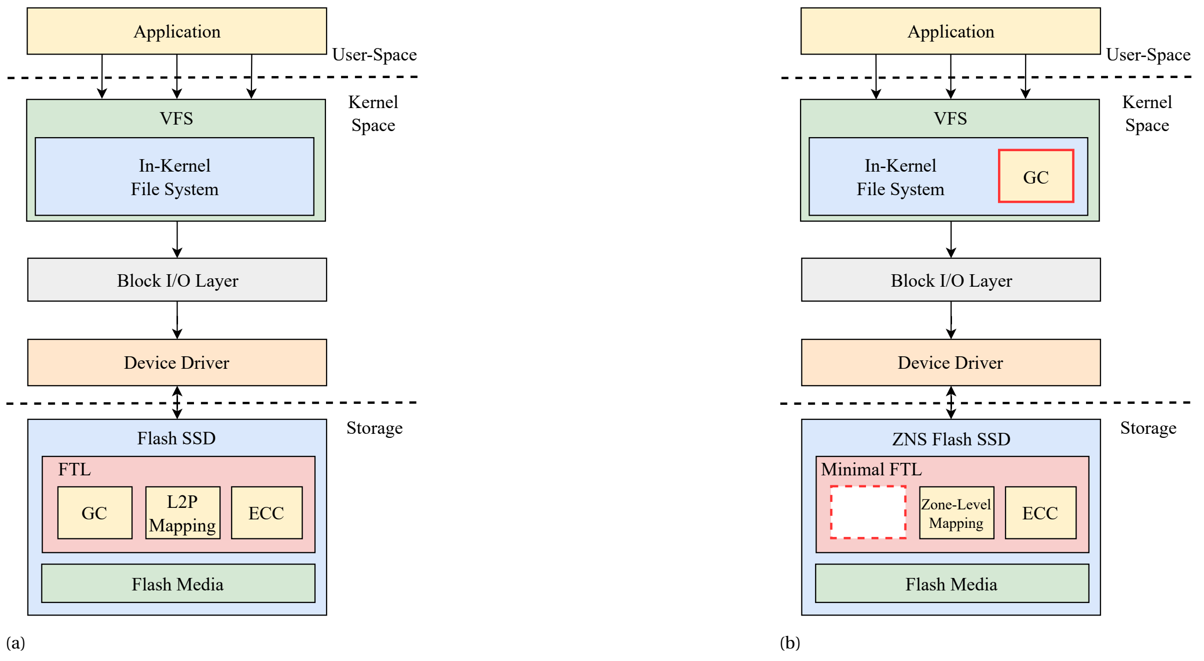


Figure 1.2: Comparison of the storage stack integration for (a) conventional flash SSD, and (b) ZNS SSD which shifts GC responsibility from the device FTL to the host software.

(OCSSD) [19, 235] attempted to reduce the semantic gap. However, these efforts failed to gain large scale adoption due to the lack of standardization, resulting in device-specific implementations, and complex interfaces, requiring software developers to have extensive knowledge of flash in order to build flash-specific software.

The most recent addition of a flash interface is that of *Zoned Namespace (ZNS)*, standardized in the *Non-Volatile Memory Express (NVMe)* 2.0 base specification (released in June 2021) [20, 301]. Learning from shortcomings of prior attempts, its standardization and simplified interface makes it an appealing solution for better integrating software with flash-based storage devices. With ZNS, the storage device is exposed as a number of *zones*, which are required to be written sequentially in an append-only manner, and a zone must be reset in order to be written again, mirroring the flash characteristics. As a result, the responsibility of GC is pushed from the storage device to the storage software, requiring the software to manage valid and invalid data in zones and issue reset commands to reset zones. Figure 1.2 compares the integration of (a) a conventional flash SSD into the Linux storage stack to (b) the ZNS SSD integration into the storage stack. By reducing the FTL responsibility, ZNS SSD pushes the device-level GC management to the storage software, such as the kernel file system.

1.4. Problem Statement

The standardization of ZNS has quickly led to a plethora of research activity and development [8, 13, 18, 20, 52, 94, 130, 169, 199, 239, 262, 268, 324]. One of the first integrations of ZNS is the F2FS file system [20, 167]. Leveraging the ZNS interface in F2FS, the possibility for concurrently accessing numerous zones matches its data structure of writing multiple logs, for the different lifetime classifications. Furthermore, the integration of ZNS into F2FS clearly defines data grouping with the ZNS zones, enhancing the data grouping abilities to reduce GC overheads. However, the current ZNS integration into F2FS focuses on functionality, instead of optimal utilization of the new ZNS zone interface, leaving several integration possibilities yet to be exploited.

(1) The current ZNS integration of F2FS utilizes a maximum of six concurrently active zones, whereas common ZNS devices are expected to support up to 32 active zones at a time [20]. Leaving the majority of the resources unutilized, expanding the usage of concurrently active zones by F2FS allows leveraging a higher degree of available resources. **(2)** The ZNS interface pushes the responsibility of GC to F2FS, however with applications running on top of F2FS, this interface is lost by F2FS failing to adequately provide ZNS management to the application. Such integration would allow the application to coordinate data placement and GC with the file system. The existing hint interface to provide lifetime classification for files utilizes a maximum of 3 classifications (hot/warm/cold), whereas the capabilities of ZNS devices allow for significantly more active zones to be utilized in the data grouping. **(3)** With the file system currently making all data placement decisions, there are no currently available tools that provide understanding of the on-device data placement for ZNS devices. The lack of such tools to understand file system utilization of storage devices fails to provide insight into shortcomings and suboptimal file system data placement decisions, as well as giving an understanding into the utilization of ZNS in F2FS.

Addressing these challenges has several possible implications. Firstly, it leads to better understanding of F2FS utilization of the new ZNS devices. With increased understanding, future improvements and enhancements on the ZNS integration become apparent. Secondly, enhancing file data placement on the storage, by utilizing application-guidance in data placement, leads to increased utilization of the flash storage. Optimized flash storage utilization is beneficial for large adopters, such as data centers, increasing the flash storage lifetime and decreasing data center cost to replace worn out flash devices. Lastly, with ZNS SSD pushing to reduce the semantic gap between the storage device and storage software, a similar push to reduce the semantic gap between the various layers in the storage software stack is necessary to enhance the utilization of storage devices, furthering the needs for a sustainable future of storage systems, capable of scaling with future storage demands.

1.5. Research Questions

In order to evaluate the proposed problem statement, we identify four key *Research Questions (RQs)* to assess the current state of storage software for flash, changes arising from the standardization of ZNS, and how to enhance coordination of storage software across its layers, from the device over the Linux Block I/O layer to the application.

RQ1. How has flash storage influenced design choices, challenges, and opportunities in file systems development?
With the increasing adoption of flash-based storage technology and its own unique characteristics, it is important to identify how flash has shaped the storage software development in addition to log-based data structures. The aim of this research question is to identify algorithms, data structures, and mechanisms for leveraging the potentials of flash-based storage, and identify the obstacles that storage software encounters with flash-based devices. Understanding these challenges allows evaluating the benefits of the newly standardized ZNS interface, and identify newly arising challenges of ZNS for storage software development.

RQ2. How to identify and leverage application-provided hints to improve performance of flash file systems for data management?
With ZNS devices reducing the semantic gap between the storage device and file systems, similar efforts are required to reduce the semantic gap between the file system and applications. By utilizing the existing infrastructure on applications providing lifetime hints to the file system, one solution is to extend this interface by leveraging the ZNS management capabilities to present the application with additional management hints to be used in file system data allocation. This research questions aims to answer what type of hints applications can pass to the file system to improve data placement decisions by the file system, how such hints can be passed to the file system, and how the file system can utilize these hints in the data placement.

RQ3. What is the quantitative impact of leveraging a larger number of concurrent zones in msF2FS with multiple streams?
The ZNS interface allows the device to concurrently utilize a maximum number of zones, whereas the current F2FS implementation does not fully exploit all these active zones. This research question aims to identify the impact of increasing the concurrently accessed zones by F2FS by increasing the logs for each lifetime classification, with a configurable number of *streams* for each log. Such a solution extends the existing log data structure to map to the ZNS characteristics, allowing to leverage its capabilities in concurrency, maximizing the achievable performance with a larger number of active zones.

RQ4. How to simplify and visualize operational data in the context of F2FS and ZNS devices as currently there do not exist any tools?
File systems are responsible for data placement of application files. While files have possible hints,

such as hot and cold lifetime identification providing improved data grouping, the file system decides on the data placement. With such a placement the file system generates particular I/O traffic to the ZNS device and the different zones. This research questions aims to understand how information on the file system data placement decisions can be collected, and represented in a visually comprehensive manner. Such tools allow for firstly, identifying I/O patterns to showcase possible unsuitable data structures and I/O characteristics, and secondly aiding the better understanding of F2FS data placement decisions and identifying the utilization of the ZNS zones.

1.6. Research Methodology

For answering each of the defined RQs, we follow appropriate and state-of-the-art practices, which we define as follows, identified with the *Research Methodology (RM)* definition, with matching number identifier for their respective RQ it covers.

RM1. Identifying how flash storage has influenced the development of file systems requires a detailed literature study for which we utilize the **Systematic Literature Review approach** [142, 155] and quantitative research [145, 177]. A systematic literature review requires a review protocol that defines the selection criteria of appropriate studies. We detail the review protocol we devise for this literature study in Section 3.2.

RM2. Integration of application-provided hints follows state-of-the-art practices using **design, abstraction, prototyping** [93, 110, 233], based upon which we synthesize requirements for the hint management interface from the file system and application level, and provide a running prototype with hint support and an example application with hint integration. Evaluation of the hint integration in the file system and application follow state-of-the-art practices in **experimental research, designing appropriate micro and workload-level benchmarks, and quantifying a running system prototype** [101, 117, 217].

RM3. The implementation of leveraging a larger number of concurrent ZNS zones in msF2FS follows state-of-the-art practices using **design, abstraction, prototyping** [93, 110, 233], followed by an evaluation for which we similarly utilize micro- and macro- benchmarks, following the state-of-the-practice in **experimental research, designing appropriate micro and workload-level benchmarks, and quantifying a running system prototype** [101, 117, 217].

RM4. Collecting, representing, and understanding operational data of F2FS for ZNS similarly relies on the state-of-the-art practices using **use-case study, collecting operation traces, and collaborating with users and partners** [156, 165, 291].

RM5. All efforts within this thesis work, including developed source code, benchmarking scripts, and collected data follow the state-of-the-art practices to provide **open science, open-source software, community building, peer-reviewed scientific publications, and reproducible experiments** [15, 70, 285, 298].

1.7. Research Contributions

With the four key RQs and detailed research methodology, the contributions of this thesis are sixfold, and detailed as follows.

RC1. We provide a detailed literature study of the effects of flash storage on file system design, depicting data structures, algorithms, mechanisms, and commonly applied techniques for enhancing their integration. This literate study is presented in Chapter 3.

RC2. We build a prototype implementation *Multi-Streamed F2FS (msF2FS)*, based on F2FS, in which we design a new abstraction, a **Stream**, which leverages the ZNS device zones by mapping a stream to a zone, and allocating data concurrently to different streams. We design and implement various stream allocation policies, and the msF2FS prototype integrates the proposed hint enhancement for application and file system coordination in data placement.

RC3. We provide a detailed empirical evaluation of added features in the process of **RQ2**, showcasing the implications of using multiple concurrent data streams in msF2FS, passing of application-specific

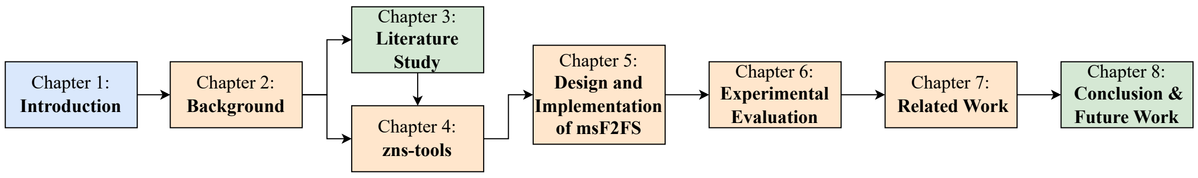

Chapter 1: Introduction
Chapter 2: Background
Chapter 3: Literature Study
Chapter 4: zns-tools
Chapter 5: Design and Implementation of msF2FS
Chapter 6: Experimental Evaluation
Chapter 7: Related Work
Chapter 8: Conclusion & Future Work

Figure 1.3: Structure of the reaming thesis content. The literature study and further contributions of this thesis work server as standalone content, and can therefore be read independently according to this structure.

hints to the file system, and how these additions affect file system and application performance, as well as changes in the utilization of the flash storage.

RC4. We provide a set of tools for understanding F2FS usage of ZNS devices, which allow for identifying the on-device locations of a file, mapping the data layout of F2FS files to ZNS zones, and a tracing framework that records I/O activity for ZNS devices and generates visuals of various operational aspects.

RC5. In an effort to provide reproducible results, and allow future work on the contributions of this thesis work, we make all the developed source code for msF2FS (**RC2**, `https://github.com/nicktehrany/msF2FS`), collected data and benchmarking scripts (**RC3**, `https://github.com/nicktehrany/msF2FS-bench`), as well as the set of tools (**RC4**, `https://github.com/nicktehrany/zns-tools`) publicly available. Furthermore, we provide detailed instructions on additional bootstrapping and setup commands in Appendix A.

RC6. This thesis work has been accepted at the ICT.open 2023 conference. A part of this work is already under submission at SIGMETRICS 2023. We plan to write further papers on the literature study of design choices for flash-specific file systems (**RC1**), the design and evaluation of msF2FS (**RC2** and **RC3**), and the developed set of analytical tools (**RC4**).

1.8. Research and Societal Relevance

The standardization of ZNS fundamentally shifts from manufactures dictating storage device operations, to allowing host system software to be part of the device coordination. With this thesis, we propose the design and evaluation of (1) better file system utilization of ZNS resources, and (2) integration of application-guided data placement, aiming at increasing the storage software layer coordination by including application decisions in the data allocation policies of the file system. The findings of this thesis work are therefore not limited to storage systems, but rather apply to the larger area of systems, which heavily relies on layered design approaches to manage system complexities. With the grand challenge for future storage systems, as projected by the CompSys Manifesto [111], focusing on the better leveraging of storage system capabilities with optimized interfaces, and the combining of storage stack layers, the addition of ZNS storage is a fundamental shift for the storage hierarchy towards this future.

Providing a case example of better software coordination in storage systems aims at showing possibility to enhance system software as a whole. Beyond the relevance to systems research and design, showcasing better utilization of storage devices is a significant improvement for the consumers of flash storage. Benefits include a decrease in the underutilized and wasted resources [275], an increase in efficiency [94], which lowers the running and repair, or replacement, costs for data centers [314]. Aligning with current societal push towards increased energy efficiency and greener system software, improving device utilization and file system performance allows for possible reduction in energy consumption [26], particularly from lowered write amplification and less flash wear [164, 290]. Such improvements are essential to build systems that are capable of scaling with future demands on data generation, and adhering with the push towards a more sustainable future.

1.9. Thesis Structure

This thesis contains two main parts, the literature study and the design and evaluation of msF2FS, which can be read independently. Therefore, Figure 1.3 depicts the structure of the thesis. Firstly, Chapter 2 establishes the background knowledge required for the literature study and the remainder of this thesis. Next, Chapter 3 presents the literature study conducted during this thesis work. It serves as a standalone survey, and can therefore be read independent of the latter chapters of this thesis. Chapter 4 presents the design and implementation of the *zns-tools*, followed by a discussion of the existing ZNS integration in F2FS and the design and implementation of msF2FS in Chapter 5. Next, we provide an extensive evaluation of msF2FS in Chapter 6, followed by related work and the conclusion with future work prospects in Chapter 7 and Chapter 8, respectively. We furthermore provide any additional code snippets and setup commands utilized during the evaluation in Appendices A to C.

2
Background

The increasing adoption of flash-based SSD [54, 172], due to its higher performance capabilities compared to conventional HDD, and decreasing cost is making it a ubiquitous storage media for storage systems. In this chapter we explain the high-level concepts of the construction of flash-based SSD (Sections 2.1 to 2.4), technical details of the newly standardizes ZNS SSD (Section 2.5), and how the de facto standard file system F2FS is designed for flash storage (Section 2.6).

2.1. Flash Storage Building Block: Flash Cells

Flash storage is based on *flash cells* providing the persistent storage capabilities through programming of the cell and erasing it to clear its content. Each flash cell is constructed with a floating gate, which can hold an electrical charge, and a control gate, inside a *Complementary Metal-Oxide Semiconductor (CMOS)* transistor [220]. Using the electrical charge present inside the flash cell, each cell is capable of representing a logical value (0 or 1). Such flash cells, capable of representing a single bit, are called *Single-Level Cell (SLC)*. Data is written to flash cells by *programming* the cell. Updating existing data in flash cells requires the cell to be *erased*, followed by being programmed. In addition to SLC, other types of flash cells are available, ranging from *Multi-Level Cell (MLC)* (2 bits per flash cell) to *Penta-Level Cell (PLC)* (5 bits per cell) [198]. To represent more bits with a single flash cell, the charge inside the flash cell is divided into a respective number of ranges in order to represent the number of bit combinations (e.g., 4 ranges to represent all combinations of 2 bits in MLC).

Increasing the number of levels in flash cells however results in slower access time [7], increased wear on the flash cells and a lower lifetime [210, 255], as the cells erode over time with it being programmed. The wear on flash cells is accelerated with more bits being stored in the cell (i.e., increasing the cell level). For SLC the number of program/erase cycles is typically 100K, and decreases to 10K or less for MLC [255]. As a result enterprise flash SSD most commonly employ SLC for increased reliability and lifetime.

Similarly, the organization of flash cells dictates the resulting type of storage device. Flash cells can be organized in the form of *NOR* or *NAND*, representing the connection architecture of the cells. With NOR flash, the resulting flash is byte-addressable, whereas NAND flash provides a page-addressable structure, with a page representing the unit of read and write. As a result NOR flash is commonly applied in *Basic I/O System (BIOS)* [161], however NAND architecture is suitable for mass data storage, making it the primary architecture for flash SSD. Throughout this thesis we focus purely on NAND flash, however for more information on flash cell architecture consult reports such as [14] and [50].

2.2. Building Mass Storage SSD with Flash Cells

Utilizing the building block of flash cells, building mass storage devices is achieved by grouping together several flash cells into *pages* (typically 8-16KiB in size), depicting the unit of access (read/write). A flash page additional has a small extra space, called the *Out-Of-Band (OOB)* area, which is utilized for *Error Correction Codes (ECC)* and small metadata. As the operation for erasing of a flash page is costly [88], multiple pages are grouped into a *block* (several MiB in size), which is the unit of erase. Increasing the erase unit from individual pages to a block helps amortize the erase overheads. Pages within a block have to be written sequentially and prior to a page being overwritten, the entire block must be erased. A number of blocks are then grouped

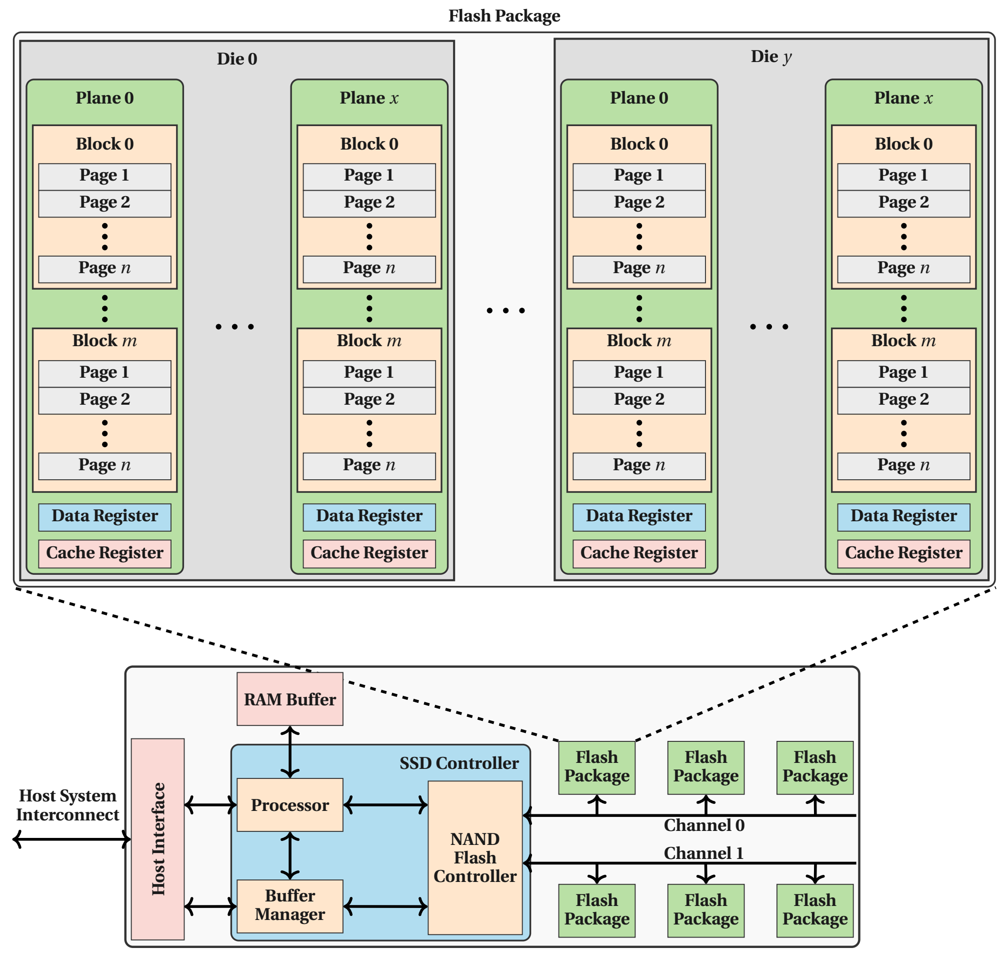


Figure 2.1: Internal architecture of an SSD, with n pages in a block, m blocks in a plane, x planes in a die, and y dies in a flash package. The example SSD has two channels however any architecture with various numbers of channels is possible. Adapted from [2, 85].

into a *plane*, of which multiple planes are combined into a *die* [1]. Lastly, numerous chips are combined into a single flash package. Figure 2.1 shows a visual representation of a flash-based SSD architecture.

In order to build full storage devices, multiple flash packages are packed together into a SSD. While a SSD can be constructed with any solid state technology such as Optane [303, 309], we focus purely on NAND flash based SSD. The SSD contains a controller, which is responsible for processing requests, managing data buffers, and contains the NAND flash controller that manages the flash packages. An additional *Random Access Memory (RAM)* buffer, most commonly in the form of *Dynamic RAM (DRAM)*, is present on the device for maintaining address mappings. Lastly, a SSD contains the host interface that provides the means to connect the storage device to the host system over connection interfaces such as *Serial Advanced Technology Attachment (SATA)* and *PCI Express (PCIe)*, and defining standards such as *Advanced Host Controller Interface (AHCI)* and *Non-Volatile Memory Express (NVMe)* [163]. NVMe is the interface specification particularly designed for fast SSD devices, capable of outperforming legacy protocols such as SATA with 8x higher performance [306], due to its increased number of I/O queues to which requests can be issued in parallel. Similarly, PCIe protocol achieves the highest throughput and lowest latency compared to SATA [72]. As a result, flash SSD are commonly connected to systems through NVMe over PCI.

2.3. Increasing Flash SSD Performance

Given the architecture of flash SSD, the it contains a large degree of possible parallelism (i.e., multiple channels, flash packages, dies, and planes). Furthermore, based on a study on a Samsung SSD, the bandwidth

[1] *Dies* are also referred to as *chips*, we use the terms interchangeably and mean the same entity.

of NAND flash *Input/Output (I/O)* buses, connecting the flash to the flash controller, is limited to 32MB/s because of physical restrictions, and 40MB/s with interleaving in dies [2]. A write operation to flash storage firstly writes the data to a data register, from which the data in the register is then programmed into the flash page. Because the programming operation takes longer than loading of data, these operations can be interleaved, such that while a page is being programmed, data of another write operation is loaded [152]. This avoids stalling whenever a page programming finishes and data needs to be loaded again, which is fundamentally similar to the concept of CPU pipelining [277].

In order to increase performance, parallelism is utilized on the different flash entities in the SSD [36]. Depending on the hardware configurations, the different types of parallelism are referred to as *channel-level parallelism*, where requests are distributed individually across the flash channels, *package-level parallelism* where flash packages on the same channel are accessed in parallel, *chip-level parallelism* where flash packages with multiple chips access these in parallel, and *plane-level parallelism* where the same operation can be run on multiple planes, on the same chip, in parallel. A commonly applied technique for enhanced parallelism is the utilization of *clustered blocks*, where requests are issued to blocks in parallel to different chips and planes.

There are several additional performance optimizations for accessing flash storage. Other than accessing a flash entity in parallel and operation interleaving, flash packages can be accessed synchronously with a single request queue through *ganging* [2]. Flash packages within a gang share the same control bus from the flash controller, however can utilize different data buses, thus allowing a single command request queue with multiple data buses to provide the resulting data. Therefore, requests that require data from multiple pages can be split among a gang from a single request queue, and provide the data on different buses, avoiding the bottleneck of a single data bus.

2.4. Hiding Flash Management Idiosyncrasies on the SSD

With the flash characteristic requiring sequential writing within blocks, and lacking support for in-place updates, flash SSD employs a *Flash Translation Layer (FTL)* to hide these management idiosyncrasies, and provide seemingly in-place updates. As a result of the sequential write constraint, if data inside a flash page is updated, the page is simply marked as invalid and the new data is appended to a new a flash page, possibly in the same block. The FTL is responsible for managing flash page information on their validity and its mappings to *Logical Block Address (LBA)*, which is the mechanism for storage software to address the storage device.

Different implementations of a FTL can utilize different mapping levels, such as block- and page-level mappings. A naive FTL design is to maintain a fully associative mapping of each *Logical Block Address (LBA)* to every possible *Physical Block Address (PBA)* [88], referred to as *Logical-to-Physical (L2P)* mapping, and inversely *Physical-to-Logical (P2L)* mapping. These mappings are maintained in a *mapping table*, which is kept in the RAM of the flash device for faster accesses. For consistency the mapping table is also maintained in the persistent flash storage, where on startup time of the devices the mapping table is reconstructed in the device RAM [2, 35]. With this, the SSD can provide seemingly in-place updates by simply writing data to a free page and invalidating the overwritten data. Further details on FTL mappings are not required for the remainder of this thesis, however for a detailed explanation of FTL mapping algorithms consult [44] and [88].

As over time an increasing number of flash pages become invalid due to data updates, blocks contain valid and invalid data, requiring the FTL to run *Garbage Collection (GC)*. During GC, the FTL selects a block from which it reads the still valid flash pages, writes these to an empty block, followed by erasing of the original block. The process of GC requires the device to provide an empty space that cannot be used as storage, but is only used for the FTL to move valid pages, referred to as the *Over-Provisioning Space (OPS)*. If a device has fully utilized all its capacity, the FTL must be able to write out valid pages, which the overprovisioning space serves at. Therefore, SSD commonly ship with an overprovisioning space of 10-28%, which is only usable by the FTL. In addition to GC, the FTL is responsible to ensure even wear across flash cells, as a flash cell has a limited number of program/erase cycles. This process is referred to as *Wear Leveling (WL)*, where the FTL ensures the flash wears out evenly across the entire device, and no particular parts are burnt out faster than others.

2.5. Zoned Namespace SSD

While the FTL provides the seamless integration of flash SSD into storage systems, its unpredictable performance as a result of GC [151, 310] has shifted the research community to new flash interfaces that expose flash

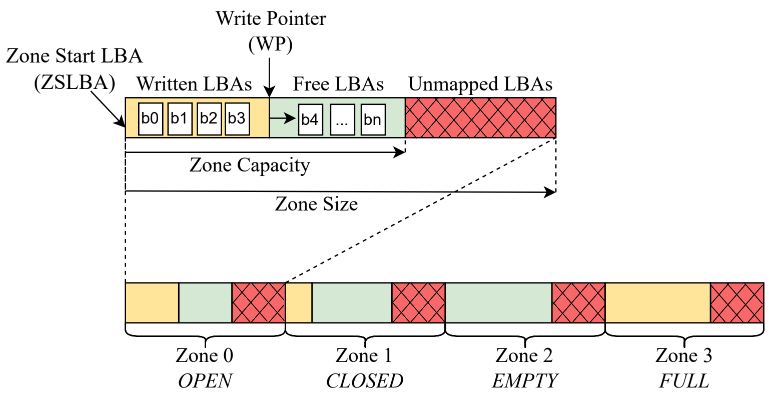


Figure 2.2: Layout of a ZNS SSD, depicting zone capacity, write pointer, and zone states associated to each zone. Adapted from [280].

management to the host system. Such interfaces allow for better coordination between the storage device and the host software by decreasing FTL responsibility, and increasing control for host software of the flash storage. Efforts for open flash SSD interfaces include *Software-Defined Flash (SDF)* [218] and *Open-Channel SSD (OCSSD)* [19, 235], which however failed to gain large scale adoption due to the lack of standardization, resulting in device-specific implementations, and complex interfaces, requiring software developers to have extensive knowledge of flash in order to be able to build flash-specific software.

The newest addition to opening flash SSD interfaces comes with arrival of *Zoned Namespace (ZNS)* SSD. The 2.0 base specification of NVMe [301] (published in June 2021), establishes the standardization of ZNS with the concept of splitting the address space of the storage device into a number of *zones*, which are independently addressable. This concept of representing the storage space with zones has previously been introduced with the addition of SMR HDDs [75, 84, 271]. These are a particular type of HDD that increase the storage density. Its concept of zones was included in the Linux kernel through the *Zoned Block Device ATA Command Set (ZAC)* and *Zoned Block Command (ZBC)* specifications [47, 48]. Similar to SMR, with ZNS zones have to be written sequentially, and must be reset prior to being overwritten, matching the internal characteristics of flash.

ZNS Interface. Figure 2.2 depicts a simplified layout of zones on a ZNS device, illustrating the management of the sequential write requirement within zones with a *Write Pointer (WP)* for each zone. The WP indicates the next *Logical Block Address (LBA)* to be written in the particular zone. The starting LBA of a zone is represented by a *Zone Start LBA (ZSLBA)*, identifying the first LBA of each zone. For zone management, each zone has an associated state, such as *FULL* and *EMPTY*.

In addition to the zone management, the NVMe standardization of ZNS introduces three new concepts. Firstly, the specification details the zone capacity, which limits the addressable region within a zone. In order to integrate into the Linux kernel, the zone size must be a power of two value, since kernel operations rely on bit shifts for addressing. However, the addressable space in a zone may not be a power of two value, and can therefore be less than or equal to the zone size. Any LBA beyond the zone capacity is not addressable.

Secondly, the specification adds a limit on the number of concurrently active zones. Zone states indicate the state of the zone, where zones that are currently in an *OPEN* or *CLOSED* state are considered to be active. As the device must allocate resources for active zones, such as write buffers, it enforces a limit on concurrently active resources. Lastly, ZNS introduces a new *zone-append* command, which instead of requiring the host to manage I/Os, such that they adhere to the sequential write constraint within a zone, allows the host to issue write I/Os to a zone without specifying the LBA. The device handles the write and returns the address at which the data is written.

The *zone append* command is particularly beneficial with large queue depths (submitting numerous asynchronous write I/O requests), which is not possible with write commands, as these must be issued at consecutive addresses and writes can be reordered in the block layer of the Linux kernel or inside the storage device. In order to adhere to the write constraint in a zone without relying on the zone append command, the `mq-deadline` scheduler within the Linux kernel must be enabled. The `mq-deadline` scheduler holds back I/Os and only submits a single I/O at a time to the ZNS device. Furthermore, this allows to merge I/O requests in the scheduler, enhancing performance by issuing a smaller number of larger I/O requests. Evaluations on the performance of the different schedulers show the benefits of merging I/O requests with larger I/Os of

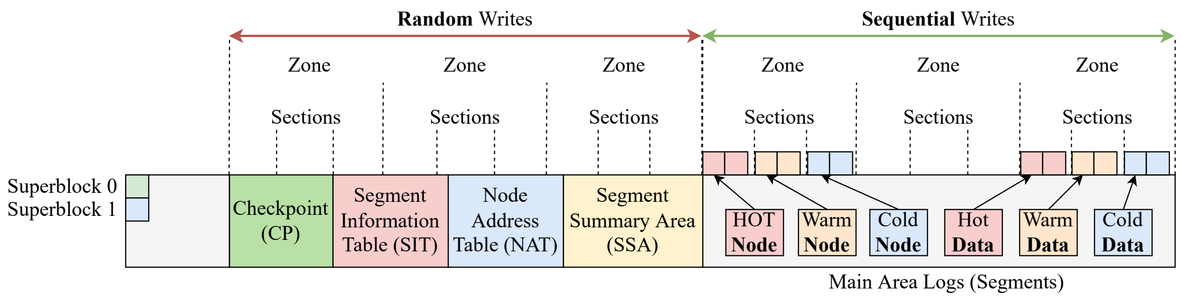


Figure 2.3: On-device layout of F2FS data. Adapted from F2FS [167].

≥ 16KiB being required to saturate the device bandwidth [262, 280].

2.6. F2FS: Flash-Friendly File System

A ubiquitous approach of managing persistent storage devices is with file systems, providing the familiar file and directory interface for organizing storage. The lack of in-place updates on flash pages, enforcing sequential writes, makes *Log-Structured File System (LFS)* [157, 251, 256] a suitable file system design. LFS revolves around writing data as a log, appending new data sequentially on the storage device. In this section, we describe the de facto standard LFS for flash-based storage devices, *Flash-Friendly File System (F2FS)* [167], as F2FS is used in this thesis. During the conducted literature study in Chapter 3, we evaluate designs of various other file systems for flash storage.

2.7. F2FS Data Layout

Internally, F2FS utilizes a data allocation unit referred to as a *block*, of 4KiB, in which blocks are allocated on the log. Consecutive blocks are collected into a 2MiB *segment*, of which one or multiple segments are further grouped into a *section*, that are combined into a *zone*. Figure 2.3 shows the layout of segments, sections, and zones for the data logs (on the right half of the figure). The left half of figure Figure 2.3 shows the metadata structures in F2FS to manage the file system, consisting of a *Checkpoint (CP)*, *Segment Information Table (SIT)*, *Node Address Table (NAT)*, and *Segment Summary Area (SSA)*. We now explain each of these data structures.

Checkpoint. F2FS utilizes *checkpointing*, in which all essential metadata for the file system is stored to provide recovery in the case of system failure. A checkpoint is periodically generated, or explicitly triggered, and persists all metadata information from memory. In the case of a system crash or power loss, the file system can recover the state from the latest checkpoint, referred to as *roll-back recovery*, since the latest changes which are not in the checkpoint are reverted. In order to recover the latest changes, the host must call *fsync()* to ensure that metadata and data are flushed from memory to the device. This recovery is referred *roll-forward recovery* since it recovers the state past the latest checkpoint. F2FS can only guarantee roll-forward recovery with *fsync()*.

Segment Information Table. With the data allocation in F2FS being organized in 2MiB segments on the log, the SIT maintains information on each of the segments. It maintains bitmaps for each segment to indicate valid and invalid blocks (blocks that have been overwritten).

Node Address Table. Similar to other file systems, a file in F2FS is managed through an *index node (inode)*, and contains all the file information, including the file specific metadata on creation time, access permissions, file name, and more. Figure 2.4 shows the inode of F2FS. For identifying the data blocks of the associated file, inodes contain a fixed number of *pointers* to the addresses of the file data. Since an inode is allocated in a block (4KiB), they can often contain inline data of the file, if the file data fits in the available space of the inode. If data for a file is updated, a new block is allocated on the log for the new data, followed by an update to the inode, in order to modify the pointer to the data to point to the newly written data. However, this allocates a new block for the inode of the file, requiring the metadata to track the inode locations to similarly be updated. These changes continue propagating to the parent nodes, resulting in a high increase of required metadata updates. Therefore, F2FS utilizes the NAT, to maintain an identifier of each node and its corresponding block address. Upon an update of a node, only the block address in the NAT is modified to depict the new block address of the node. Finding a node address then checks the NAT entry for the respective

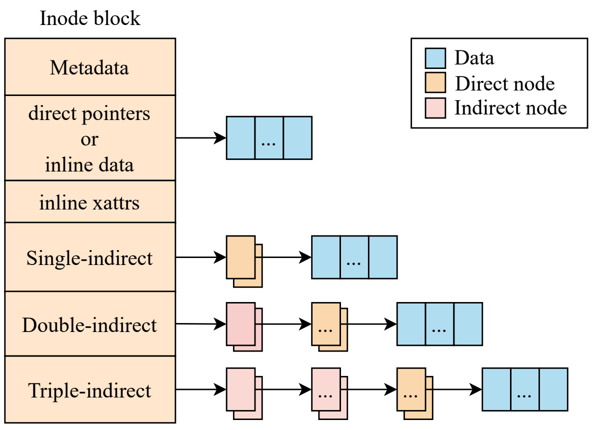


Figure 2.4: The inode structure of F2FS. Retrieved from F2FS [167].

node identifier.

Segment Summary Area. In addition to the segment information in the SIT, F2FS tracks information such as the owner of segments in the SSA. Furthermore, the SSA provides a cache for frequently accessed NAT and SIT information.

Main Area Logs. The right half of Figure 2.3 shows the layout of the main area logs, to which data and inodes are written. F2FS utilizes multiple concurrently writable logs to enhance data grouping and performance, with 3 logs to which nodes are written, and 3 logs for data. An essential mechanism for limiting the required GC, which F2FS must run to free space on the logs, comes from efficient *data grouping*. With data grouping, data that has a similar lifetime is grouped together. Given that data has a similar lifetime, it is likely to be updated within close proximity. Therefore, when GC is run, fewer valid blocks are present, as the data with the same lifetime has likely been updated, reducing the amount of valid data that must be moved by the GC process.

To support data grouping F2FS utilizes the three types of lifetime classes (hot/warm/cold), which are separated into the three different logs for node and data. The lifetime of data can be explicitly set for each file by an application through the passing of a *lifetime hint* with the `fcntl()` function. The Linux kernel provides a total of 5 different lifetime hints, which F2FS reduces to the three lifetime hints it utilizes. If a lifetime hint for a file is not set, F2FS either assigns the default warm lifetime classification, or assigns a lifetime classification based on the file type. With the extension of a file (e.g., `.txt`, `.pdf`), F2FS identifies which files are likely to be updated in the future. Multi-media files (e.g., `.mp4`, `.gif`, `.png`) are less likely to be updated and are directly classified as cold data.

2.8. F2FS Garbage Collection

As F2FS is log-structured, over time it contains valid and invalid blocks, similar to the FTL, and must therefore also run GC. In F2FS the process of GC is done at the unit of a section, where valid blocks in all the segments of the section are read and written to a free space, prior to all the segments in the section being freed. In F2FS GC is referred to as *cleaning*, and is run periodically (called *background cleaning*) or when free space for writing is needed (called *foreground cleaning*). The foreground cleaning utilizes a *greedy* approach for finding the section to garbage collect, which results in the largest amount of space being freed by erasing the section. Background cleaning on the other hand utilizes a *cost-benefit* method that considers the required data blocks to be moved during the GC of a section, and the resulting free space that is generated by the erasing of the section.

Given that during F2FS GC block addresses are modified, as the cleaning moves still valid blocks to free space, the metadata is not directly updated to depict these changes, in order to provide recovery. Therefore, F2FS is required to create a checkpoint after each GC call. Similarly, discard commands, issued after GC calls to delete data from the flash SSD, can only be issued after a checkpoint, such that in the case of a necessary recovery, the prior checkpoint still points to existing data that has not been discarded. Once a new checkpoint has been written a discard command can be issued.

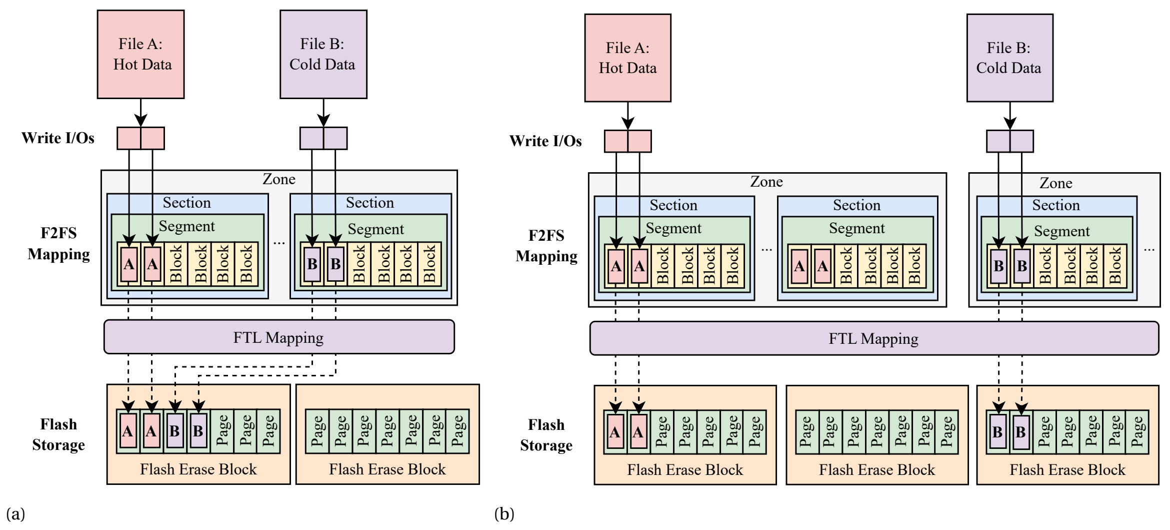


Figure 2.5: (a) *zone-blind allocation* suffering from inadequate physical data separation, (b) *zone-aware allocation* utilizing larger zones to ensure physical data separation.

2.9. Aligning F2FS Data Layout with the FTL

To ensure the data grouping of F2FS is similarly depicted by the mapping of data to flash pages in the FTL, F2FS utilizes the separation provided by sections and zones. The goal of sections is to align the F2FS allocation with the GC unit of the underlying FTL. Therefore, the F2FS GC occurring at the unit of a section, matches the FTL GC. Zones are utilized to avoid sections in different zones to be mapped into the same on-device erase unit by the FTL. With a mapping that results in different sections of different lifetimes (e.g., hot and cold data) being in written to the same erase unit on the flash SSD, as is illustrated in Figure 2.5a, data grouping is violated, furthermore resulting in possible GC overheads if only the hot data is updated while the cold data remains valid. This allocation of inadequately separating sections is referred to as *zone-blind allocation*.

With *zone-aware allocation*, illustrated in Figure 2.5b, the zone serves the purpose to provide a large enough separation between particular sections, such that the FTL similarly separates the written flash pages for the different file data into different erase units. As a result, only data of similar lifetime is within the same erase unit, resulting in reduction of GC overheads. As the erase unit of flash SSD is commonly hidden from users, the default F2FS configuration utilizes a single segment in each section, and a single section in a zone. However, if the flash storage device characteristics are known, these values can be configured.

2.10. Summary

The foundational building block of flash SSD is based on the *flash cell*. However, the characteristics of flash cells, and their utilization to construct mass storage flash SSD introduce flash management idiosyncrasies, such as having to erase flash prior to updating the data. Due to the resulting lack of in-place updates for flash storage, flash SSD employ firmware called the FTL, that hides the complexity of managing the flash, which however introduces garbage collection overheads. File system design, particularly F2FS, has therefore focused on utilizing flash-friendly data structures and mechanisms to integrate with flash storage. The hiding of flash management idiosyncrasies, specifically the process of GC, however results in unpredictable performance and high tail latency [55, 151, 310]. Therefore, interfaces that expose flash storage characteristics are appearing, with ZNS being the first standardized effort. The *zone* interface of ZNS eliminates the flash SSD GC, moving the responsibility of GC to the storage software on the host (e.g., the file system).

3
Literature Study - Integration of NAND Flash Storage in File System Design

With the increasing adoption of flash storage and the large utilization of file systems for storage systems, it is important to evaluate the data structures, algorithms, and mechanisms that arose during file system development for flash storage. Identifying the key approaches allows understanding the state-of-the-art approaches, and aims at evaluating if these approaches remain valid for newly arising SSD interfaces. In this section we identify the key challenges of flash storage, and evaluate how work has integrated solutions for these challenges into file system design. To this end, we conduct an extensive literature study on file system designs particular to flash storage. This section therefore makes the following contributions:

- We devise six key challenges for storage software arising from the integration of flash-based SSD, particularly focusing on leveraging its capabilities and enhancing device utilization.
- For each of the six devised flash integration challenges, we summarize the main methods of relevant work on dealing with and integration the particular challenge(s) into file system design.
- Based on the findings of this literature study, we present a discussion on the future applicability of the presented methods during this study, and evaluate the effects of newly arising flash-based SSD devices.

3.1. Research Questions

In order to evaluate the various work on flash storage implications for file system design, we devise three key *Survey Research Question (SRQ)* that aim at analyzing past, current, and future trends.

SRQ1. What are the main challenges arising from NAND flash characteristics and its integration into file system design?
Flash storage has particular characteristics, such as sequential writing, no in-place updates, and requiring erasing of flash blocks. This SRQ aims at analyzing what particular challenges arise for storage software from the flash-specific constraints and resulting effects of on-device operations. Devising a list of key challenges provides the foundation based on which relevant work in this literature study is selected, and the final report is structured.

SRQ2. How has NAND flash storage influenced the design and development of file system and the storage software stack?
Using the identified challenges in **SRQ1**, this SRQ evaluates for each of the challenges, how file system design has changed to integrate with it. As file systems are commonly built on top of existing storage software layers, such as the Linux Block I/O layer, we include methods and mechanisms in the storage software stack particularly devised for file systems and flash storage integration. As a result, this SRQ evaluates how the depicted challenges are addressed throughout the various software stack layers, up to the file system.

	Planning the review	
✓	Identification of the need for a review	(Chapter 3)
✗	Commissioning a review	-
✓	Specifying the research question(s)	(§3.1)
✓	Developing a review protocol	(§3.2.1)
✗	Evaluating the review protocol	-
	Conducting the review	
✓	Identification of research	(§3.2.2)
✓	Selection of primary studies	(§3.2.3)
✗	Study quality assessment	-
✓	Data extraction and monitoring	(§3.2.4)
✓	Data synthesis	(§3.2.4)
	Reporting the review	
✗	Specifying dissemination mechanism	-
✓	Formatting the main report	(Chapter 3)
✗	Evaluating the report	-

Table 3.1: Outline of the Systematic Literature Review approach presented by Kitchenham et al. [155], inspired by the structure of the literature study on Graph Analysis by Hegeman and Iosup [99].

SRQ3. How will NAND flash storage and newly introduced NAND flash-based storage devices and interfaces affect future file system design and development?
With a particular goal of this literature study being to evaluate the validity of data structures, algorithms, and mechanisms of flash, and understanding the applicability to ZNS, a newly arising storage technology, this research question furthermore aims at evaluating future challenges that may arise from new technology.

3.2. Literate Study Research Methodology

Several methodologies for conducting literature studies exist such as an unguided traversal of the literature [99] and using *snowballing* [295, 299]. However, we find that said methods lack systematic mechanisms in their evaluation, where the search space of relevant literature with unguided is not clearly defined and can become incomprehensibly large. Snowballing on the other hand allows limiting the search space by evaluation going forward, studies that reference the seed paper, and backwards, studies that the seed paper references, in the citations from a set of seed papers systematically. However, it can introduce possible bias from the seed paper selection. Therefore, we utilize a combination of approaches and additionally apply the *Systematic Literature Review (SLR)* presented by Kitchenham et al. [155]. The SLR approach relies on three separate stages to construct a systematic literature review, depicted in Table 3.1. As we do not inclusively apply every possible stage of the SLR method, the table additionally depicts which methods we apply in this study (indicated with a ✓), and where relevant information can be found in this thesis.

With these three stages, clear protocol and process definitions are established prior to conducting the review (discussed in Section 3.2.1), limiting possible reviewer bias and additionally enhancing reproducibility. The initial stage consists of planning the review, which includes establishing the need for this certain review and developing the review protocol. The protocol encompasses the inclusion and exclusion criteria, as well as research question definition, and the establishment of the review process. Based on an established protocol we conduct the review, establish the selection of studies to evaluate, and proceed to extract and analyze the studies. Lastly, with all collected data on selected studies, we format this thesis chapter to present the review in a comprehensible manner.

3.2.1. Review Protocol

By making use of the SLR process, we initially define a review protocol and review processes that we detail for this study. With emphasis on making this study reproducible, we provide detailed descriptions of each phase in the review process. A visual representation of the application of the review protocol phases is depicted in Figure 3.1. It revolves around three phases, firstly establishing the search space from which we extract relevant studies for this literature study. In the second phase we collect the studies and apply the defined selection criteria (explained in Section 3.2.3). Lastly, we analyze the collected relevant studies. The following

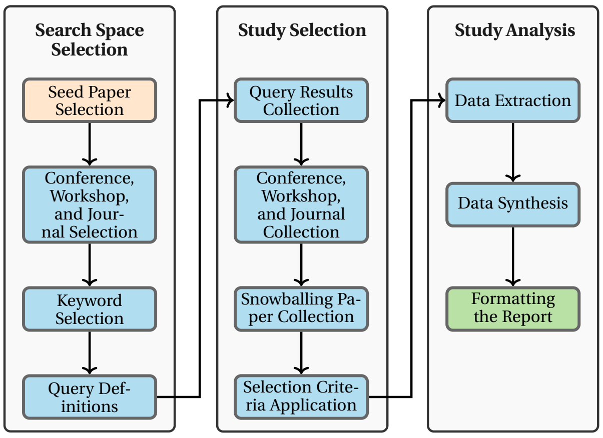


Figure 3.1: Review Protocol phases applied in this literature study.

sections explain each of the phases in detail, Section 3.2.2 explains methods used for establishing the search space of literature. Next, Section 3.2.3 depicts the selection criteria for extracting relevant studies from the defined search space. Lastly, Section 3.2.4 provides the methods of data extraction from studies and gives the organization of studies for this literature review.

3.2.2. Search Space Selection

The first stage of our review protocol defines the definition of the search space from which relevant studies are extracted. We make use of several approaches for identifying relevant studies. Firstly, we apply the snowballing method on a set of seed papers. Seed papers selected for this literature review are depicted in Table 3.2. With these seed papers, we analyze the studies from forward and backward citations of the seed paper. To further expand the search space of relevant studies, we examine the publications of numerous conferences, workshops, and journals which are focused on the area of systems and storage research. These conferences are analyzed in the range of 2010-2022, if present, as some are bi-annual or may not have been established in the given time range. The following venues are checked in this literature review:

- USENIX Annual Technical Conference (*USENIX ATC*)
- USENIX Conference on File and Storage Technologies (*FAST*)
- Networked Systems Design & Implementation (*NSDI*)
- European Conference on Computer Systems (*EuroSys*)
- USENIX Symposium on Operating Systems Design and Implementation (*OSDI*)
- Symposium on Operating Systems Principles (*SOSP*)
- ACM International Systems and Storage Conference (*SYSTOR*)
- ACM Workshop on Hot Topics in Storage and File Systems (*HotStorage*)
- Architectural Support for Programming Languages and Operating Systems (*ASPLOS*)
- ACM Special Interest Group on Management of Data (*SIGMOD*)
- International Conference on Very Large Data Bases (*VLDB*)
- IEEE International System-on-Chip Conference (*SOCC*)
- International Conference on Distributed Computing Systems (*ICDCS*)
- ACM/IFIP Middleware Conference (*Middleware*)
- ACM Transactions on Storage (*TOS*)

Title	Venue	Publication Year
F2FS [167]	FAST	2016
JFFS [300]	OLS	2001
LogFS [71]	Linux Kongress	2005

Table 3.2: Seed papers used for this literature review. Titles are shortened.

- International Conference for High Performance Computing, Networking, Storage, and Analysis (*SC*)
- International Conference on Massive Storage Systems and Technology (*MSST*)
- IEEE International Conference on Computer Design (*ICCD*)

Lastly, we run individual queries on academic search engines for scholarly literature; Google Scholar, Semantic Scholar, and dblp. We utilize two types of queries; *Relevant Studies Query (RSQ)* for finding of relevant studies for this literature study, and *Related Literature Studies Query (RLSQ)* for finding related work to this literature review. Related work encompasses surveys on file systems for flash, flash specific algorithms and data structures, and additional studies of flash related application and system integration. For each query, with each search engine, we analyze the 100 most relevant results (or less if there are fewer query results). The keyword queries for finding relevant studies and relevant related literature studies are, respectively:

RSQ1. Flash File System

RSQ2. NVM File System

RSQ3. SSD File System

RSQ4. File System SPDK [1]

RLSQ1. Flash File System Survey

RLSQ2. NVM File System Survey

RLSQ3. SSD File System Survey

Finding of related surveys is not limited to the established queries, but additionally during the snowballing and synthesis of conference, workshop, and journal publications we identify related work based on the prior defined classifications. Inclusion of relevant studies outside of the time range from 2010-2022 is only applicable when the study is retrieved using snowballing from seed papers, or the study is present in one of the respective query results. The timing constraint is thus only applied to the extraction of relevant studies by analyzing publications at conferences, workshops, and journals.

3.2.3. Study Selection Criteria

The second phase of the review protocol defines the study selection, which extracts the relevant studies from the defined search space with a set of established criteria. We define a specific Inclusion/Exclusion criteria, with which studies are selected for this literature review based on the appropriateness for the criteria. These criteria are based on the defined research questions and are aimed to narrow down the search space to a particular set of studies of interest in this review, and enforce only relevant work is included. While studies do not have to exclusively meet all the inclusion requirements to be included in this review, any if any of the exclusion criteria is present, the study is not included in this review.

I1. The work is novel.

I2. The work designs a file system specifically for NAND flash storage.

I3. The work adapts an existing file system to integrate with NAND flash storage.

E1. The work designs or adapts a file system not specifically for NAND flash storage.

[1] *Storage Performance Development Kit (SPDK)* [315] provides a number of tools and libraries for building high performant user-level storage software over NVMe, making it applicable to file system development on flash SSDs.

E2. The work designs or adapts a hybrid/tiered file system that utilizes various storage technologies, not focusing file system design to NAND flash storage.

E3. The work designs or adapts a file system which is evaluated on NAND flash storage, but not particularly built for NAND flash storage.

E4. The work designs or adapts a file system for SSD, but not specifically for NAND flash storage.

While meeting inclusion requirements does not mean a study is guaranteed to be included, it is more likely to be included. In the case of exclusion criteria, exceptions are made in cases of using papers to establish background knowledge or building context, however they are not the core focus of the respective section where its content is discussed.

Furthermore, there has been a plethora of flash-based file systems which utilize hybrid/tiered storage devices, including NOR-based flash [168], *Storage Class Memory (SCM)* [129, 184, 204, 232, 241], byte addressable Non-Volatile Memory (NVM) [166], and methods that expose byte addressable NVRAM on SSDs with custom firmware for metadata placement [124, 325]. Our focus is on block address storage, which is the most prevalent for storage. Additionally, combining of multiple storage technologies, such as SSD and HDD, commonly utilize SSD as a cache for the file system on the HDD, similarly shifting focus away from flash storage integration for the file system. For this reason we exclude such file system designs from the core study of this literature review. Similarly, file systems that are not designed specifically for flash storage, but have flash-friendly characteristics are also excluded. This mainly includes log-structured file systems that are intended for HDD, with a focus on writing sequentially to minimize arm movement on the disk, which coincidentally matches the sequential write requirement of flash.

3.2.4. Study Analysis

The last stage of the review protocol defines the extraction of data from the relevant studies, and establishing the final report. During evaluation of the different studies selected for this literature review, we disseminate the information presented based on their answering of the defined research questions. For this, we define the various key integration challenges of flash storage integration (Section 3.4), based on the hardware and software characteristics of flash storage. Using the defined integration challenges, we divide the contributions of the studies evaluated in this literature study into the respective challenge, and discuss its mechanisms for solving the particular flash integration challenge. These challenges are evaluated in Sections 3.5 to 3.11. Lastly, we discuss on the findings of the main findings from this literature study, followed by the relevant related literature studies for flash storage in Section 3.13.

3.2.5. Limitations

Albeit this literature study being extensively defined and established through clear protocol definitions, there are several limitations that remain. Firstly, the search space selection is only based on studies that have scientific literature. Therefore, file systems for flash which may be in the mainline Linux kernel, are not guaranteed to be included in this survey, if no scientific literature on it exists. Secondly, the search space is limited to only scientific literature written in English. This additionally limits the inclusion of conferences, workshops, and journals to venues with proceedings in English. Thirdly, given that snowballing uses a manual selection of meeting inclusion/exclusion criteria, possible bias is inherent. Lastly, as the resulting queries on the selected literature search engines produces several hundred thousand results, and we select to evaluate the first 100 results, we rely on the sorting of results based on relevance that each search engine provides. While we utilize multiple search engines in an effort to avoid bias from each search engine, it does not completely eliminate it.

3.3. Flash Storage Integrations

As there are various methods for integrating flash into storage devices, in addition building full SSD devices, ranging from directly attaching the flash chip to the motherboard, as is common with embedded mobile and IoT devices, or custom integrations of flash chips, we evaluate the file systems based on their level of integration. Given that a different integration exposes a different interface, the possibility to enhance particular operations is highly dependent on the integration.

Therefore, throughout the literature study of this thesis, we divide the relevant work based on the type of flash integration. Figure 3.2 shows three integration levels for flash, where Figure 3.2a depicts the conventional integration with a SSD. Figure 3.2b shows a custom integration of flash storage for devices such as

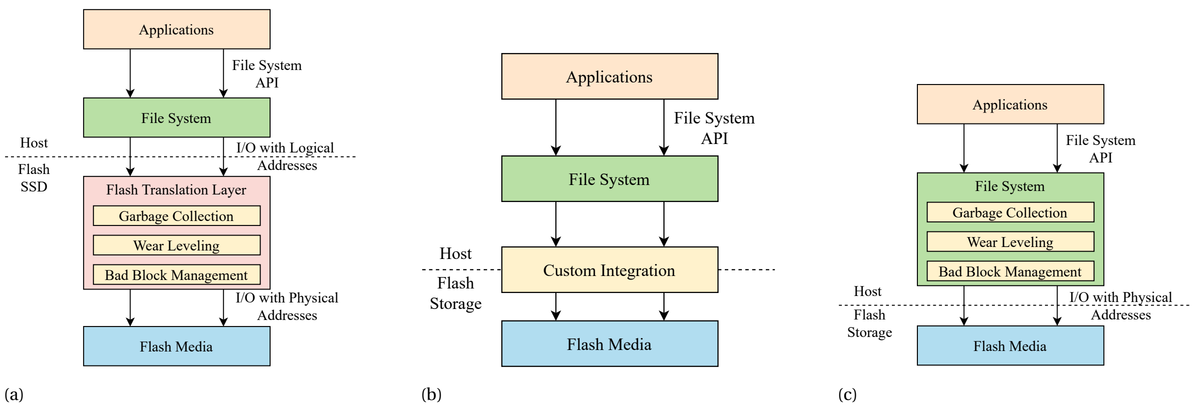


Figure 3.2: Integration of flash storage into host systems with (a) showing a conventional SSD with a FTL on the storage device, (b) a custom flash integration, through interfaces such as OCSSD, custom FTL, custom device drivers, and multi-stream SSD and (c) flash storage on embedded systems with direct management for flash from the file system.

Open-Channel SSD (OCSSD) [19, 197], multi-stream SSD [16, 134], and *Software-Defined Flash (SDF)* [218]. The main benefit of these types of integration is that the flash characteristics are no longer hidden behind the device, giving the host an increasing level of storage control. OCSSD is a type of SSD that exposes the device geometry to the host, allowing the host to manage device parallelism and allocation. While such a device allows increased data management for the host software, it comes at increased complexity for managing the device constraints.

Lastly, Figure 3.2c shows flash integration at the embedded level, such as is commonly used in mobile devices and IoT devices. In embedded flash configuration the flash chip is commonly directly attached to the motherboard, giving the host system full control over the underlying flash storage. Throughout the literature study chapter of this thesis, we group file system design and mechanisms based on these three integration levels, as different levels of integration allow different degrees of flash management and ranging possibility for flash integration.

3.4. Challenges of Flash Storage Integration

While SSD uses the same block interface that is used with HDD, flash has different characteristics that software must account for to better integrate flash storage. This section details the challenges that arise from integrating flash storage into systems, providing the guidelines along which we dictate the bottom up view of changes in the host storage software stack, up to the file system. We define the challenges to account for the characteristics of flash storage, as well as enhance its integration into host systems. In the case of SSD devices, these challenges often largely depend on the underlying FTL, as it is making the final decision, independent of what data placement the host implements, however aiding the FTL can increase the performance. Embedded devices provide a higher level of host data placement by eliminating the FTL and directly attaching flash chips to the motherboard. Each challenge is assigned a specific identifier with the *Flash Integration Challenge (FIC)*, in order to refer back to the specific challenge throughout this literature review. Table 3.3 summarizes the 6 key challenges arising from flash storage devices.

FIC1: Asymmetric Read and Write Performance. On flash storage write operations require more time than read operations [35, 94, 109, 188, 269], making it important for software to limit write operations. Particularly, frequent small writes that are smaller than the allocation unit, referred to as *microwrites* incur significant performance penalties, and should be avoided where possible. Similarly, methods for enhancing the write performance are important to account for the lower write performance, compared to read performance.

FIC2: Garbage Collection (GC). While the FTL hides the flash access constraints from host applications, providing seemingly in-place data updates, it adds the cost of performing garbage collection to free up space. GC overheads have unpredictable performance penalties for the host system [151, 310], resulting in large tail latency [55]. Dealing with, and aiming to minimize required garbage collection for the flash device is a key challenge in integrating flash storage.

FIC3: I/O Amplification. Due to the characteristics of flash avoiding in-place updates of flash pages, writes often encounter *Write Amplification (WA)*. With this the amount of data that is written on the flash storage is larger than the write that is issued by the host system. For example a 4KiB issued write may increase to 16KiB

ID	Flash Integration Challenge	Description
FIC 1	Asymmetric Read and Write Performance	Write operations require more time than read operations [35, 94, 109, 188, 269]
FIC 2	Garbage Collection	The lack of in-place updates results in flash storage running garbage collection to free space and clear invalid pages.
FIC 3	I/O Amplification	The lack of in-place and the required garbage collection introduce write amplification, writing more flash pages than the size of the I/O issued by the host.
FIC 4	Flash Parallelism	The architecture of flash utilizes a high degree of parallelism (channels/chips/planes) to be utilized to enhance performance.
FIC 5	Wear Leveling	Limited lifetime of flash cells requires careful consideration during writes to ensure flash is worn out evenly across the storage space.
FIC 6	I/O Management	Optimizations on the I/O requests, such as merging, aims at leveraging the flash storage capabilities and reducing I/O latency.

Table 3.3: Overview of the challenges arising from integrating flash storage. The identifier corresponds to the respective *Flash Integration Challenge (FIC)* referred to throughout this literature study.

being written on the device, due to possible garbage collection requiring to copy data, resulting in a WA factor of 4x. WA furthermore adds to an increase in wear on the flash cells [194]. *Read Amplification (RA)* similarly is caused by requiring to read a larger amount of data than is issued in the read I/O request. RA most commonly happens when reading metadata in order to locate data, thus requiring an additional read of metadata on top of the request read I/O request for the data. This is most often inevitable, as all data requires metadata for management, however this should be kept to a minimum at application-level. Furthermore, minimization of WA is more important than RA, since write requests have a higher latency than read requests, and writing has a more significant impact on the flash storage, resulting in increased flash wear. While read requests also incur wear on the flash cell, called read disturbance [189], it is not as significant as for write requests.

FIC4: Flash Parallelism. With the various possible levels of parallelism on flash storage devices (discussed in Section 2.3), exploiting of the various possibilities requires software design consideration to aligning with these. Although the I/O scheduling of on-device parallelism, such for channel-level parallelism, is responsibility of the FTL (on devices at SSD integration level), the FTL implements particular parallelism, given that the host I/O requests aligning with the possibility of parallelizing the request, such as with large enough I/Os to stripe across channels and dies. Embedded device and custom flash integrations have more possibility to manage flash device parallelism at the host software level.

FIC5: Wear Leveling (WL). Given limited program/erase cycles for flash cells, even wear over the entire device is required to ensures that no specific areas of the device are burnt out faster than others. Similar to flash parallelism, this largely depends on the flash integration level, as the FTL at the SSD integration level ensures WL, however embedded flash integration and custom flash integration is required to place more significance on ensuring even wear across the flash cells. Strongly related to prior flash integration challenges, wear is commonly a result of GC, which in turn increases the I/O amplification, and particularly the WA and RA [57, 103].

FIC6: I/O Management. As SSD ships with integrated firmware to expose the flash storage as a block addressable storage device, integration into the current software stack is seamless. Figure 3.3 shows the integration of a flash SSD into the Linux kernel storage stack. Since flash storage devices are significantly faster than prior storage technologies, such as HDD, the storage software stack becomes the dominating factor in I/O latency [27, 28]. One particular optimization for performance of I/O requests to flash storage devices is provided by an I/O scheduler, deciding when to issue I/O requests to the storage device. As is visible in Figure 3.3, the block I/O layer implements various schedulers with different functionality. Providing a ranging degree of optimizations for I/O requests, such as varying scheduling policies and merging of I/O requests, or possible reordering, specific configurations are favorable to increase performance with flash storage. Particularly the utilization of multiple queues, with multiple software and hardware dispatch queues (visible in the `blk-mq` configuration of the block I/O layer), allows better exploitation of flash storage capabilities, and avoids certain Linux kernel overheads. Furthermore, evaluating mechanisms that reduce the latency of I/O operations,

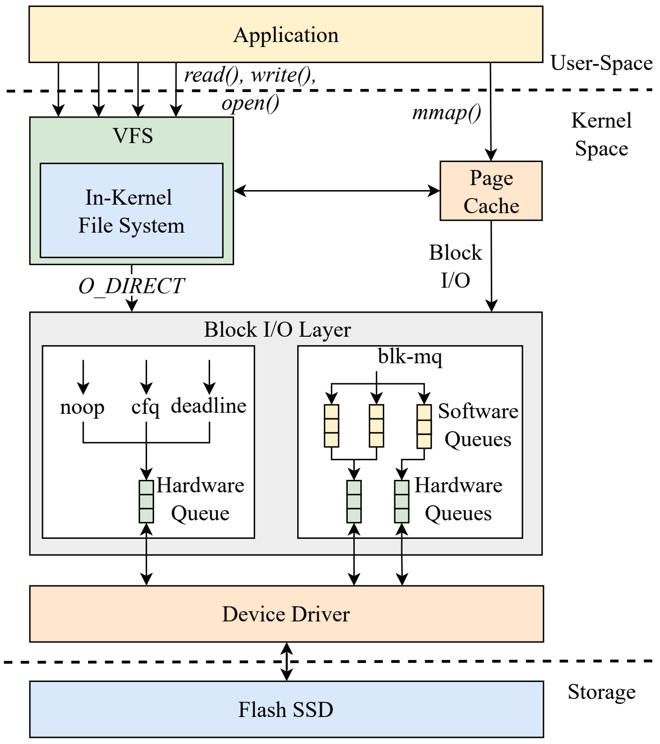


Figure 3.3: Visual overview of the storage stack in the Linux kernel. Adapted from [76, 208].

Integration Level	File Systems
SSD Flash Integration	[1, 90, 105, 118, 123, 135, 138, 167, 185, 190, 195, 216, 249, 283, 318]
Custom Flash Integration	[67, 126, 141, 174, 175, 197, 240, 246, 302, 317, 320]
Embedded Flash Integration	[4, 71, 107, 108, 120, 128, 147, 150, 173, 176, 186, 187, 201, 202, 213, 222, 223, 226–228, 230, 254, 282, 300, 312, 322, 323, 329]

Table 3.4: Classification on the flash integration level utilised for the file systems evaluated in this literature study.

and particularly write I/O operations.

3.4.1. Flash Integration Organization

With the different possible levels for integration of flash storage (recall Figure 3.2), and while mechanisms for solving flash integration challenges are frequently applicable at various integration levels, several of the mechanisms we present require deeper integration of flash storage, levering increased control, in order to be implemented. For instance, the incorporation of the various levels of on-device parallelism is not directly possible at the SSD integration level, as the FTL hides the parallelism on the physical device from the host system. The custom and embedded level flash integration provide the host with more possibility to manage these. In order to separate the possibility of mechanisms to be implemented with a particular flash integration level, Table 3.4 provides a classification of each evaluated file system in this literature study to the respective integration level. During the evaluation we present the different mechanisms to solve a FIC, and indicate which file systems utilize these. Therefore, when considering the feasibility of a mechanism for a particular flash integration consult this table to see its applicability. Note that file systems are not limited to the classification we provide, as for instance file systems designed for SSD flash integration also work on some embedded flash integration. However, we utilize only a single classification for each file system to avoid confusion. Exceptions are made only in specific cases where an existing file system is adapted for a different flash integration level.

We divide the discussion of mechanism by the FIC for which the evaluated study presents a novel solution. This implies that mechanisms that solve multiple FIC are discussed in detail in the first section they appear

Mechanism	File Systems
Write Optimized Data Structures (§3.5.1)	[67, 107, 123, 249, 283]
Write Buffering (§3.5.2)	[125–127, 138, 197, 227–229]
Deduplication (§3.5.3.1)	[67, 105, 186]
Compression (§3.5.3.2)	[67, 108, 120, 214, 300]
Delta-Encoding (§3.5.3.3)	[67, 105]
Virtualization (§3.5.3.4)	[176, 329]
Flash Dual Mode Switching (§3.5.4)	[173, 302]

Table 3.5: Mechanisms for file systems to deal with **FIC1**, asymmetric read and write performance of flash storage, and the respective file systems that implement a particular mechanism.

in, however are also mentioned in all latter sections for the FIC that the mechanism solves. Therefore, each FIC section contains a table of the respective mechanisms presented to solve that particular FIC, along with a reference to the corresponding section of its detailed discussion.

3.5. FIC-1: Asymmetric Read and Write Performance

Given that read and write performance on flash storage is asymmetric [35, 109, 269], this section provides the various mechanisms to handle the asymmetric performance. Table 3.5 shows the various methods discussed in this section, for dealing with asymmetric performance, and how to increase file system performance. Such methods and mechanisms include data structures particularly optimized for characteristics of flash storage, efficient methods of data caching, and effective organization of data on the storage device.

3.5.1. Write Optimized Data Structures

The characteristic of flash having lower write than read performance requires that data structures for flash are write optimized. Such data structures which are optimized for write operations are referred to as *Write Optimized Data Structure (WODS)* [12]. With the addition of the missing support for in-place updates on flash, the best suiting data structure for flash-based file systems is a log-based structure. As a result, all file systems discussed in this section are LFS. The nature of a LFS being append-only writes accounts for the lower write performance on flash, which matches the write updates to the operations more optimal for flash, which are smaller fine-fine grained updates [193] in a log structured fashion [167].

However, while the log provides increased write performance, metadata is scattered throughout the log, requiring a full scan of the log to locate metadata. Therefore, file systems commonly employ tree-based data structures for metadata, decreasing the worst case time complexity from $\mathcal{O}(n)$ to $\mathcal{O}(\log n)$. B-tree is a commonly used data structure for storage systems, including databases [46, 87] and file systems [249]. Nodes in a B-tree can be larger than in conventional binary search trees, allowing the node to align to a unit of the underlying storage. B-trees furthermore are maximizing the breadth of the tree, instead of its height, in order to minimize the I/O requests in order to locate data, since each traversal to a child node requires an I/O request. The tree itself is sorted and self-balancing, making the worst case complexity for search relative to the tree height, however also requiring to balance the tree. The B+tree further reduce required I/O by only having data in the leaf nodes, such that higher nodes only contain keys, increasing the number of keys that fit inside a block. Leaf nodes are linked in a linked list, for faster node traversal, which in turn speeds up searching.

In order to write optimize these trees, B$^{\varepsilon}$-trees [12, 24] adapt the node structure of the B-tree to include a buffer to which updates to its children nodes are written. As node updates are initially written in memory, this allows to gather a larger number of small writes, encode these in the added buffer of the respective nodes, and write these as a larger unit, avoiding frequent small updates to nodes. The most recent version of BetrFS [123] (published in 2022) implements such a B$^{\varepsilon}$-tree as the metadata storage for indexing data. It is implemented in the Linux kernel as a key-value store, based on TokuDB [234], which exposes a key-value addressable interface. The benefit of this B$^{\varepsilon}$-tree is that the nodes have a larger (2-4MiB) sequentially written log, batching updates into larger units and thus avoiding small updates. Initially all updates go into the root node message log, which when full gets flushed to its child nodes. While WODS provide optimized write performance by batching updates, read requests require reading an entire node (2-4MiB), causing small read requests to suffer from significant read amplification.

SSDFS [67] adapts the tree design to utilize hybrid nodes, since the node allocation uses blocks, which are several KB in size, and may not directly be filled directly. Therefore, hybrid nodes in the tree adapt the size of the node, such that if a hybrid node is allocated and filled, it allocates an additional hybrid node, which upon being filled is merged with the first hybrid node and becomes a leaf node. This allows to reduce write amplification (solving **FIC3**) if a node is not filled enough. An additional optimization to B-trees is the *Parallel I/O B-Tree (PIO B-Tree)* [250], which implements a parallel flash-optimized B-Tree variant (solving **FIC4**), utilizing a larger I/O granularity to exploit package-level parallelism, maintain a high number of outstanding I/O requests to utilize the channel-level parallelism, and avoid mixed read and write operations.

3.5.2. Write Buffering

In addition to WODS optimizing write operations file systems need to also avoid small writes, referred to as *microwrites*. Particularly, as file systems write in units of blocks, which commonly are 4KiB, small writes require a full unit to be filled. The majority of file systems provide the possibility for inline data in inodes, such that the inode data, which is written regardless, has a small capacity to include data. However, this still requires that for small files inodes are directly written to flash. Therefore, several schemes that involve buffering and caching of data in memory, before flushing to the flash storage, provide increased write performance, and additionally increase overall performance.

While buffering of I/O prior to flushing to flash storage provides performance gains, buffers are often limited in size and are much smaller than the persistent storage. Furthermore, in addition to buffering write requests for new data, accessing or updating existing data is also cached in the buffer. Therefore, the caches utilize effective methods that minimize the cache misses by optimizing the eviction policy. *Least Recently Used (LRU)* [276] is a common caching policy that maintains the items in the cache in the order of usage, where the least recently used item is selected for eviction. This mechanism is extended by DFS [126, 127], which utilizes *lazy LRU*, which does not insert accessed data into the buffer upon a cache miss, but rather inserts the data into the buffer only on an additional cache. Requiring of two cache misses implies that the cache only contains data that is frequently accessed, instead of caching all data from a cache miss.

Clean First Least Recently Used (CFLRU) [229] is another extension on the LRU algorithm, which splits the buffer into two segments, one as the working region in which recently accessed pages reside, which are managed in *Most Recently Used (MRU)* fashion, therefore depicting the frequently accessed pages. The second region, called the clean-first region contains the less commonly accessed pages in LRU fashion. On eviction (e.g., when writing a new page and freeing space in the buffer), it first attempts to evict a clean page, rather than a dirty page from the clean-first region, as this does not require a flush of the dirty data to the flash storage, and only resorts to evicting dirty pages as last resort. *Flash Aware Buffer (FAB)* [125] optimizes the caching policy to align with the flash characteristics by organizing pages in the cache in a larger unit, called *block*, and upon eviction flushes entire blocks to the flash storage, issuing larger I/Os and aligning better to the flash erase unit.

Similarly, NAFS [227] implements a *double list cache*, containing a clean list with only clean pages for caching of data, and a second dirty list for writing of modified pages and to prefetch of pages based on the access patters, only containing dirty pages. The benefit of two separate lists is that firstly it allows caching write operations and avoid small writes to the flash device, and secondly it prefetches data into the clean list in order to minimize cache misses. EnFFiS [228] presents the *dirty-last cache* policy, which considers the flash characteristics by utilizing a *delay region* and a *replacement region*, which correspond to a multiple of the flash blocks. By using a multiple of the flash blocks, the data written is sequentialized and written to the flash as a larger unit, where dirty pages are initially moved from the replacement region to the delay region, before being flushed to the flash. This buffering in the delay region allows collecting more dirty pages, improving performance and avoiding smaller writes to flash.

StageFS [197] likewise utilizes two stages, where write operations are initially written into the first stage, called the *file system staging area*. The issued writes to the staging area are completed in a *persistence-efficient* way, which utilizes a log to account for asymmetric flash performance for the staging area. Subsequently, writes are then regrouped, based on the file system structure and hot/cold identification, and are written to the second stage, based on the group assignment. The staging area allows writing synchronous I/O directly to the staging log with optimal flash write characteristics, lowering completion latency, followed by better grouping of data when writing from the staging to the second stage file system area.

In an effort to limit the required memory for data caching, DevFS [138] proposes the *reverse caching* mechanism for effectively managing host memory and device memory. Reverse caching aims at keeping only active files in the device memory, which is limited in size, and upon closing of a file migrates the metadata to the host memory. Reopening a file migrates it back to the device memory, and consistency of metadata is not violated as any actively modified metadata is in the device memory, and only inactive metadata is in the host memory. To efficiently utilize reverse caching, DevFS uses a host-memory cache that is able to use *Direct Memory Access (DMA)* to move metadata between itself and the device memory, additionally minimizing host overheads.

3.5.3. Reducing Write Traffic

In order to avoid the slower write performance of flash storage another mechanisms aims at minimizing the amount of data that is written to the storage device. Whenever data is updated on the flash storage, updating all metadata and writing the new data incurs a significant amount of write traffic, in addition to causing WA. Therefore, reducing the amount of data being written through mechanisms such as deduplication, compression, and delta-encoding helps at avoiding the increased WA.

Deduplication

File systems commonly contain duplicate data, from backups or archival copies made and general work. Therefore, file systems often aim to avoid creating duplicates of existing data, which is referred to as *deduplication*. Evaluations show that in high-performance computing centers on average about 20-30%, with peaks of 70%, of the stored data can be removed through deduplication [205]. Deduplication avoids writing the same data multiple times, which in turn helps reduce the write traffic and minimizes write and space amplification (solving **FIC2**) and prolong the device lifetime. Effective deduplication relies on hash functions, which provide a deterministic output, called the *digest* or commonly referred to as the *fingerprint* of the data, based on which duplicates can be identified. The utilized hash functions for deduplication are *one way hash functions* [206], which calculate a fingerprint of the data, or digest, but given the digest or fingerprint, the data cannot be generated. Given data from a file for instance, hashing the data provides a digest, and if the same data is to be written again, the same digest is generated. By identifying if a digest already exists in the file system, can it identify if the data is being duplicated, and avoid writing the duplicate by making the metadata of the newly written data point to the existing data.

In order to apply deduplication on file system data, two methods exits. **(1)** Deduplication with fixed-sized chunks, where hashes are generated based on the entire file or a pre-determined chunk size, and **(2)** variable-sized chunks where the chunk size that is hashed depends on the file and content. The effectiveness of deduplication with fixed-sized chunks is limited and highly dependent on the chunk size and modification sequences in the chunks [186], as for example a small change in a large file, where the chunk size is the entire file, results in a completely different hash, even if much of the data is duplicated. While variable-sized chunks reduces the configuration dependability it is significantly more complex to implement. DeFFS [186] implements a duplicate elimination algorithm that reduces the complexity of identifying duplicates. As comparing duplicates of every byte is not possible, the algorithm finds the smallest modified region in a file adapting the chunk size, which is initially assigned the default value that corresponds to the flash page size.

The idea of eliminating duplication through fingerprints is not new for SSD, as existing FTL implementations implement such mechanism. *Content Aware FTL (CAFTL)* [37] generates collision-free fingerprints of write requests and maintains fingerprints of all flash resident data in order to avoid writing the same data twice. Flash Saver [179] is a similar architecture running between the SSD and the file system, which manages the file system I/O requests and ensures deduplication using SHA-1 fingerprints [294]. A plethora of similar hash-based deduplication mechanisms exist [79, 91, 154, 304], which may not be particular to file systems, but can be adopted by a wide range of storage systems. File systems with deduplication are CSA-FS [105], which implements deduplication by calculating the MD5 digest [248] (MD5 is a hash function applied to the input) and looks up the result in a hash table which provides the corresponding LBA of the block. If it already exists and the user requested a new file with it, the new inode simply uses the given LBA from the hash table, instead of duplicating the data again. Similarly, SSDFS [67] maintains fingerprints with its metadata B-Tree, and DeFFS [186] stores hash keys with all inodes for their data to avoid duplicates.

Compression

An effective method for reducing the required storage space for data is to utilize *compression*. While there exist lossy compression algorithms [113, 140], which discard parts of the data, and lossless compression algorithms [140, 260], which maintain all data, lossy compression is very application dependent, and in the case of file systems we assume a lossless data storage and therefore focus on lossless compression algorithms. Lossless compression methods rely on three methods. **(1)** *Run Length Encoding (RLE)* which minimizes size by taking the repeating characters, referred to as the run, and replacing them with a 2 byte sequence of the

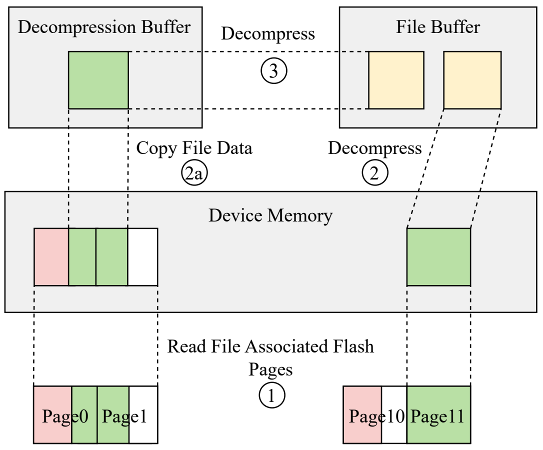


Figure 3.4: The benefit of *Page Boundary Alignment (PBA)* during decompression arises from the elimination of copying associated file data.

number of repetitions of the character, called the run count, followed by the replaced character. For example the sequence of “aaabbccc“ is represented as “a3b2c3“ with RLE. **(2)** *Lempel-Ziv (LZ)* [327, 328] which utilizes a dictionary to replace strings with their dictionary value. Variations of the LZ algorithm exist, such as LZ77 [327], LZ78 [328], and LZW [297]. The dictionary is constructed at the compression time and used a decompression time [139]. A well-known compression algorithm GZIP [58] is based on two LZ algorithms. **(3)** Huffman Coding [106] which creates a full binary tree based the frequency of occurring characters and generating code of the character and the frequency.

JFFS [300] and several other file systems [49, 67, 86, 108, 120, 214, 300] provide a data compression, however metadata compression has shown to cause decompression overheads [136]. Applying compression at the file system level allows reducing the utilized storage space, especially on flash this reduces the space and write amplification factors (solving **FIC3**), and in turn prolonging the flash lifetime as shown by Li et al. [181] on the benefits of different compression algorithms for NAND flash lifetime. SSDFS [67] utilizes LZO compression [215], a library for data compression with LZ algorithms, for data and metadata. Given that different data benefits from different compression mechanisms, where for instance data is less frequently read can be compressed more efficiently than frequently read data, which may require more simplistic decompression to minimize overheads, or embed metadata in the compressed data to avoid decrypting data and metadata separately. Adaptive compression selection is employed in an extension of F2FS [120], where *File Access Pattern-Guided Compression (FPC)* selects the compression method for files based on how compressible they are.

An issue of compression is that it must be aligned to the flash unit, in order to avoid unnecessary read amplification and required data copying. Figure 3.4 shows that if the compressed data is not aligned to the flash unit of a page, such that compressed data can extend across pages, the decompression process becomes more complex. This is due to first having to read all the associated flash pages into the device memory. However, if compressed data can extend across flash pages, this implies that non-associated data can be included. Therefore, only the associated data must be copied to a *decompression buffer* (step 2a), from which the data can be decompressed (step 3). LeCramFS [108] (short for less crammed FS) implements a read-only file system with compression for embedded devices, that avoids this additional copy of data by using PBA. Any compressed data will not extend across page boundaries, implying that no additional pages with non-associated data are read, and the data can be decompressed directly (step 2). Therefore, it avoids the additional copy of data and reduces the read amplification (solving **FIC**3). In order to avoid wasting space if the page is not fully utilized from a single compression, LeCramFS extends the compression implementation with a *partial compression*, which splits the data into parts that fit into the available space. These parts are then compressed separately, allowing to utilize all available space.

Delta-Encoding

Similar to deduplication, delta-encoding lowers the write traffic, further limiting the space and write amplification (solving **FIC2**) by instead of writing out entire data when updated, with delta-encoding the new data is compared to the old data and only the differences, called the *delta*, are written. This is especially benefi-

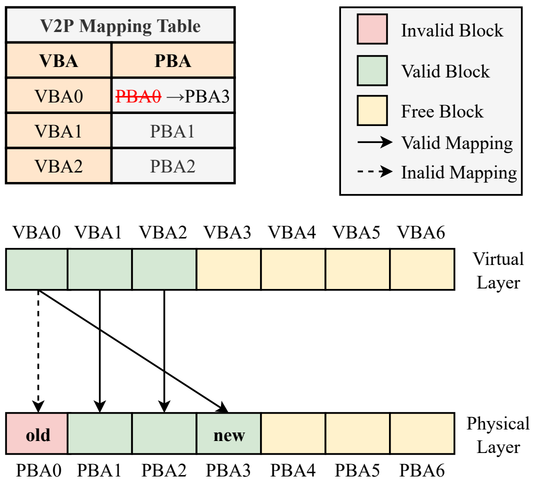


Figure 3.5: A simplified example scenario of virtualization on top of the physical storage.

cial in the case of small changes in large files, where it avoids writing the entire file again and writes only the changes in the data. While delta-encoding provides a similar space reduction to compression, particular file characteristics of what is being encoded can affect resulting performance. For instance, data sets that have a significant redundancy across them, such as email data sets, require significantly less space than being compressed [66]. However, non-textual data, such as pdf documents, will have large deltas for even small changes, where compression is likely to be a better choice. Therefore, the application of delta-encoding depends on the data characteristics and what type of content is being encoded, in order to achieve better utilization of delta-encoding compared to simpler compression. CSA-FS [105] implements delta encoding on its file system metadata (superblock, descriptor, bitmaps, and tables), allowing minor updates such as the access time of metadata and bitmaps to only write the new changes. Similarly, SSDFS [67] uses delta-encoding for user data.

Virtualization
As Butler Lampson mentioned in his famous quote from 1972 (which was originally stated by David Wheeler), "Any problem in computer science can be solved with another level of indirection" [162]. By adding a layer on flash storage the write traffic can be reduced as a result of avoiding metadata update propagation (as with the wandering tree problem [17]). Adding a virtual layer is conceptually identical to *Logical Block Address (LBA)* on top of PBA. Figure 3.5 shows an example of how virtualization maintains the same *Virtual Block Address (VBA)* when file data is updated, eliminating a need for metadata updates. Note, for simplicity we assume the virtual layer to be on top of the physical layer, providing PBAs, however it can be directly on the physical layer of the flash storage, depending on the flash integration level, and would therefore be using *Logical-to-Physical (L2P)* mappings. In the illustrated scenario, three blocks are mapped to PBA0, PBA1, and PBA2, respectively. Assuming these are part of a file, metadata points to these respective virtual block addresses. If the first block of the file is modified, a new data block is written at PBA3. With virtualization, the VBA remains unchanged, only the V2P mapping table is modified to depict the new mapping of PBA3. As a result, file metadata remains unchanged, avoiding write amplification for updating file metadata.

A similar mechanism is implemented in NANDFS [329], using a *sequencing layer* that implements the block allocation as immutable storage on the logical layer. Therefore, when file system data is updated, it cannot overwrite and simply marks the data as obsolete, and the new updated data is written in a newly allocated block. The sequencing layer implements the L2P mapping, such that the LBA of the new data is the same as the old LBA, avoiding the updating of data addresses in the file system metadata. This implies that the file system does less writing by eliminating metadata updates, which in turn also reduces the required GC.

The concept of virtualization can similarly be applied through *Address Remapping (AR)*. With AR only the L2P mapping table of the storage device is modified, which similar to virtualization avoids rewriting of metadata. AR is an effective mechanism for solving additional overheads in file systems, such as file system garbage collection [133], journaling [43, 133, 296], and data duplication [325]. Lee et al. [176] propose *Remap-Based In-Place-Updates (RM-IPU)*, an address remapping scheme implemented in F2FS to solve issues of out-of-place updates. Instead of applying AR into a single file system operation, RM-IPU includes all write operations to utilize AR. All updated data is first stored in the log, followed by AR to update data pointers, thus avoiding metadata updates for file overwrite operations. File write operations to new files are appended to the log as usual, and the contiguity in LBAs for files is maintained.

3.5.4. Flash Dual Mode Switching

Some modern flash devices allow the host to switch the cell level of underlying flash blocks (e.g., switching MLC to SLC) [183, 302]. This provides the benefit that a lower cell level, representing fewer bits, has a lower read, write, and erase latency, thus providing the flash with a higher performance than with a larger cell level configuration [302]. This switching between flash modes is referred to as *flash dual mode*, which however comes at the cost of being able to store less data in the same amount of flash, as the block is only able to store half the data in the example of switching from MLC to SLC. DualFS [302] utilizes the flash dual mode feature to provide a dynamically sized SLC area, alongside the remaining MLC area, to accelerate the performance of critical I/O requests. By evaluating the I/O queue depth of I/O requests, DualFS determines the criticality of the incoming request, and maps it to the SLC area for increased performance. It further profiles incoming request based on the hotness and allocates hot data into the SLC mode for lower request latency.

FlexFS [173] implements a similar mechanism for increased performance on the SLC area. The drawback of a design that utilizes a SLC area for incoming write I/O, is that data is required to be moved from the SLC area to the MLC area as it has a lower write lifetime. With the SLC are being used for critical I/O requests that require lower completion latency, it introduces *data migration overheads*. To solve this, FlexFS implements several migration techniques that aim to hide the overheads for the data migration from the host. The first technique revolves around *background migration*, which pushes the migration to happen when the system is idle. The second technique is *dynamic allocation*, which writes non-critical requests to the MLC area, saving on flash degradation in the SLC area, as well as avoiding future data migration. The dynamic allocator functions based on measurements of prior system idle times from I/O requests, to predict current idle time, and if sufficient idle time is predicted in order to complete data migration, the data is written to the SLC, and otherwise part of the data, depending on required migration and idle time, is written to the MLC area. The last technique, *locality-aware data management*, takes into account the hotness of data, and the dynamic allocator attempts to migrate only cold data from the SLC area.

3.5.5. Summary

With the write performance of flash storage being lower than its read performance, particular attention is paid to designing effective mechanisms and methods for achieving faster write performance. Effective caching methods, buffering data before writing, allow to reduce write latency. Similarly, mechanisms that reduce the write traffic to the device, such as deduplication, delta-encoding, and virtualization allow are effective for dealing with the lower write performance of flash storage. Lastly, the possibility of flash-dual mode switching allows to change the flash cell level to provide lower latency write request completion for critical write request.

3.6. FIC-2: Garbage Collection

A significant challenge of flash storage is GC overheads having unpredictable performance penalties for the host system [151, 310], resulting in large tail latency [55]. Dealing with, and aiming to minimize required garbage collection for the flash device is a key challenge in integrating flash storage. Naturally, as flash storage does not provide in-place updates, data is written in a log-based fashion, sequentially in the flash blocks. Therefore, over time as data is overwritten, the blocks contain an increasing number of invalid pages that must be erased to free space. However, as the block also contains valid data, and the erase unit is a block, the still valid pages are moved to a new block, such that the old black can be erased.

In a LFS updates to file data are written at the head of the log, resulting in the parts of the file that are updated to be located in a different flash block than the parts of the original file data. Furthermore, GC causes file data to moved around the storage space as well, resulting in scattering of parts of files, referred to as *fragmentation*. Figure 3.6 shows the resulting fragmentation for several files that occurs over time from file updates and GC. Fragmentation results in increased read time [35], due to files being in non-contiguous regions, and introduces increased garbage collection overheads due to the failed grouping of data. Ji et al. [119] show an empirical study on fragmentation in mobile devices, identifying that fragmentation introduces performance degradation due to increased I/O requests, and it further produces increased pressure on host caching. The

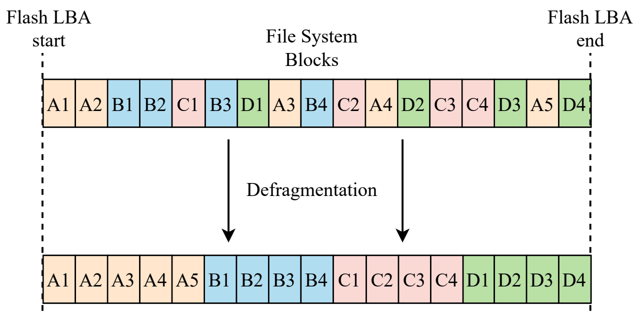

Flash LBA start
File System Blocks
Flash LBA end
A1 A2 B1 B2 C1 B3 D1 A3 B4 C2 A4 D2 C3 C4 D3 A5 D4
Defragmentation
A1 A2 A3 A4 A5 B1 B2 B3 B4 C1 C2 C3 C4 D1 D2 D3 D4

Figure 3.6: Illustration of single file fragmentation on the storage over time, as parts of file data are overwritten and, due to the LFS design new data is appended at the head of the log.

Mechanism	File Systems
Reducing Write Traffic (§3.5.3)	[67, 105, 108, 120, 176, 186, 214, 300, 329]
Aligning the Allocation Unit (§3.6.1)	[226, 283]
Data Grouping (§3.6.2)	[167, 187, 246, 322, 323]
GC Policies (§3.6.3)	[1, 90, 222, 317]
Coordinating the Software Stack Layers (§3.6.4)	[174, 175, 302, 320]

Table 3.6: Mechanisms for file systems to deal with GC overheads from flash storage, and the respective file systems that implement a particular mechanism. Green highlighted table cells depict previously discussed mechanisms with their respective section.

effects on caching are due to increased difficulty of prefetching data, since it no longer is in contiguous physical ranges, but scattered throughout the physical space. Therefore, prefetching cannot bring correct data into the caches, resulting in an increase in cache misses.

In addition to increased I/O requests for reading and rising cache pressure, increased garbage collection is caused when frequently modified files or file fragments, referred to as the *hot data*, are in the same block as rarely accessed files, referred to as the *cold data*. When hot data is modified, its flash pages are invalidated. Once enough flash pages in a block are invalidated, it can be erased. However, if it is co-located with cold data, the cold must be copied to a free space, as it is still valid. If this cold data is then again co-located with hot data, the modifications of the hot data cause the block to be cleared during GC, which requires the cold data to be moved again. Therefore, co-locating hot and cold data in the same physical erase unit results in significant GC increase due to the unnecessary moving of the cold data.

Reducing the write traffic to the storage device, an effective method to handle the asymmetric flash performance (see Section 3.5.3), is a solution to minimize fragmentation and reduce possible future GC overheads. However, additional mechanisms are required for effective GC management. With fragmentation being a significant contribution to increased garbage collection, avoiding it is a key objective to reducing garbage collection overheads. Fragmentation is classified into three different types [253]. Firstly, *single file fragmentation*, where data in a single file is dispersed over the storage (as is shown in Figure 3.6). Secondly, *relevant file fragmentation*, where files that are relevant to each other and should be grouped together are split over the storage, such as co-locating hot and cold data in the same erase unit. Lastly, *free space fragmentation*, where the file system has a large amount of small free space, because of deletion of dispersed small files. The cause of fragmentation occurring over time is referred to as *file system aging* [265]. While several tools exist that implement *defragmentation* [92, 149, 224], additional mechanism can be utilized to avoid fragmentation and GC overheads. Table 3.6 depicts the mechanisms for file systems to deal with and minimize garbage collection overheads. The process of countering the different types of fragmentation is commonly referred to as *storage gardening* [143, 144], and for file system development various aging tools exist in order to generate real-world file system workloads and simulate file system aging [51, 132].

3.6.1. Aligning the Allocation Unit

GC is a result of having to move valid blocks in the erase unit to a free space, in order erase the flash block. While data grouping allows to align the validity inside the block, such that blocks are likely to be updated within close proximity, multiple files may be co-located in the same block. Therefore, a similar method is to align the allocation unit of data blocks for a single file to the erase unit, resulting in only a single file being located in a block. Such a mechanism is implemented in URFS [283], which aligns the data allocation unit for large files to the flash erase unit, allowing files to be erased as a single unit, limiting required GC. However, as the erase unit of flash can be several hundreds MBs, resulting in significant over-allocation for small files, it makes such a mechanism only beneficial with large files. NAMU [226] similarly showcases a file system that aligns its content with the requirement of large files. Focused on the multimedia domain, where files have the particular characteristics of rarely being modified, and if removed all file data blocks are erased in one unit, GC in NAMU is done at the granularity of a file. In addition to improving on GC, the memory requirements for mapping tables are also minimized. For generic file systems that vary in file characteristics, *Adaptive Reserved Space (ARS)* [312] minimizes the issue of over-allocating space by allocating in a smaller unit of 2MB (a single segment). File data is written to the space until it is exhausted, upon which a new segment is allocated. While this does not map entire files to the flash erase unit, it allows writing of file data sequentially for each segment, eliminating fragmentation to a degree. Therefore, the resulting reduction in fragmentation provides benefits in reducing the amount of data that is required to be moved during GC.

3.6.2. Data Grouping

A key circumvention method for fragmentation relies on grouping of related data. Most commonly this is applied in the type grouping data by its access and modification frequency into hot and cold data, however other less commonly used groupings based on *death-time prediction* exist [30]. Commonly more classifications than simply hot and cold are utilized for more effective grouping. A plethora of methods for grouping data in such a way have been proposed, which we split by its application type into several groups.

Data Type Grouping

Common write patterns in storage systems follow a bi-modal distribution, where many very small write requests and a numerous very large write requests are issued [32]. This stems from the fact that small changes are caused by metadata updates, which occur the most frequently, whereas large changes are file updates. Given that metadata is more likely to be updated frequently, separating metadata from data improves on required garbage collection. Specifically, since metadata is often updated even if the data is not updated, for example in scenarios where the file attributes (access time, permissions, etc.) are updated, or the file is moved. Therefore, based on the request size, the data can be classified to be metadata and be grouped accordingly. F2FS also groups based on the data type, where metadata is considered to be always hot data, which is implemented by similar file systems [187].

Dividing of file system data is similarly applied in ELOFS [322, 323], which splits the flash storage two partitions, where a directory partition contains the data of directory entries, and a data partition contains the file system data, which is compacted with the inodes. Jung et al. [131] propose the addition of classifying data based on the *Process Identifier (PID)*, as a process is likely to generate similar access patterns and data types throughout its lifetime, classifying by the PID allows to indirectly infer a data type. A similar data type separation is implemented by Fstream [246], for which the authors modify ext4 [25] and xfs [273] to map different operations to different streams on a stream SSD. Ext4Stream, the modified ext4 to support streams, maps different metadata operations to different streams, including the journal writes for consistency, the inode writes, as well as different streams for the directory blocks and the bitmaps (inode and block). Furthermore, it utilizes different streams that can be created for different files and for different file extensions. The goal of such streams is to map particular files, such as `LOG` files (temporary hot data) for key-value stores for example to a particular stream, separating its access patterns from that of other files and file system data. Similarly, the modified XFStream utilizes different streams for the log, inodes, and specific files.

Dynamic Grouping

While grouping is an effective method for minimizing GC, it commonly relies on a static definition on classification targets for the number of hot/cold degrees to classify to (e.g, hot/warm/cold). Shafaei et al. [259] identify that the majority of hot/cold data grouping methods fail to account for the accuracy in the hot/cold grouping mechanism, as well as relying on an individual classification of each LBA, making the management of increasingly larger flash storage difficult. Therefore, Shafaei et al. [259] propose an extent-based temperature identification mechanism. It is based on the density stream clustering problem [39, 78, 112, 121], which is a common approach of classification in artificial intelligence and stream processing, however has not been applied to storage before. The density-based stream clustering groups data in a one dimensional space as the data arrives, hence its applicability for stream processing.

Applying this method to storage, the one-dimensional space is the range of LBAs, and extent-based clustering splits the available space into a number of extents to group by. Initially, the entire space is a single extent and as writes occur the extent is split into smaller extents with different classifications. Over time as more writes are issued, extents are expanded and merged (merging of extents with the same classification). Such a grouping allows a more detailed grouping due to the increase in classification targets, compared to binary hot/cold grouping. However, an evaluation by Yang and Zhu [313] on a configurable garbage collection policy, where the number of hotness classification targets is evaluated, shows that various hotness classification targets can significantly increase the write amplification during garbage collection.

LBA Hotness Classification
Different to grouping data based on its type, hotness can be classified on the LBA based on its access frequency. The naive approach at modeling hotness for each LBA is with table-based classification model [102]. This however comes at a high overhead cost, as an entry for each LBA is needed, which becomes increasingly expensive as flash storage grows. Therefore, a more non-trivial method is based on two-level LRU classification [33], with two LRU lists. Upon an initial LBA access, the LBA is stored in the first list, and a subsequent access moves it to the next list, which is referred to as the *hot list*. Therefore, if a LBA is in the hot list, it is considered to be frequently accessed.

A different approach is implemented by *Multiple Bloom Filters (MBF)* [116, 221], which uses bloom filters to identify if a LBA is hot. Bloom filters rely on a hash function that, given an input such as the LBA, provide an output which is mapped to a bit array, and sets the bit to true. Therefore, if a LBA is accessed, applying the hash function sets the respective bit in the array to true, and checking if a LBA is hot simply applies the hash function and checks if the bit is set. However, depending on the length of the array, multiple LBAs can map to the same bit location, as proven by the pigeonhole principle [3], resulting in false positive hotness classifications for a LBA. To avoid frequent false positive classifications, MBF utilizes multiple bloom filters at the same time. With multiple bloom filters, the same amount of arrays exist, applying all bloom filter hash functions to the LBA, and setting the respective bit in each of the arrays. As a result, collisions on all bloom filter hash functions are less likely, minimizing the possibility for false positives.

Similar to MBF, Kuo et al. [159] present a hot data identification method by using multiple hash functions and a hash table. Upon a write, the LBA is hashed by multiple hash functions, and a counter for each hash function is incremented in a hash table. To check if a LBA is hot, the LBA is hashed and a configured H most significant bits of the resulting hash table indicate if the LBA is hot if they are non-zero, as the counter is increased on accesses the most significant bits are only non-zero if the LBA is frequently accessed. Multiple hash functions are used for the same reason multiple bloom filters are used in MBF, to avoid false positive classifications. Lee and Kim [170] provide a study into comparing performance of two-level LRU, MBF, and *Dynamic Data Clustering (DAC)*, which are similar to the density-based stream clustering by Shafaei et al. [259] (discussed in the prior Section 3.6.2.2). The authors show that on the evaluated synthetic workloads DAC provides the highest reduction in write amplification factor, which in turn leads to a decrease in GC overheads.

Unlike all prior approaches basing classification on the access frequency directly to identify hotness, Chakraborttii and Litz [30] propose a temporal convolutional network that predicts the *death-time* of a LBA, based on modification history. This allows to more optimally group data based on the death-time of individual LBA, which has been shown to be an effective grouping mechanism [98]. Grouping related death-time LBA reduces the required garbage collection, as blocks containing LBAs with similar death-times are erased together, which in turn reduces the write amplification (solving **FIC3**).

3.6.3. Garbage Collection Policies

While data grouping provides benefits of co-locating data based on their update likelihood, an additional essential part of garbage collection is the policy of victim selection for segments to clean. Since during garbage collection a segment to clean is required to be selected, where all still valid data in the segment is moved to a free space, selecting a victim becomes non-trivial. With the importance of data grouping with hotness for effective GC, conventional GC policies, such as greedy and cost-benefit lack their inclusion. SFS [1] proposes the *cost-hotness* policy to account for the hotness of segments instead of the segment age, better incorporating the data grouping into victim selection. The cost-hotness is calculated as

$$\text{cost-hotness} = \frac{\text{free space generated}}{\text{cost} * \text{segment hotness}} = \frac{(1-u) * \text{age}}{2u * h}$$

where the cost considers reading and writing the valid blocks (equivalent to $2u$) with the segment hotness. A further GC policy that is based on the cost-benefit policy, is the *Cost-Age-Time (CAT)* policy [41]. It extends cost-benefit by including an erase count for each block, improving wear leveling (solving **FIC5**).

The majority of GC policies have a fixed algorithm, limiting configuration possibility. The d*-Choice* algorithm [286] is a configurable GC policy that combines greedy selection with random selection. The tunable parameter d defines the number of blocks to be selected randomly out of the N total blocks. Therefore, configuring $d = 1$ results in fully random victim selection, as a single block is randomly selected from the total blocks, providing effective wear leveling through randomness [313] (solving **FIC5**). Configuring $d \rightarrow \infty$ selects a larger subset of blocks to use greedy selection on, such that $d = N$ is equivalent to fully greedy victim selection, allowing to provide the lowest cleaning latency. Configuration of d thus allows to define the tradeoff between wear leveling and performance.

An evaluation by Yang and Zhu [313] of the algorithm shows the significance of the number of hotness classification targets that are utilized, and the configuration of the d parameter, where various hotness classification targets can significantly increase the WA during GC. Similarly, *File-aware Garbage Collection (FaGC)* [307] is a GC policy that maintains an *Update Frequency Table (UFT)* for each LBA of a file, in order to group valid pages in the victim block based on the access frequency when these are copied to a new block during GC. This is similar to grouping hot and cold data, but at the level of GC for each LBA in a file. SFS [1] implements a GC policy that accounts for data grouping, with a lower overhead of having to maintain an UFT for each file. It maintains a hotness classification for each block, and combines blocks with k-means clustering [96], into groups with similar hotness classification.

The GC cost of foreground cleaning in F2FS can take up to several seconds [317], as it is not a preemptive task. Implementing preemptive scheduling in kernel file systems is challenging, as during the preemption the kernel has to store the file system state to continue after higher priority tasks have finished, and the state being restored when returning depicts possibly outdated data. Due to the continued writing on the file system, restoring the prior state may no longer be valid, as ongoing writes may have changed file system metadata. OrcFS [317] implements *Quasi Preemptive Segment Cleaning (QPSC)*, which sets a maximum time interval T_{max} (default of 100ms). After cleaning a segment it checks if the timer has expired, and if so it checks if outstanding writes are present from the host. If there are outstanding writes, the locks are released and the write is executed, and if there are no outstanding writes the next segment can be cleaned and the timer is reset. This allows any host write command to not encounter a segment cleaning overhead higher than T_{max}.

A further drawback of segment cleaning with LFS, in particular F2FS, is that the modification of metadata during segment cleaning requires a checkpoint to be created after each segment clean. This constitutes to a significant overhead for the segment cleaning process. To avoid the excessive checkpointing after segment cleaning, *Segment Cleaning Journal (SCJ)* [90] adds support to F2FS to journal metadata updates made during segment cleaning, instead of creating a checkpoint. This journal is stored in a journal area, which delays the updating of the original metadata until the journal becomes large enough or the checkpointing time interval is reached. However, metadata still points to old invalid data blocks (referred to as *pre-invalid blocks*), which requires that data only be invalidated once the metadata is updated by the SCJ. Therefore, SCJ implements an adaptive checkpointing that evaluates the cost of checkpointing to flush the metadata updates and the accumulation of pre-invalid blocks, and checkpoints if its cost is lower.

In addition to utilizing GC policies to reduce its overheads, the policy can further be used to incorporate management of fragmentation. Park el at. [225] propose to use a *Valid Block Queue (VBQueue)* in which valid blocks are sorted during garbage collection. Typically, a victim segment in the LFS is selected for cleaning, valid data is copied to the free space at the log head, and the old segment is erased. The VBQueue is added such that after a victim segment is selected, it is copied into the VBQueue, where the blocks it contains are sorted by the inode number. Then are the valid blocks written to a new segment and the old one is erased. This sorting allows maintaining of file associated blocks together based on their inode number.

A different approach to mitigate GC overheads is to design the GC procedure such that accesses do not suffer from high tail latency when GC is running. TTFlash [308] achieves this through several mechanisms. It implements *plane-blocking GC*, which limits any resources that are blocking I/Os to only the affected planes on the flash. However, this leads to blocking of requests to the GC affected planes. Therefore, TTFlash implements *rotating GC* that only runs at most one GC operation on a *plane group*. The plane group assignment is based on the possible parallelism of the device, such as plane- and channel-level parallelism. This ensures that a plane group is never blocked for more than a single GC operation, implying that any request will not be blocked for more than one GC operation.

On embedded devices, and in particular mobile devices, in order to save energy the device suspends

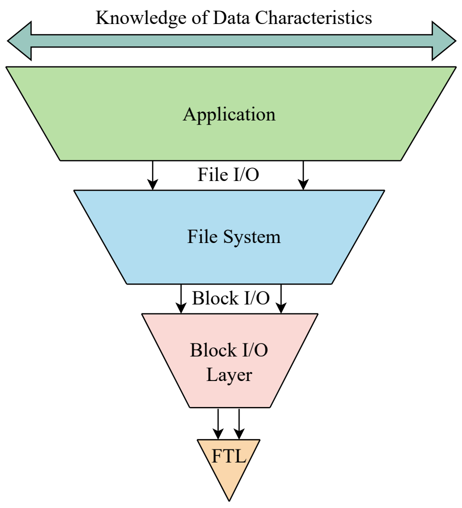


Figure 3.7: Layers in the storage software stack across which data specific knowledge decreases as I/O is passed downwards to the FTL, representing the semantic gap [321] with flash storage.

all threads when not needed (e.g., when the mobile screen is turned off). This implies that all file system threads are also suspended, which means the file system cannot run the background GC during these inactive sessions. Therefore, *Suspend Aware Cleaning (SAC)* [222] is an addition to LFS, which is the time between the suspend initiation and the suspending of the file system threads, also referred to as the *slack time*. It uses this slack time to run background GC, however it does not write any data on the flash storage, but instead selects a victim block and brings the still valid pages in the page cache and marks these as dirty. As a result, all pages in the victim block to be dirty, which allows it to be erased. This process is referred to as *virtual segment cleaning*, which however is not run every time the screen is turned off, but rather based on the device utilization, which is called *utilization based segment cleaning*.

3.6.4. Coordinating the Software Stack Layers

The various layers in the storage software stack introduces several redundant operations, such as GC that is run in the LFS and GC run on the storage device. This duplicate work leads to significant performance impacts [310], requiring a co-design of the storage device software and the file system. Qiu et al. [242] show that the co-design of FTL and the file system show benefits of reduced memory requirements for L2P mappings, and increased device parallelism that can be exploited with better file system knowledge of the storage characteristics. In addition to duplicated work, information about data characteristics are lost across the various, since in order to integrate communication across the layers, each utilizes an interface for the other layers to communicate with. However, the expressiveness of the interfaces limits the capability to communicate data characteristics across the layers, as Figure 3.7 illustrates. Applications have the highest knowledge of data characteristics, how it is best allocated on the flash and managed in order to reduce GC, however cannot forward all of this information to the file system due to the lack of such APIs. Similarly, the file system may group specific data together, however cannot forward this information to the block layer, and similarly the FTL may take the submitted I/O requests and organize these different on the flash storage. This *semantic gap* between the storage device and storage software is a result of the device integrating into the existing block I/O interface [321], however failing to represent the flash-specific characteristics. Such mismatch between storage device and its accessing interface, requires storage software to enhance capabilities to pass information across the layers to avoid increasing the semantic gap across the layers.

DualFS [302] utilizes the custom integration with OCSSD to merge the garbage collection of the file system with that of the FTL, and present this scheme as *global garbage collection*. ParaFS [320] implements a similar coordination of file system GC and FTL GC. A different approach taken by Lee at al. [175] is to modify the block interface with the flash characteristics, moving responsibility directly to the file system, or other application built on top of it. The resulting interface called *Application Managed Flash (AMF)* exposes a block

Mechanism	Amplification Type	File Systems
Write Optimized Data Structures (§3.5.1)	WA	[67, 107, 123, 249, 283]
Write Buffering (§3.5.2)	WA	[125–127, 138, 197, 227–229]
Reducing Write Traffic (§3.5.3)	WA, RA, SA	[67, 105, 108, 120, 176, 186, 214, 300, 329]
Data Grouping (§3.6.2)	WA, RA	[167, 187, 246, 322, 323]
GC Policies (§3.6.3)	WA, RA	[1, 90, 222, 317]
Space Optimized Data Structure (§3.7.1)	SA	[167, 185, 195]
WA with Coarse Granularity Flash Mappings (§3.7.2)	WA	[147, 171, 317, 317]
Reverse Indexing (§3.7.3)	WA	[195]

Table 3.7: Mechanisms for file systems to deal with I/O amplification caused by flash storage integration, and the respective file systems that implement a particular mechanism. Green highlighted table cells depict previously discussed mechanisms with their respective section.

interface that does not allow overwriting unless an explicit erase is issued for the blocks. This matches the flash requirement that prior to overwriting data it has to be erased. The interface is implemented as a custom FTL, called AFTL, on top of which the ALFS file system is built. This avoids the duplicate garbage collection of the FTL and the file system, as the garbage collection of the ALFS erases blocks during garbage collection, informing the FTL to erase the physical block.

Co-designing the FTL and the file system allows removing uncoordinated duplicate work, and coordinate the flash management. To this end, Lee et al. [174] present a redesigned I/O architecture, called REDO, which avoids the duplicate operations from file system and FTL by implemented the new framework directly as the storage controller, and building the *Refactored File System (RFS)* on top of the new controller interface. By combining the file system operations with the storage controller, the file system is responsible for running GC and managing the storage by maintaining the L2P mappings.

3.6.5. Summary

With GC being a significant performance bottleneck for flash storage, adequate data grouping and effective GC policies allows to reduce and manage the GC. Furthermore, the complexity of storage system software lacks support for effective APIs for coordination on data placement between the storage stack layers. Such coordination, allows to reduce the semantic gap and eliminate redundant and duplicated operations of the various layers.

3.7. FIC-3: I/O Amplification

Several of the applied mechanisms for dealing with asymmetric flash performance and garbage collection are key mechanisms to eliminate I/O amplification. Table 3.7 shows the different mechanisms that can be applied to lower the various types of I/O amplification, including *Write Amplification (WA)*, *Read Amplification (RA)*, and *Space Amplification (SA)*. These include the benefit of write buffering, which in the case of write requests that are smaller than the flash allocation unit, avoids the unnecessary WA to fill the flash allocation unit. Similar buffering as is employed in WODS, such as the B^{ε}-trees, allows decreasing the WA. Another key mechanism is reducing the generated write traffic, which minimizes WA, RA, and SA through deduplication, compression, delta-encoding, and virtualization. Similarly, the discussed methods of grouping data, avoiding fragmentation, allows reducing the RA to locate data, and furthermore limits GC overheads and GC caused WA. Throughout this section we evaluate the additional methods for limiting the various types of I/O amplification.

3.7.1. Space Optimized Data Structures

Similar to the design of a WODS, data structures with particular focus on optimally utilizing available space are a mechanism to deal with SA. A commonly applied method for file systems to optimize space utilization is to possibly embed file data in the inode. Commonly the allocation of the file system inode occupies at least a block, such as 4KiB in F2FS, which however is more space than file metadata requires. Therefore, several bytes (~3.4KB in F2FS) are free, which are used to inline file data in the inode. This particularly allows for small files to entirely fit into the inode, avoiding writing the inode and leave the unused space empty, and additionally write an additional data block, which also has free space. Different to inline data, ReconFS [195] uses an inode with a size of 128B, allowing to place numerous inodes in a single flash page. With such an inode size, writing each inode change directly requires filling the flash page with unnecessary data. Therefore, ReconFS implements a *metadata persistent log*, in which metadata changes are logged and compacted to align with pages, and are only written back to the storage when evicted or checkpointed, in order for the file system to remain consistent.

Similar to effective tree-based WODS, the radix tree is a space optimized tree variant, that is commonly used as the directory and inode tree for file systems [167, 185]. The directory tree is commonly constructed and maintained in memory and written to the persistent flash storage. Its space optimization revolves around merging of nodes that have a single child with that child node. This eliminates the need for an individual node that is assigned to each child, lowering the space requirement. As a result, the radix tree is also referred to as a compressed tree, due to the compression of single child nodes.

3.7.2. Coarse Granularity Flash Mappings

A similar goal of file systems is to reduce the amount of memory that is required for the mapping table to maintain the L2P mappings. L2P mappings are commonly persisted periodically, from the storage device RAM to the flash storage, such that in the case of system shutdown or the device is unplugged, upon reconnection the mapping information can be recovered. Hence, the mapping information similarly requires flash pages to be stored. A common solution is to increase the granularity of the mapping table (e.g., block-level mapping instead of page-level mapping), requiring fewer mappings. MNFS [147] manages flash storage with page-level and block-level mapping, depending on the update frequency. Metadata is updated more frequently and therefore utilizes a page-level mapping compared to larger mapping granularity for data. OrcFS [317] similarly utilizes a page-level for metadata, and a superblock-level mapping for data, which represents several flash blocks. The allocation unit is called a superblock as it consists of multiple blocks (not to be confused with the file system superblock). Furthermore, logical addresses are mapped to the same physical addresses in the data partition, requiring no mapping table, and file system sections are aligned to the superblock unit. Therefore, OrcFS only requires a block allocation information for each file in the superblock, which are stored in the inode block in the metadata area.

However, this comes at the cost of having a larger allocation unit, and if a host write is smaller than the allocation unit it causes WA, due to the partial flash page write when the flash page size and the allocation unit are not aligned [171, 317]. OrcFS [171, 317] implements *block patching* to solve this issue. It takes write requests that are smaller than the flash page size and pads the remaining space with dummy data to align the write request to flash page size. This mechanism avoids copying data if a flash page is partially written, and the next LBA in the same flash page is written, which triggers a copy of all LBAs in the flash page followed by writing the LBA for the new write. For instance, for a flash page containing 4 LBAs, if LBAs 1-3 are written by one request, the first 3 LBAs are mapped to the data and the fourth holds dummy data, such that the page is fully filled. If a second request to LBA 4 is issued, it cannot fill the flash page as it has already been written. Therefore, to write the newly written data after the already written LBAs, it must copy LBAs 1-3, append the new write to LBA 4, and write the 4 LBAs to a new flash page. Adding of dummy data to fill pages reduces the WA, which would be caused by copying of all data in the flash page, as it now avoids copying the added dummy data on consecutive writes. While reduction in WA are presented, the adding of dummy data nonetheless adds WA to fill the flash page. However, as latter updates require less data written, and the importance of data grouping indicates, maintaining related data in the same flash page is more beneficial and possibly decreases future WA. Related data remains in the same flash page, as only the valid data in flash pages is copied on writes, as opposed to copying the entire flash page, introducing copied dummy data.

3.7.3. Reverse Indexing

As file system metadata is commonly maintained in a tree-based data structure, updates to metadata in the leaf nodes can propagate changes to the root node, known as the Wandering Tree Problem [17]. Due to the update of leaf metadata, such that when file data is modified, the metadata points to the new location of the file data, causing new metadata to be written, which in turn requires its parent to be updated to point to the new location of the metadata. This propagates up to the root note, causing significant WA. F2FS utilizes a table based indexing, with the NAT, such that only a table entry is required to change to update the data location, and metadata points to the table entry to locate the data. ReconFS [195] utilizes an inverted indexing tree, which also avoids the wandering tree problem. With such a tree, each node points to its parent node, in-

Mechanism	File Systems
Aligning the Allocation Unit (§3.6.1)	[226, 283]
Clustered Allocation & Striping (§3.8.1)	[1, 4, 67, 201, 202, 227, 317, 320]
Concurrency (§3.8.2)	[135, 138, 166, 175, 185, 320]

Table 3.8: Mechanisms for file systems to exploit flash parallelism capabilities, and the respective file systems that implement a particular mechanism. Green highlighted table cells depict previously discussed mechanisms with their respective section.

stead of the parent node pointing to a child node. Therefore, upon address change of a child node, the parent does not need to be modified, since the child node points upwards to the parent node. Similar mechanisms are utilized in FTL design [193], where indexing data is written in the OOB space of the flash page from the data, in order to locate its metadata. In order to avoid increased scan times on failure recovery, which can no longer traverse the tree from the root, the updated pages are tracked to locate the most recently updated valid page, which is then periodically included in the checkpointing to ensure consistency.

3.7.4. Summary

The introduced I/O amplification of flash storage, particularly a result of GC, requires careful consideration to reduce the write requests, such that the flash device lifetime can be extended. Several of the previously discussed mechanisms, such as reducing write traffic and utilizing effective GC policies, aid in reducing the I/O amplification, however furthermore particular data structures optimized for space utilization similarly provide efficient methods for reducing I/O amplification.

3.8. FIC-4: Flash Parallelism

With the capabilities of flash storage relying largely on increased parallelism, several existing mechanisms are leveraging these. Depending on the level of flash integration, different mechanisms are possible, where at the SSD integration the host has not control over the possible physical parallel utilization of flash, as the FTL controls this, however deeper flash integration at the custom and embedded levels provide more possibility. Table 3.8 depicts the various mechanisms to aid the utilization of flash parallelism and exploit the physical characteristics of flash.

3.8.1. Clustered Allocation & Striping

As host software has no direct access to flash storage with SSD, the FTL implements and manages all device-level parallelism. The possibility for the host to utilize flash parallelism comes from aiding the FTL in providing large enough I/Os such that the FTL can stripe data across flash chips and channels. This is achieved with *clustered blocks/pages* [153], where blocks or pages on different units (such as blocks on different planes) are accessed in parallel. The FTL can possibly stripe data across these clustered blocks, given that the I/O request is large enough to fill the clustered unit. Such a mechanism aligns with prior discussed aligning of the allocation unit (Section 3.6.1) to a physical unit to reduce I/O amplification, such as making the file system segment unit a multiple of the flash allocation unit. SFS [1] takes advantage of the achieved device-level parallelism with clustered blocks by aligning segments to a multiple of the clustered block size. During garbage collection SFS ensures that cleaning of segments, that do not have enough blocks to fill the clustered block size, is delayed until enough data is present.

SSDFS [67] utilizes the custom flash integration to map data allocation of segments to the unit of a *Physical Erase Block (PEB)*, where the PEB is split over the parallel unit on the device, such as previously mentioned parallel erasing of flash blocks over channels. Therefore, utilizing PEBs over the parallel unit allows striping writes into a segment over the varying channels, increasing the device parallelism. In order to achieve this, I/O requests have to be large enough such that they can be striped across the channel and fill the PEBs mapped to the segment. For this, SSDFS utilizes aggressive merging of I/O requests to achieve the larger I/Os that can be striped across the parallel units (solving **FIC7**). Instead of merging I/O requests, OrcFS [317] increases its file system allocation unit to a *superblock* (not to be confused with the file system superblock), which represents a set of flash blocks. These flash blocks are then split over the parallel units of the flash storage by the file system for increased parallelism. The file system utilizes a custom flash integration, and hence is capable of managing the parallelism of the flash storage.

The large allocation unit however introduces increased GC overheads, and block-level striping has lower performance than page-level striping [320]. Therefore, ParaFS [320] implements a 2-D allocation scheme, with page-level striping over the flash channels, where striping is also based on the data hotness, hence having a 2-dimensional allocation scheme. Different groups are assigned for the hotness levels, where writes are issued to the corresponding hotness group striped over the flash channels. Several other file systems implement variations of striping across different parallel units on flash storage [4, 201, 202, 227]. These mechanisms are also present in the design of storage applications, such as key-value stores [293, 321].

3.8.2. Concurrency

Similar to increasing the data allocation unit, concurrency is a mechanism to exploit the flash parallelism. LFS design relies on a single append point at the head of the log, in the simplistic implementations, depending on methods such as locking to ensure only one write is issued at the log head. This has a significant impact on the performance where other I/Os are idle while a single I/O completes. F2FS however suffers from severe lock contention overheads, where the performance of the multi-headed logging is nearly fully deprecated due to the serialization of data updates [185]. In particular, as data has to be written persistently before inode and other metadata can be written. Furthermore, F2FS suffers from contention of the in-memory data structures, for which it uses reader-writer semaphores for read and write operations from the user (termed *external I/O operations*), and reader-writer locks for writing of checkpoints and other metadata (termed *internal operations*) [185]. As lock counters are shared among all cores, cache coherence adds a significant overhead that increases with more cores. Max [185] extends F2FS to increase the concurrency scalability with three main modifications.

Firstly, in order to eliminate cache coherence overheads, it introduces a *Reader Pass-through Semaphore (RPS)* that uses a per-process counter. Secondly, the shared data structures in memory are partitioned by the inode, such that concurrent accessing does not require locking on parts of the radix tree, but instead on an inode basis. Lastly, it utilizes multiple independent logs, called a *Minor Log (mlog)*, which are accessed concurrently. The different between mlog and multi-headed logging in F2FS is that atomic data blocks are mapped to the same mlog, eliminating the need to ensure concurrency control across different logs. Ordering for persistence, ensuring data blocks are written before metadata, is delegated to the recovery mechanism using a global versioning number in each inode to identify ordering across mlogs, and recover the most recent version number in case of a system crash. These mechanisms eliminate much of the needed concurrency control, which sequentialized major parts of operations and hindered multicore scalability.

With this increased concurrency capabilities, the file system can issue more I/O requests to the device, allowing to leverage a higher degree of on-device parallelism. Similarly, DevFS [138] utilizes the parallel capabilities by exploiting the high number of I/O queues supported by NVMe. It maps I/O queues to individual files, allowing single file operations to submit I/Os concurrently without interfering on the I/O queue, therefore increasing the per-file concurrency as well. Likewise to the concept of mlog, SpanFS [135] maps files and directories to different *domains*, such that individual domains can be accessed in parallel. Such methods have a higher lock granularity, where concurrency below the lock granularity is not possible, as a single process is holding the lock. Therefore, instead of holding locks for individual inodes or files, preventing concurrent writing to the same inode or file, Lee et al. [166] extend F2FS to utilize *Range Locking (RL)*, in which ranges of a file are locked, and different ranges can be written concurrently. Therefore, it provides the possibility for intra-file parallelism.

ALFS [175], exploits the flash parallelism by mapping consecutive file system segments to the flash channels and utilizing different I/O queues for each flash channel. Similarly, ParaFS [320] implements *parallelism-aware scheduling*, which also maintains different I/O queues for each channel. However, it extends this concept by using a *dispatching phase* and a *request scheduling phase*. The dispatching phase optimizes write I/Os by scheduling I/O requests to the respective channels based on the utilization of the channel, such that the least utilized channels receives more requests. All requests are assigned a weight, which indicates their priority in the queue, where read requests weight is lower than that of write requests, because of the asymmetric performance of flash storage. During the request scheduling phase the scheduler assigns slices to the read and write/erase operations in the individual queues, such that if the time from the slice of a read operation is up and the queue contains no other read requests, a write or erase is scheduled, based on a fairness formula that incorporates the amount of free space in the block and concurrently active erase operations on other channels. This allows to minimize the erase operations on the flash, giving always free channels to utilize and maintain a fair schedule between write and erase operations.

Mechanism	File Systems
Write Optimised Data Structures (§3.5.1)	[67, 107, 123, 249, 283]
Reducing Write Traffic (§3.5.3)	[67, 105, 108, 120, 176, 186, 214, 300, 329]
GC Policies (§3.6.3)	[1, 90, 222, 317]
Write & Read Leveling (Sections 3.9.1 and 3.9.2)	[173, 175, 187, 302]

Table 3.9: Mechanisms for file systems to deal with wear leveling of the flash storage, and the respective file systems that implement a particular mechanism. Green highlighted table cells depict previously discussed mechanisms with their respective section.

3.8.3. Summary

Due to the architecture of flash storage providing a high degree of parallelism, numerous methods are employed to leverage these parallel units in order to maximize the performance. Depending on the level of flash integration, particular design choices can be made, such as clustered allocation and striping can be achieved with flash SSD integration, by providing large write I/O requests such that the FTL can stripe the data across parallel units. With a higher degree of control over the flash storage, file systems can directly rely on utilizing concurrency to leverage the parallelism of flash storage.

3.9. FIC-5: Wear Leveling

As flash cells wear out over time, it is important to utilize the flash evenly to avoid burning out particular flash cells faster than others. The possibility for ensuring even wear at the different levels of integration is limited, as at the SSD flash integration level, the FTL handles all wear leveling, without host considerations. However, similar to prior flash integration challenges, several mechanisms are nonetheless applicable. In particular, reducing the write traffic to the flash device, as less writing incurs less flash wear, and particularly flash-specific data structures inherently provide a degree of wear leveling. Based on the sequential write requirement, data structures, such as the LFS must write sequentially in an append-only fashion, which evenly writes the space. At closer to flash integrations, where host systems have more control over the flash management, there are particular mechanisms to ensure better flash wear. Table 3.9 depicts the methods we discuss in this section for enabling increased wear leveling for file systems.

3.9.1. Write Leveling

Several flash integration challenges proved data grouping to be an effective method for dealing with GC overheads and I/O amplification. However, this can have an effect on the flash storage. In particular the hot data, such as file system metadata which is more frequently updated and written, must be moved across the flash space more than cold data. CFFS [187] therefore switches the allocation areas for metadata and data blocks, such that an erased metadata block becomes a free data block. Therefore, cold data should be placed in blocks that have been written more frequently, whereas hot data should be placed in blocks with a lower write count history. The principle of migrating cold data from less written blocks, which are also referred to as *younger blocks*, to more frequently written, *older blocks*, is referred to as *cold-data migration*, and similarly moving hot data from old blocks to younger blocks is known as *hot-data migration* [31]. These methods are commonly used in FTL implementations due to their simplicity and effectiveness, and similarly in file systems such as ALFS [175].

Wear leveling is an increasingly vital concern on file systems that utilize flash dual mode [173, 302], where it switches the flash level to increase performance for critical I/O requests. Due to the lowering in flash cell level, the same amount of written data requires a larger amount of space, where a switch from MLC to SLC divides the capacity in half, requiring double the space for the same I/O request. Therefore, these file systems include a *write budget* that is maintained for the areas, and dynamically resizes the available lower cell level area, such as decreasing the space if the wear is reaching a threshold. This switches the cell level back to a higher number, allowing to write more with less wear. Additionally, the file systems utilize the wear budget in order to identify if an I/O request should be redirected to the larger cell area, instead of being written to the lower cell level area.

3.9.2. Read Leveling

While write operations are the major cause of flash cells burning out, read operations also pay a toll on flash cells, as the current flash cell technology utilizes flash cells that are only capable of holding very few electrons

Mechanism	File Systems
I/O Operations (§3.10.1)	[283]
I/O Scheduler (§3.10.2)	[240]
I/O Path - User-Space File Systems (§3.10.3)	[138, 190, 283, 318]

Table 3.10: I/O scheduling mechanisms to exploit performance capabilities of flash storage, and the respective file systems that implement a particular mechanism.

(determining the charge of the gate) due to their size [191, 264]. This makes the cells increasingly susceptible to *read disturbance* [189], where reading of a page results in shifting of voltages in nearby cells (typically in the same block), requiring frequent rewriting to ensure the charge stays consistent. In order to control read disturbance, Liu et al. [189] propose to read-leveling mechanisms in the FTL. While their proposal is aimed at FTL implementations, the ideas are applicable to file systems for flash storage devices. With the proposed read-leveling, the read-hot data, that is read more frequently, is isolated from other data pages by placing the hot pages into *shadow blocks*, which contain no valid data, in order to avoid disturbing that data. However, this requires to identify the read-hot pages, where a tracking of read counters for each page would require significant resources. Therefore, a *second-chance monitoring strategy* is proposed, which initially tracks the reads for each block, therefore requiring counters at a higher block granularity, and upon reaching a threshold indicating the block contains read-hot pages, the individual pages in the block are tracked on their read counters. Finally, the pages in these blocks that reach a certain threshold are copied to the shadow blocks. Therefore, this avoids the tracking of read counters for individual pages and only copies read-hot pages into the shadow blocks. While this strategy requires copying of read-hot pages to shadow blocks, it minimizes read disturbance which in turn minimizes the WA it causes.

3.9.3. Summary

In addition to previously discussed mechanisms to reduce I/O amplification and GC, resulting in decreased wear of the flash storage, write leveling, ensuring that write requests are spread across the available storage space, and read leveling are important mechanisms for ensuring the longevity of the flash storage.

3.10. FIC-6: I/O Management

Given that flash storage has the capabilities to achieve single digit μ-second latency, whereas overheads in the software stack, such as context switching in the kernel caused by system calls, can already require μ-seconds to complete [266], making software the dominating factor in overheads [27, 28, 77, 257, 289]. In addition, *interrupts* cause significant overheads for systems. Aimed at slow storage devices, the I/O request is submitted to the device, the context is switched, such that the process can continue with other work, and upon completion of the I/O, the host is interrupted, and the context is switched again. Any added interrupt on the I/O path can cause significant delays [263]. Cache effects are another drawback of context switching, since other work is continued, replacing data in the caches, it requires bringing the replaced data back into the caches after the interrupt and resuming of the prior context. Similarly, it also causes *Translation Lookaside Buffer (TLB)* pollution on the host system. A different approach to submitting I/O requests is with *polling*, which eliminates the need for context switches. With polling, the I/O request is submitted and instead of continuing other work, the process regularly checks the I/O for completion. Using polling for I/O requests has been shown to be a favored method of building application for fast storage devices [60, 158, 311].

Particular for fast storage devices, with the utilization of *synchronous I/O*, to saturate the storage device, additional threads are required, which each submit I/O requests to the numerous I/O queues in the of the storage device. However, this mechanism does not scale efficiently, where each thread must wait until the I/O request is completed, and thread scheduling overheads are introduced. Therefore, with asynchronous I/O the threads do not wait for completion, but instead submit a larger number of I/O requests each, allowing to fill the device I/O queues more effectively. The I/O requests for which a thread as submitted a request, but have not completed, are referred to as *outstanding* or *in-flight* I/O requests. Table 3.10 shows the mechanisms for host systems to better leverage the flash storage performance and minimize overheads. In addition to file systems implementing particular mechanisms, we discuss more general methods applicable to all applications for benefiting from flash storage and enhancing performance.

3.10.1. I/O Operations

Given that particular I/O patterns can have degrading affects on the SSD performance, such as mixing read and write operations, as they share resources on the device, including the mapping table and ECC engine, and furthermore possibly invalidating the cached data in the SSD RAM. Similarly, mixing I/O operations with different block sizes can result in increased fragmentation [283]. As the Linux kernel relies on a submission and completion queue for I/O, user-space frameworks such as SPDK and NVMeDirect provide more flexibility for user-space file systems to design different queues, depending on the requirements. URFS [283] utilizes this possibility to create adaptive queues that can better optimize I/O submissions to the device. Based on the workload characteristics URFS dynamically creates flexible I/O queues (e.g., group by size, read/write operation) to increase SSD performance. Similarly, Borge et al. [203] show with a case study on HDFS performance with SSD, that in order to leverage the capabilities of flash SSD, direct I/O, and increased parallel requests with buffered I/O are needed.

3.10.2. I/O Scheduler

With the possibility for asynchronous I/O to merge and reorder requests, the Linux kernel implements several schedulers, such as *NOOP*, *deadline*, and *CFQ* [100, 209, 238, 270]. NOOP being the least intrusive scheduler only merges I/O requests in its FIFO, but does not reorder them, which is beneficial on devices such as OC-SSD, that require consecutive LBAs. Son et al. [267] showcase the benefits of merging random write requests, regardless of contiguity of the LBAs, in order to better enhance performance with fast storage devices. The deadline scheduler adds to NOOP by utilizing merging and reordering, however also applies a deadline for each I/O request to ensure requests are submitted to the device eventually. Two separate queues, one for read requests and an additional one for write requests are utilized, which are both ordered by the deadline of the request. Another scheduler variant of deadline exists, called *mq-deadline*, which is aimed at multi-queue devices, such as NVMe SSD. *Completely Fair Queuing (CFQ)* implements a round-robin based queue that assigns time slices to each in order to prevent starvation and provide fairness. While these are the common schedulers in the Linux kernel, to see details on all schedulers present in the Linux kernel consult [284]. Such scheduling configuration begs the question on which scheduler is best suited for file systems on flash storage. Several studies into performance of the schedulers exist [270, 319], showcasing that merging of read I/O requests in synchronous I/O provides beneficial performance gains, and similarly the merging of write I/Os in asynchronous I/O shows performance gains.

Qin et al. [240] argue that I/O ordering limits exploiting the parallelism of flash devices. Especially as the Linux block layer does not guarantee particular ordering, flags such as *Forced Unit Access (FUA)*, indicating that I/O completion is signaled upon arrival of data on the storage, and *PREFLUSH*, which before completing the request flushes the volatile caches, have to be set in order to ensure a specific ordering [240]. With file systems, the I/O of metadata and data has a particular ordering, such that metadata can only point to data that exists, needing to ensure that data is written prior to metadata. Removing of I/O ordering allows eliminating this need and better utilize the flash parallelism. Utilizing the OOB area on flash pages, the file system developed by Qin et al., called NBFS, maintains versioning in order to identify out of order updates. Furthermore, updates are done using atomic writes (discussed in Section 3.11). The issuing of FUA requests further implies that its I/Os cannot be merged in the scheduler [240], implying that if a smaller than flash page size FUA I/O request is issued, it is padded to the page size, causing WA. NBFS solves this with its atomic writes that imply that the FUA request does not immediately have to be written to the flash, but instead wait for all data blocks to arrive, which are then used to fill the pages, allowing to reduce the WA (solving **FIC3**).

3.10.3. I/O Path - User-Space File Systems

A mechanism that is gaining significant attention in the research community is the utilization of user-space file systems, bypassing the kernel layers and avoid its associated overheads. These file systems run only in the user-space, as opposed to commonly used file systems (e.g., F2FS) running in kernel space. In addition to the benefit of avoiding kernel overheads, user-space file systems are easier to develop, have increased reliability and security by avoiding kernel bugs and vulnerabilities, and provide increased portability [279]. A widely adopted framework for building user-space storage applications is *Filesystem in USErspace (FUSE)* [274]. It is implemented over a kernel module with which it exports a virtual file system from the kernel, where data and metadata are provided by a user-space process, hence allowing user-space applications to interact with it. Since FUSE is implemented with a kernel module, FUSE based file systems suffer significant performance penalties, requiring more CPU cycles than file systems in kernel space. Particularly contributing to overheads is the need to copy memory between user-space and kernel-space, caused by the way FUSE handles I/O requests [287, 288]. Furthermore, FUSE still suffers from context switching [243, 287, 288] overheads and *Inter-Process Communication (IPC)* between the FUSE kernel module and FUSE user-space daemon [326].

A similar framework for building user-space applications with direct storage access is NVMeDirect [146]. However, it also relies on a kernel driver to provide enhanced I/O policies. SPDK [315] is another framework for building user-space storage applications, however it provides the mechanisms to bypass the kernel and submit I/O directly to the device, by implementing a user-space driver for the storage device. Such a framework allows building high performance storage applications in user-space, which eliminate the overheads coming from the kernel I/O stack.

URFS [283] provides increased concurrency performance by implementing a multi-process shared cache in memory, in order to avoid the kernel overheads of copying data as is present in FUSE. It furthermore helps avoid contention on the storage device. Eliminating of data copy is also addressed in ZUFS (zero-copy user-mode file system) [95], which is a user-space file system for persistent memory, which completes I/O requests by requesting exact data locations instead of copying data into the own address space. A similar user-space file system that implements a shared cache for process is EvFS [318], which is SPDK-based. While this file system can also support multiple readers/writers in the page cache, it only supports these for a single user process. User-space frameworks often provide capabilities to either expose the storage device as a block device, which the user-space application then accesses, or build a custom block device module (e.g., with SPDK, which also has default driver modules such as NVMe). For NVMe devices that support NVMe controller memory buffer management, the file system can manage parts of the device memory. DevFS [138] utilize such an integration to manage the device memory for file metadata and I/O queues.

Different from prior discussed development frameworks, *File System as Processes (FSP)* [190] provides a storage architecture design for user-space file systems. The emphasis of FSP is to scale with the arrival of faster storage, and similarly to other user-space frameworks, minimize the software stack. For this it bases development on running file systems as processes, providing safer metadata integrity, data sharing coordination, and consistency control, as the process running the file system is in control of everything, instead of trusting libraries. Furthermore, FSP relies on IPC for fast communication, which unlike FUSE has a low overhead since it does not require context switching. Inter-core message passing comes at a low overhead and cache-to-cache transfers on multi-core systems can complete in double-digit cycles [266]. DashFS [190] is built with FSP, providing a safe user-space file system with isolation of different user processes, and efficient IPC.

3.10.4. Summary.

With the complexity of the storage software stack and the high performance of flash storage, the storage software stack dominates the I/O latency [27, 28], requiring careful I/O management to enhance the performance. In addition to I/O scheduling, allowing to merge consecutive I/O requests and reduce the overall issued I/O requests, user-space file systems, bypassing the Linux storage stack, avoid the storage stack layer overheads.

3.11. Failure Consistent Operations

An important aspect of file systems is to ensure that in the case of power failure or system crashes, the file system state and its data remain in a consistent state, and can be recovered upon a subsequent mount. While we do not classify it as a challenge of flash storage, we discuss the methods for ensuring failure consistent operations for flash. Most commonly used mechanisms for ensuring this are journaling, CoW, and checkpointing. However, journaling suffers from having to write data twice, once for the metadata log and again for the data log, to ensure full consistency [236, 305], also referred to as the double-write problem [138]. Therefore, file systems commonly only provide metadata consistency by logging only the metadata. Furthermore, power failure is a concern with flash storage, as it has capacitors that can flush parts of the on-device memory buffers, unless more expensive capacitors are used that can flush the entirety of the memory buffers. In the case of power failure, partially written data or metadata updates can result in the file system being in an inconsistent state.

While checkpointing, CoW, and other consistency mechanisms aim to handle these failures and provide recovery after failure, other methods exist for providing interfaces that eliminate partial operations. Particularly, the design of flash storage, such as the available OOB space of pages, provides beneficial options for such implementations. Table 3.11 depicts the methods for ensuring failure consistency with flash storage, and the file systems implementing such methods.

Mechanism	File Systems
Atomic Writes (§3.11.1)	[40, 148, 240]
Transactions (§3.11.2)	[216]

Table 3.11: Failure consistency mechanisms to for flash storage, and the respective file systems that implement a particular mechanism.

3.11.1. Atomic Writes

A possible method for ensuring that operations are not completed partially is through atomic writes. This is important to file systems as an update of data requires its respective metadata to also be updated. Failure of one should result in the other not being completed. A variety of flash devices support atomic writes through mechanisms such as in FTL [219] and through user-programmable SSD [258], however it can also be implemented in the file system itself. F2FS supports multi-block atomic writing, which allows updating multiple file system blocks in a single *ioctl* command [40, 148, 240]. ReconFS [195] provides multi-page atomicity by using a flag in each page to indicate if it is valid. This is achieved by writing a 1 in the head of the last flash page of the metadata updated, and all other pages have a 0 in the head. Therefore, in the case of power failure, when the file system is reconstructed, any metadata pages in between two pages with a 1 in the head (depicting two ending page updates) are valid pages, and in the case when the log has no page with the flag set to 1, the update failed and all pages after the last 1 flag are invalid.

Similarly, Qin et al. [240] showcase the utilization of the OOB area on flash with OCSSD in order to store metadata for ensuring data integrity in the No-Barrier File System (NBFS), which is extending F2FS. The file system uses *Data Node Chain (DN-Chain)* to ensure consistency measurements. DN-Chain is a linked list of all the blocks in an atomic operation, which are stored through a pointer in the OOB space, thus representing a linked list of pointers in an atomic operation. The last block in the DN-Chain points to itself to indicate the end of the linked list and the atomic operation. Furthermore, each block contains a checkpoint version, allowing the recovery after a failure to traverse the DN-Chain and identify if an invalid checkpoint version number is present, which indicates a failed atomic operation. As the initial writing of journal or CoW is done in memory and later flushed to the flash storage, mechanisms such as failure-atomic *msync()* [231], provide atomicity in such operations.

3.11.2. Transactions

An additional approach for ensuring consistency is to use transactions, enforcing an “all-or-nothing“ mechanism that either writes all data or no data at all if there are any failures during the transaction. Transactional support can stem from transactional block devices, such as TxFlash (Transactional Flash) [237] LightTX with embedded transaction support in the flash storage FTL [192, 196], and similar block device implementations that expose transaction support [45, 104, 137]. Additions to file systems can similarly implement transactions for block devices without transaction support. exF2FS [216] is an extension on F2FS that provides it with support for transactions. In order to support transactions, exF2FS implements several features. Firstly, with transactions relying on either all updates being present or no updates at all, the GC policy is adapted such that the GC module cannot reclaim flash pages that contain data from a transaction that is in progress. If such a page is garbage collected without the transaction having finished, it can be recovered and thus violates the all-or-nothing mechanism.

For this, exF2FS implements *shadow garbage collection*, which prohibits the GC module from using pages involved in pending transactions. Secondly, in order to provide large scale transactions with multiple exF2FS maintains *transaction file groups*, which are kernel objects that identify the set of files involved in a transaction. Lastly, it implements *stealing*, which allows dirty pages of pending transactions to be committed to the flash storage, however do not allow it to be garbage collected until the transaction completed. This mechanism is referred to as *delayed invalidation and relocation record*, and the process of allowing dirty pages to be evicted in uncommited transactions is referred to as *stealing* [216].

3.11.3. Flash Failures

Flash failures are another important aspect of file systems on how they manage and recover from these failures. A study into file system reliability showed that 16% of injected faults resulted in an unmountable file system, which furthermore was not fixable by a file system checker [115]. Especially, as the density of flash is increasing, where more layers of NAND is stacked on top of each other, resulting in a decrease in the flash reliability [207]. While flash employs ECC to handle correcting of data, there are errors that are not fixable such as the *uncorrectable bit corruption* [11, 21, 22], which can be caused by flash wear, read disturbance, and other factors [115]. We do not go into detail of flash induced errors, for a detailed evaluation on flash errors recovery mechanisms consult [81, 115]. Jaffer et al. [115] provide several guidelines for file systems to enhance reliability with flash errors, including adding more sanity checks, especially of metadata, and finding better tradeoffs between checksums and the checksum granularity, where a large checksum granularity can result in significant data loss if it has an unrecoverable invalid checksum, and a small checksum adds overheads on the needed space and performance. While file system checkers aim to solve a degree of faults, they are no panacea to fixing corrupted file systems [81] and require better failure handling in the file system itself.

3.11.4. Summary

The importance of failure consistent operations for storage systems has resulted mechanisms, such as atomic writes, to utilize the available OOB area of flash pages for metadata to ensure consistency in the case of failures. Similarly, in order to avoid flash errors, checksum methods have been introduced to alleviate errors as a result of flash failures.

3.12. Discussion

With the multitude of methods for dealing with flash integration, there are several that are of key concern. In particular, GC and mechanisms of dealing with GC have shown to be applicable, and enhance other flash integration challenges. However, there is no panacea for solving all flash integration challenges. Incorporating a particular mechanism causes difficulty of integration with other flash challenges. Especially, depending on the level of integration of flash storage, there is limited possibility of integrating particular mechanisms. Therefore, the closer integration is largely necessary in order to more optimally integrate flash, however coming with the additional flash management complexity. Therefore, a tradeoff between the level of integratin and the required complexity must be made. A higher level of management allows the application to more optimally be designed for the flash. This however comes at the expense of increased complexity. Furthermore, generic file systems working on numerous integration levels, require more generic interface management, limiting their customizability for different integrations.

Clear trends in the storage community are becoming apparent, focusing on eliminating the hiding of flash management idiosyncrasies, and exposing its characteristics to the host. Therefore, allowing the host to integrate and optimize the software for its access patterns with increased knowledge of the underlying storage device characteristics [16, 19, 20, 29, 127, 134]. As stated in the CompSys Manifesto [111], ”the grand challenge in storage systems is to combine heterogeneous storage layers, leveraging their programmability and capabilities to deliver a new class of cost, data, and performance efficiency for all kinds of applications”. Reducing the semantic gap between storage hardware and software must be a driving concern in future storage system design, leveraging the existing hardware capabilities and enhancing integration for more efficient, effective, sustainable, and reliable storage systems.

The recent addition of *Zoned Namespace (ZNS)* SSD [20, 281], standardized in the NVMe 2.0 specification [301], similarly provides host the control over GC management on the storage device, and matches the interface to the underlying flash characteristics. Leveraging such device interfaces allows the file system development to partially take control of on-device operations. In addition to eliminating duplicate work of file system and GC on the device, where the increased coordination provides better performance capabilities, the higher-level control of the file system allows it to apply its data knowledge, such as particular grouping based on file characteristics, without a significant increase in complexity through the zone interface of ZNS SSD. Such integration presents a well-defined interface aimed at eliminating the interface mismatch between flash storage and storage software, leaving a plethora of possibility for storage software to enhance flash integration. Exposing more storage device control to the host system, particularly in the configuration of ZNS, allows not only data placement to be better integrated into storage software, but furthermore allows to fully utilize the parallel units of the storage device. With ZNS devices, the parallelism unit of the storage device is clearly defined [8], and can be leveraged by the storage software.

We imagine interfaces exposing an increasing level of flash hardware characteristics to the host software, to begin appearing, in order to better coordinate and integrate the storage. Similarly, we envision future efforts to aim at further decreasing the semantic gap between storage hardware and software, and leverage a higher degree of coordination across storage software/hardware layers. The gain in popularity of user-space based applications, as we discussed in Section 3.10.3, presents prominent possibility for future development and limiting kernel involvement, which has become an increasing overhead in the software stack. Therefore, we furthermore picture an increase in user-space applications, and particularly file systems.

3.13. Related Work

With the growing adoption of flash based storage systems [54, 172], there has been a plethora of proposed systems to optimize for flash characteristics. We focus on the existing file systems for flash storage, but other aspects such as key-value stores are another popular use case for flash storage.

Flash Optimized File Systems. Egger [69] provide a survey on the file systems for flash storage at the time (2010), comparing the file systems in key features important for flash storage. This includes the feature set of the file system, time complexity of operations such as mount time, and space requirements for memory. However, evaluated file systems are not limited to flash specific file systems. While the survey presents an insightful summary of flash file systems, it was published in 2010, thus limiting the number of available file systems significantly. Similarly, Gal and Toledo [80] present a survey of algorithms and data structures for flash storage, which encompasses flash mappings and flash-specific file systems.

Jaffer [114] provide a comparison of five file systems for flash storage, analyzing the feature set of each and discussing limitations. In particular, the author focuses on the evolution of design trends for flash storage over the past decades, where the earliest file system included in the comparison was presented in 1994. The author additionally includes a discussion on file system optimizations for data management with Streams [134] and OCSSD [19, 235]. While the author presents an insightful analysis into design trends for flash storage, the literature review is limited to only the five discussed file systems and does not differentiate in the file system application domain.

Dubeyko [67] presents SSDFS, a file system designed for SSDs. Albeit not being a literature review of flash file systems, the author presents an extensive comparison of related work that proposes flash file systems. Including a discussion on flash-friendly and flash-oriented file systems, and summarizing the available methods for optimizing flash specific operations and storage management. While the discussion of flash file systems provides a comparison of flash file systems, it is similarly not differentiating between application domain of the file system. Munegowda et al. [212] showcase a study into several file systems on Windows and Linux for SSD and flash devices. Furthermore, the authors present a comparison of features for the varying file systems, including flash support and FTL integration, as well as a high level performance comparison. However, the study is focused on the main adopted file systems, lacking the inclusion of less adopted flash file systems.

Ramasamy and Karantharaj [244] survey the challenges of building file systems for flash storage, including the performance implications, caching techniques, and implications of the FTL. The authors additionally discuss available solutions to the presented design challenges for storage systems on flash memory. Similarly, Di Carlo et al. [59] present a study into the design issues and challenges of flash memory file systems. The authors analyze the inherent implications of the flash storage from the type of cell type used and required wear leveling, error correction, and bad block management. A comparison of several available flash file systems at the time of publication (2011) is presented, with focus on the discussed design challenges.

Flash Optimized Applications. Not focusing of file systems for flash storage, there have been several surveys into flash characteristics. Luo and Carey [5] present a survey into log-structured merge-tree (LSM-tree) design techniques for storage systems. With LSM-trees being a widely adopted and popular choice of database and key-value store design to optimize for flash storage, the presented survey showcases relevant flash storage optimizations for data management. Similarly, Doekemeijer and Trivedi [65] present a study into key-value stores optimized for flash storage, showcasing several techniques that can likewise be applied to file system design for flash storage.

Flash Translation Layer. Chung et al. [44] provide a survey into the various FTL algorithms, discussing the design issues of various algorithms. Flash file system performance will largely depend on the FTL implementation for SSDs, making optimal FTL design an important aspect of optimizing file system performance. A similar survey on FTL algorithms is presented by Kwon et al. [160]. As embedded devices and possible custom integrations require management of flash at the file system level, the concepts for efficient and performant FTL algorithms are applicable to file system level management of flash storage.

3.14. Conclusion

The move from HDD to flash SSD, has been one of the most fundamental shifts in the storage hierarchy. The increased performance of flash SSD over HDD, capable of achieving single several GB/s bandwidth with millions of IOPS, however required adaptations in the software stack, changing the design of file systems and storage software to integrate with these devices. In this literature study we evaluate the current state of file systems for flash storage devices, how these file systems design to align with introduced flash characteristics, and how the integration of flash storage into file system design has affected storage stack design. We evaluate the findings of this literature study to each of the proposed *survey research questions (SRQs)*.

SRQ1: What are the main challenges arising from NAND flash characteristics and its integration into file system design?
The architecture of flash-based storage introduces six key challenges for file system and storage software developers to consider during software design and development. The first key challenge of flash storage is the (1) **asymmetric read and write performance**, for which file systems developers commonly resort to methods and data structures for enhancing write performance. Due to the architecture of flash storage, the lack of in-place updates introduces required (2) **garbage collection**, presenting a significant performance implication for flash storage. Furthermore, the implications of GC extend to introducing (3) **I/O amplification**, which additionally increases the wear of flash cells, requiring effective (4) **wear leveling** methods to employed. In order to leverage the performance capabilities of flash storage, the (5) **flash parallelism** must be exploited with particular methods that are capable of increasing the concurrency on the flash storage. The last key challenge of flash storage arises from the storage software stack into which the devices integrate, where the performance of the storage stack becomes the bottleneck. Therefore, (6) **I/O management** for flash storage is a vital aspect at limiting software overheads, and maximizing the utilization of the flash storage.

SRQ2: How has NAND flash storage influenced the design and development of file system and the storage software stack?
The main challenges of integrating with NAND flash storage resulted in file system design to utilize specific data structures, algorithms, and mechanisms. Log-based data structures are widely adopted for flash-based file system, due to the matching characteristics to the flash storage. In addition to log-based data structures, file system development has focused on several key methods to exploit the parallelism capabilities of flash storage, including clustered allocation, data striping, and increasing the I/O sizes. Similarly, the design of I/O management has propagated out of the file system design, into the I/O scheduler architecture, to optimize I/O activity for fast flash-based storage.

SRQ3: How will NAND flash storage and newly introduced NAND flash-based storage devices and interfaces affect future file system design and development?
Given the increasing rise in adoption for flash storage, and the introduction of new flash-based storage devices and interfaces, future implications of flash storage **(RQ3)** provide a promising ground of better integrating the flash storage with software, leveraging the increased performance capabilities further. Through new interfaces exposing a larger amount of flash characteristics, the host software gets an increasing level of possibility to design application specific flash management, integrating the application design with flash management. Future storage software developments are likely to continue integrating the closer integration of flash storage into host storage software design, optimizing the flash utilizing for full leveraging of flash performance. We envision the semantic gap between storage hardware and software to slowly decrease over time, allowing to build more performant, efficient, reliable, and responsible storage systems. Such efforts align with the grand challenges of future storage [111], particularly with increasing demand of systems due to the digitalization of the world, and the push towards a more sustainable future.

4

zns-tools - Collecting, Analyzing, and Visualizing File System Operational Data with ZNS

The complexity of file systems and storage systems makes understanding file system operations and on-device data placement a challenging task. In particular the recent standardization of ZNS currently has little tools for understanding and representing file system operation and utilization of these new devices. In this chapter, we present `zns-tools`, a set of tools for identifying and understanding operational aspects of F2FS data placement and file mappings on ZNS, and a tracing framework for collecting and visualizing of ZNS activity and device utilization. We present a total of five different tools for these purposes. A comparison of the feature set of each tool is provided in Table 4.1. We make all source code for the tools publicly available at `https://github.com/nicktehrany/zns-tools`. Furthermore, we detail the implementation of each tool and component in Section 4.6.

4.1. zns.fiemap - Mapping F2FS Files to ZNS Zones

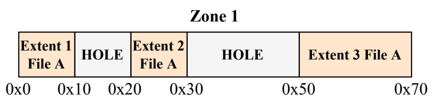
With file system complexity of ZNS integration, it becomes important to identify the on-device data placement of files. In particular LFS where contention on log resources of a single zone results in file fragmentation, `zns-tools` provides a tool (`zns.fiemap` - ZNS filemapping) that extracts the file mapping information from the file system, maps this data to the ZNS device zones, and provides mapping information and fragmentation statistics. File fragmentation particularly affects read performance, decreasing the sequential read performance to that of random reads [35], and reduces data grouping, resulting in decreased GC performance. With fragmentation, if a file is deleted or its data is updated, the individual file fragments become invalid, causing remaining valid data around the fragments to be moved during GC.

`zns.fiemap` relies on the `ioctl()` call with the `FIEMAP` flag, which informs the file system to return extents of the file. An extent is a physically contiguous region of the file data on the storage device. Note, this is not a mandatory feature of file systems to implement with POSIX compliance, however all currently ZNS supported file systems provide this option (F2FS and Btrfs). With this system call all extents of a file are collected by looping over the logical address range of the file and retrieving the physical extent mappings. Next, each extent is mapped to the zone it is contained in on the ZNS device. For this the extents are firstly sorted, such that the per-zone mappings of which extents are contained in that particular zone can be generated. Once sorted, the contained extents in a zone are identified by finding the first PBA of an extent that is within the zone's address range. Since the zone capacity can be less than the zone size, extents do not span across the zone boundary, as the block address after the zone capacity is unmapped, and hence not addressable. Therefore, the last extent within a particular zone can be identified as having an ending block address below or equal to the zone capacity.

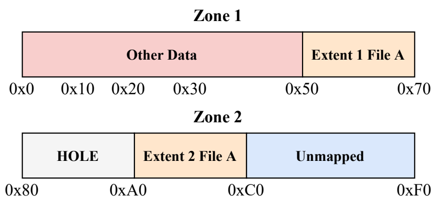
In order to provide fragmentation information the tool additionally gathers statistics on the holes between extents. For this we define three definitions of how holes can be present between extents, depicted in Figure 4.1. (1) The ending LBA of an extent is not equal to the starting LBA of the next extent in this zone, as illustrated in Figure 4.1a. This implies that between the two extents other data resides. The type of data is

Tool	Domain	Features
zns.fiemap	F2FS with ZNS	• File-to-Zone Mapping • Intra-File Mapping Holes • Intra-File Fragmentation Statistics
zns.imap	F2FS with ZNS	• Inode-to-Zone Mapping • Show Inode Contents • Show Superblock Contents • Show Checkpoint Contents
zns.segmap	F2FS with ZNS	• File/Directory-to-Segment Mapping • Segment-to-Zone Mapping • Segment Classifications Statistics
zns.fpbench	F2FS with ZNS	• Write Workload Configuration • msF2FS Stream Hints Integration • Segment Classifications Statistics
zns.trace	ZNS	• Per Zone Write, Append, Read Statistics Heat Maps • Per Zone Reset Statistics Heat Maps

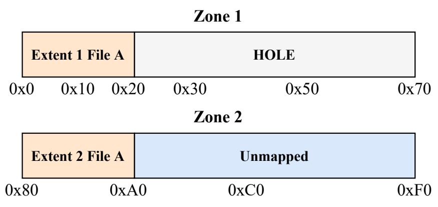
Table 4.1: Comparison of the features and application domain of all tools present in the zns-tools framework. The application domain depicts where the particular tool can be used, where zns.trace is a generic tool to trace any ZNS activity, independent of file system or application.



(a)



(b)



(c)

Figure 4.1: Possible holes that `zns.fiemap` shows, with (a) being a hole where either data from different files or invalid data is present between extents of the same file, (b) a zone ends with an extent of a file however the next zone does not start with data of the same file, and (c) an extent of a file not continuing up to the zone capacity if the next zone starts with an extent of the same file.

not considered, as it can be data from other files or old blocks of the current file that have been updated. The only requirement is that the extents are broken up and are non-contiguous. (2) An extent of a file continues up to the end of the zone, the zone capacity, however in the latter zone the next extent does not start at the beginning LBA of the zone, depicted in Figure 4.1b. This implies that the first zone was fully written with the file's data and the latter zone is written with other data prior to the next data of the file. Note that with F2FS there are multiple data logs, hence zones of a single temperature classification may be non-contiguous, such that for example hot data is placed into zone 1, while warm and cold data occupy zones 2 and 3 respectively,

```
user@stosys:~/src/zns-tools/src$ sudo ./zns.fiemap -f /mnt/f2fs/db0/LOG -s
====================================================================
                        EXTENT MAPPINGS
====================================================================

**** ZONE 13 ****
LBAS: 0x3000000  LBAE: 0x321a800  CAP: 0x21a800  WP: 0x3400000  SIZE: 0x400000  STATE: 0xe0  MASK: 0xffc00000

EXTID: 33    PBAS: 0x30e2938   PBAE: 0x30e2980   SIZE: 0x48
--- HOLE:    PBAS: 0x30e2980   PBAE: 0x30ebe00   SIZE: 0x9480
EXTID: 34    PBAS: 0x30ebe00   PBAE: 0x30ebe10   SIZE: 0x10
--- HOLE:    PBAS: 0x30ebe10   PBAE: 0x3181e28   SIZE: 0x96018
EXTID: 35    PBAS: 0x3181e28   PBAE: 0x3181e30   SIZE: 0x8
--- HOLE:    PBAS: 0x3181e30   PBAE: 0x31d1d80   SIZE: 0x4ff50
EXTID: 36    PBAS: 0x31d1d80   PBAE: 0x31d1da8   SIZE: 0x28
--- HOLE:    PBAS: 0x31d1da8   PBAE: 0x321a800   SIZE: 0x48a58

**** ZONE 16 ****
LBAS: 0x3c00000  LBAE: 0x3e1a800  CAP: 0x21a800  WP: 0x4000000  SIZE: 0x400000  STATE: 0xe0  MASK: 0xffc00000

--- HOLE:    PBAS: 0x3c00000   PBAE: 0x3c26890   SIZE: 0x26890
EXTID: 70    PBAS: 0x3c26890   PBAE: 0x3c268a8   SIZE: 0x18
.
====================================================================
                        STATS SUMMARY
====================================================================

NOE: 76    TES: 0xef8       AES: 0x31        EAES: 49.766234   NOZ: 17
NOH: 92    THS: 0x2230d60   AHS: 0x5f23b     EAHS: 389691.478261
```

Figure 4.2: Example `zns-fiemap` output, depicting file mappings of a RocksDB LOG file, containing zone mappings, hole information, collected statistics. For illustrative purposes, we only show a subset of the mappings.

and node data occupies the next 3 streams. Therefore, the next hot data zone is mapped into zone 7.

Consequently, the number of zones that are between extents is ignored in the hole statistics, in order to avoid exaggerating the fragmentation distance, but rather utilize the total number of LBAs in the defined holes. If an extent does not end at the end of a zone, and does not start at the beginning of a latter zone, however has an additional extent in that latter zone, which is counted as 2 separate holes. One hole from the extent ending address to the zone capacity, and a second hole from the beginning address of the latter zone to the beginning address of the next extent. (3) The inverse of the prior scenario, where in a latter zone an extent starts at the starting LBA of the zone, while in an earlier zone there is an extent that does not end at the zone capacity, illustrated in Figure 4.1c. This implies that in the earlier zone file data is followed by other data and the new zone contains file data again. With the depicted hole definitions statistics are collected on the total number of holes, depicting the intra-file fragmentation, the accumulated extent and hole sizes, representing the degree of fragmentation over the occupied address range, and the average sizes of extents and holes.

Figure 4.2 shows an example output of the file mappings for the `LOG` file in RocksDB after a synthetic workload. Collected statistics showcase the number of extents (NOE), total extent size (TES, in 512B sectors), average extent size (AES, in 512B sectors), exact average extent size (EAES, in 512B sectors), number of zones (NOZ), number of holes (NOH), total hole size (THS, in 512B sectors), average hole size (AHS, in 512B sectors), and the exact average hole size (EAHS, in 512B sectors). The depicted example shows the mapping of the `LOG` file, generated by RocksDB to maintain database information and statistics, which is mapped to a total of 76 distinct extents (depicted with the NOE of the statistics). We can identify the mapping of several of its extents being in zone 13 of the device, where extents are commonly small (several KiB), and holes in between the extents are significantly larger, and similar in size. Knowing that the `LOG` file is only appended to, and analyzing its resulting mapping information, we can gather the write pattern of the `LOG` file being periodic small writes, as the average extents are small in size, and contain frequent holes.

4.2. zns.imap - Locating and Analyzing F2FS inodes in ZNS Zones

Focusing on file system metadata, `zns.imap` (ZNS inode map) locates and maps inodes of a file and prints the full inode. It firstly reads information directly from the storage device, starting with the superblock, in order to retrieve the offset of the NAT, followed by reading the NAT and finding the physical location of the inode for the file. The tool then reads the ZNS device at this location and provides all information in the inode to the user, including the zone mapping of the inode and inode contents. During the process of locating the inode,

```
user@stosys:~/src/zns-tools$ sudo ./src/zns.imap -f /mnt/f2fs/LOG -l 1 -s -c
=================================================================
                        SUPERBLOCK
=================================================================
Note: Sizes and Addresses are in 4KiB units (F2FS Block Size)
magic:                  0xf2f52010
major_version:          1
minor_version:          15
log_sectorsize:         9
log_sectors_per_block:  3
log_blocksize:          12
log_blocks_per_seg:     9
segs_per_sec:           1024
secs_per_zone:          1
.
=================================================================
                        CHECKPOINT
=================================================================
checkpoint_ver:                 747629827
user_block_count:               17883136
valid_block_count:              550951
rsvd_segment_count:             15092
overprov_segment_count:         18972
free_segment_count:             52921
.
=================================================================
                        INODE
=================================================================

File /mnt/f2fs/LOG with inode 4 is located in zone 2

============ ZONE 2 ============
LBAS: 0x400000  LBAE: 0x61a800  CAP: 0x21a800  WP: 0x405730  SIZE: 0x400000  STATE: 0x20  MASK: 0xffc00000

***** INODE:  PBAS: 0x4000f8    PBAE: 0x400101    SIZE: 0x9         FILE: /mnt/f2fs/LOG

>>>>> NODE FOOTER:
nid:                    4
ino:                    4
flag:                   3
next_blkaddr:           1572896
.
```

Figure 4.3: Example `zns.imap` output, depicting the superblock and checkpoint data, alongside the inode mapping and inode data of the file. For illustrative purposes, we only show a subset of the information.

the superblock and checkpoint must be read, and therefore furthermore enable to print all information in the superblock and checkpoint, which is the latest version of the chckpoint. This is particularly beneficial for understanding F2FS configuration and information in the superblock. Understanding inodes allows identifying the inode flags set during its lifetime and understand inode-to-ZNS mappings.

Given that this device locates the metadata of an inode, it can furthermore be extended to support ZNS compliant file systems by adding the retrieval of the inode and the inode structure of other file systems. Figure 4.3 illustrates the collected information shown by this tool on the superblock, checkpoint, and location and information of a particular inode. Important information from the superblock shows the number of segments per section being 1,024, aligning with the zone size of 2GiB containing exactly 1,024 2MiB sized segments. Furthermore, the checkpoint allows understanding the reserved segments by F2FS, and information on current F2FS utilization. With the inode mapping of the RocksDB `LOG` file, we can identify it being placed into zone 4.

4.3. zns.segmap - Mapping F2FS Files to Segment Types in ZNS Zones

Similar to `zns.fiemap`, this tool (`zns.segmap` - ZNS segment map) collects extents, representing physically contiguous data, and maps these to ZNS zones. Additionally, it allows mapping numerous files or a directory and all contained files, as opposed to `zns.fiemap` mapping only a single file. Furthermore, it maps the extents to F2FS segments (2MiB allocation unit). With the segment mapping of extents, `zns.segmap` provides the possibility to retrieve segment information from `procfs`, a special file system which provides process and system information, to enhance provided statistics. `Procfs` is a configurable option during Linux kernel building, it is therefore provided as an additional flag to enable retrieving `procfs` information, if it is available. With the segment mapping and segment information extents are mapped to temperature classification of the segment and collect according statistics. In particular such information becomes increasingly beneficial to identify the number of segments of a file that map to particular temperatures (hot/warm/cold data), and how this classification changes over time through GC moving blocks and reclassifying file temperatures, as GC reclassifies any moved data as cold data.

The tool works by first collecting all extents of the files that are being mapped, followed by sorting the extents. Extents must be sorted in order to map these to the ZNS zones because there may be multiple files that can have interleaved extent mappings. During sorting the segment number each extent maps to is identified, and if enabled the segment information from `procfs`is collected. This information includes the temperature classification of the segment and the number of valid blocks the segment contains. After sorting extents, statistics for each file are collected, in addition to global statistics for the entire directory that is being mapped. The collected statistics contain for each of the files a counter on the number of segments of each temperature type that hold at least one block of data for this file, the number of extents the file has, the number of segments containing data of this file, and the number of zones that are occupied by the extents of this file. The global statistics collected are the exact same as for individual files, therefore simply adding all individual file statistics together.

Lastly the tool presents information on the segments, their zone mappings to the ZNS device, the contained extents with file information, and a summary of all the collected statistics. Figure 4.4 shows a snippet of an example output that maps all files in a RocksDB directory to the F2FS segments and ZNS zones, alongside collected classification statistics. From the example output we can gather firstly, the mapping of several files to the ZNS zones, where for instance the `LOG` file is mapped to 7 extents all residing in the same hot segment, as is seen in the statistics, and one of the extents being mapped to the particular addresses of zone 14 in segment number 13,478. Note, for brevity we do not show the entire mapping of the segment, however all remaining extents of the `LOG` file follow in the mappings of this segment. Secondly, we can identify the mapping of the `db0/000042.sst` file being mapped to 38 extents, of which one resides in segment 13,468, following the `LOG` file. Lastly, with this information allowing detailed understanding of file mappings in F2FS to ZNS we can identify utilization information of the collected segments. For instance, the information of the mapped segment 13,468 contains 32 valid blocks, which is equivalent to $32 * 512\text{B}/1024 = 16\text{KiB}$ valid data in the 2MiB segment (note the LBAF is configured to 512B). With such we can identify the write characteristics to such a segment having overwritten a majority of its data.

4.4. zns.fpbench - F2FS with ZNS File Placement Benchmark

With the different F2FS temperature classifications, this tools (`zns.fpbench` - ZNS file placement benchmark) aims at benchmarking the conformity of F2FS with the host application provided lifetime classification. It allows configuring synthetic micro-workloads to write to the file under specific lifetime hints, with customizable concurrency under several files with the same hint. The tool issues write requests with randomly generated data, based on the provided workload configuration, and collects statistics on the resulting mappings of the file data. Similar to `zns.segmap` this is done by collecting extents, mapping these to segments, collecting segment information from procfs, and gathering statistics of the segment classifications. In addition to benchmarking the file system, this tools was particularly built for the development of msF2FS, as it provides easily configurable workloads to test data classification and integrates msF2FS stream management with AMFS by allowing the workload configuration to specify the stream mapping for a file, which is passed to msF2FS using `fcntl()` with the stream bitmap.

4.5. zns.trace - Tracing ZNS I/O Activity

In order to understand ZNS utilization of F2FS, we design `zns.trace`, a tracing framework using *extended Berkeley Packet Filter (eBPF)* to collect statistics on I/O request and zone utilization. eBPF is part of the Linux kernel, allowing to run programs inside the kernel itself by adding custom code to run during kernel events. This is commonly used in order to trace and benchmark kernel activity. To trace ZNS activity, eBPF is used to insert *probes*, providing hooks of kernel or user-space functions being called which trigger the provided code to be run, in order to trace kernel activity. The inserted probes are triggered during I/O submission and completion, at which point information from the I/O parameters are collected. This information includes the LBA destination, which is converted to its respective zone number, type and size of the requests, and zone management statistics. The zone management statistics include counting the zone reset commands being issued and tracing the duration of these calls. This is done by storing the system time during the I/O sub-

```
user@stosys:~/src/zns-tools$ sudo ./src/zns.segmap -d /mnt/f2fs/ -p -i -s 7 -e 9
====================================================================
                        SEGMENT MAPPINGS
====================================================================

============ ZONE 14 ============
LBAS: 0x3400000  LBAE: 0x361a800  CAP: 0x21a800  WP: 0x35f79e8  SIZE: 0x400000  STATE: 0x20  MASK: 0xffc00000

-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
SEGMENT: 13342  PBAS: 0x341e000   PBAE: 0x341f000   SIZE: 0x1000
+++++ TYPE: CURSEG_HOT_DATA  VALID BLOCKS:  80
-------------------------------------------------------------------------------------------------------------
***** EXTENT:   PBAS: 0x341e800   PBAE: 0x341e848   SIZE: 0x48  FILE: /mnt/f2fs//db0/LOG         EXTID:   2/7
***** EXTENT:   PBAS: 0x341e850   PBAE: 0x341e858   SIZE: 0x8   FILE: /mnt/f2fs//db0/000038.sst  EXTID:   3/3

-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
SEGMENT: 13468  PBAS: 0x349c000   PBAE: 0x349d000   SIZE: 0x1000
+++++ TYPE: CURSEG_HOT_DATA  VALID BLOCKS:  32
-------------------------------------------------------------------------------------------------------------
***** EXTENT:  PBAS: 0x349c868   PBAE: 0x349c870    SIZE: 0x8   FILE: /mnt/f2fs//db0/000042.sst  EXTID: 38/38
***** EXTENT:  PBAS: 0x349c870   PBAE: 0x349c880    SIZE: 0x10  FILE: /mnt/f2fs//db0/LOG         EXTID:   3/7
***** EXTENT:  PBAS: 0x349c888   PBAE: 0x349c890    SIZE: 0x8   FILE: /mnt/f2fs//db0/000043.sst  EXTID:   3/3
.
====================================================================
                        SEGMENT STATS
====================================================================
-----------------------------------------------------------------------------------------------------------
Dir/File Name   | # Extents | # Segments | # Occupying Zones | Cold Segments | Warm Segments | Hot Segments
-----------------------------------------------------------------------------------------------------------
/mnt/f2fs/      | 179       | 2621       | 6                  | 2002          | 532           | 87
-----------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------------
db0/LOG         | 7         | 1          | 1                  | 0             | 0             | 1
db0/000042.sst  | 38        | 146        | 3                  | 145           | 0             | 1
db0/000044.sst  | 5         | 113        | 3                  | 112           | 0             | 1
db0/000047.sst  | 5         | 113        | 3                  | 112           | 0             | 1
.
```

Figure 4.4: Example `zns.segmap` output, depicting statistics on the file placement in F2FS with RocksDB generated files.

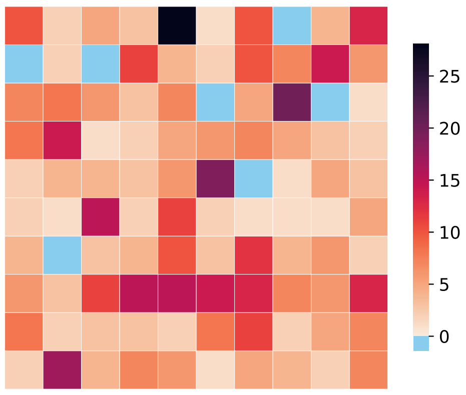


Figure 4.5: Example `zns.trace` visualization of ZNS zone resets issued by F2FS during a RocksDB benchmarking workload. Blue squares indicate zones with 0 reset commands issued.

mission and during the I/O completion calculating the difference of the stored time and the current system time, representing the duration of the zone reset request. In order to comprehensible present all gathered information the tool generates a heat map, depicting visuals for all collected statistics on each individual ZNS zone. The generated heat maps depict the following aspects for each zone:

- Average read I/O size
- Total read I/O (in 512B sectors)
- Total read I/O requests
- Average write I/O size
- Total write I/O (in 512B sectors)
- Total write I/O requests
- Total zone reset commands issued
- Average zone reset latency (over all reset commands issued in that particular zone)

Since this tool traces I/O submissions, it is generically applicable to any application utilizing ZNS devices with the Linux kernel, only requiring eBPF support enabled in the kernel build. Figure 4.5 shows an example heat map of the total number of reset commands issued by F2FS on a ZNS device during a synthetic workload of RocksDB.

4.6. Implementation

In this section, we detail the implementation of each of the tools and components. As several of the presented tools rely on similar functionality, such as retrieving file mapping information, we develop a library, `libzns-tools`, for all common operations against which the tools are linked. Similarly, retrieving F2FS information, such as the ZNS device containing the data, is identical for all tools requiring the information, and is therefore coded into a `libf2fs` library, against which the tools are linked. Table 4.2 provides a summary of the required lines of code for each of the libraries and tools.

4.6.1. libzns-tools

This library provides the foundation for the additional tools and components in this framework, as Table 4.3 illustrates key components of its API for the tools to use. Particularly, a control `struct`, which stores all configuration information, including the F2FS information up to the configuration from the parameters of each of the tools. Therefore, this `struct` can be passed around to all tools and libraries to interact with, and record their necessary information. The functions provided within this library set up basic information that is required for the other tools to function, which includes several features. Firstly, it retrieves information on particular block devices being used. The devices being used are identified by `libf2fs`, using the device name

Library/Tool	Programming Languages	Lines of Code
libzns-tools	C	901
libf2fs	C	965
zns.fiemap	C	262
zns.imap	C	140
zns.segmap	C	800
zns.fpbench	C	569
zns.trace	bpf, python, and bash	497
Total	-	4,134

Table 4.2: Lines of code required for each of the libraries and tools developed.

Function	Return Value	Description
get_extents()	`struct extent_map`	Retrieve all file extents with `FIEMAP` `ioctl()` for the open file descriptor, and return a map of all extent mappings.
sort_extents()	void	Sort the provided extents in the extent map based on their PBAs.
print_zone_info()	void	Given a zone identifier, this function retrieves all information of the particular zone and reports a summary.
set_super_block_info()	void	Read the F2FS superblock from the block device, and construct necessary metadata information and mappings.

Table 4.3: Subset of the `libzns-tools` API, illustrating the functions utilized by the developed tools in this framework.

(e.g., /dev/nvme0n2), it collects information on the device, identifying if the device is zoned or conventional, the sector size of the device (LBAF configuration), and the total size of the device. At the startup of each tool, the control `struct` is initialized with these values, and derives further values, such as the bit shift number that is used during LBA calculations.

With general configuration information collected, this library provides a vital additional functionality, which the majority of tools rely on, that collects all extents for a file. The process of collecting extents requires to individually issue a system call to map a particular extent, within a range, to its physical address space. Therefore, retrieving of extents becomes nontrivial, as the logical address range must be traversed up to the last extent in the logical address space, which are then mapped into physically present extents. The respective file system implements the functionality for retrieving and mapping of extents, the containing extents are collected by issuing `ioctl()` calls with the `FS_IOC_FIEMAP` flag, passing a control `struct` to the function call. The control contains information for the file system on what starting LBA to begin mapping extents from, how many extents to retrieve, and ending LBA to stop retrieving extents at. All this information is identified prior to the function call by using `fstat()` on the file, providing statistics on the file that include the address range information. To gather the extent information, an `ioctl()` call is issued for each extent individually, setting the number of extents to retrieve in the control to one, and iteratively after each returned extent, increase the starting address to the ending address of the prior returned extent, such that the next extent is retrieved. This process is repeated until the last extent of the file, which has a `FIEMAP_EXTENT_LAST` flag set, indicating the end of the file extent mappings is reached. During each extent retrieval, the required fields of the returned extents are copied to a `struct` to hold all extent mapping information. With all extents for a file mapped, the library returns the `struct` containing all mappings to the caller, and frees any no longer needed memory.

Additional functionality provided by this library includes the collection and printing of current information on a particular zone. As the majority of tools require this during the report printing phase, providing the final output for each tool, this functionality is provided in this library. Collection of statistics is additionally common among several of the tools, and is therefore provided by the library, with collected statistics being written into the global control `struct`. In order to provide statistics on the zone mappings, which tools maintain on a per-zone basis that is indexed by the zone number, all addresses must be converted to respective

```c
uint32_t get_zone_number(uint64_t zone_size, uint64_t lba) {
    uint64_t zone_mask = 0;
    uint64_t slba = 0; /* starting LBA of the zone */

    zone_mask = ~(zone_size - 1);
    slba = (lba & zone_mask);

    return slba / ctrl.znsdev.zone_size;
}
```

Figure 4.6: Source code with bit operations to retrieve the respective zone number for a LBA.

zone numbers. With the benefit of zone sizes being a power of 2 value, zone conversions are achieved with bit operations. Figure 4.6 depicts a function to calculate the respective zone number (index starting at 0) from a LBA, using bit operations. It relies on firstly calculating the zone mask by using bitwise NOT on the zone size minus 1. The resulting value represents the mask, which once applied to the LBA in the bitwise AND operation, clears the respective bits of the LBA, resulting in a value that is a multiple of the zone size, since the mask of the zone size is used. Using this value and the given size of the zone, the exact ZSLBA is calculated. Lastly, given that this library provides the extents, it furthermore provides the functionality to sort the extents, which is required by several tools to map extents to zones if extents from different files can be interleaved in the same zone. Sorting is achieved by applying bubble sort with worst case time complexity $\mathcal{O}(n^2)$.

4.6.2. libf2fs

This library provides all functionality that is required to retrieve any F2FS related information. It firstly must read the superblock from the storage device, and store required information in the global control `struct`. This information is needed throughout the tools for any additional operations. As F2FS places the superblock at a particular offset within the storage device (at 1,024 bytes), a read request is issued at that particular address and all information is stored in a `struct`, based on the kernel definition of the F2FS superblock. Given that the first step of all tools is to collect statistics on a file and initialize the control `struct`, the statistics include the identifier (minor and major number) of the device that is storing the data. However, with the current F2FS implementation the returned device for any file on the ZNS address space is the conventional device that the metadata is on. Even if the file data is physically present on the ZNS device, the conventional device is returned. However, knowing the conventional device a read to the superblock is issued, followed by analyzing of the superblock information to identify the ZNS device, and retrieving its information. All this information is inserted into the global control `struct` for the other tools to access. With the superblock information, the offset of the checkpoint is identified, in addition to the offset of the NAT, which is also stored into respective data `structs`. This information is required by the `zns.imap` tool, which traverses the NAT to locate the address of particular inodes. Therefore, this library additionally provides the functionality to read and traverse the NAT entries within a block, identifying if a particular inode number is present in the block, and identifies the NAT entry for the inode. Additional functionality provided by this library includes reading the `procfs` entries of F2FS and providing particular information, such as the lifetime classification of a particular segment.

4.6.3. zns.fiemap

Using the library functions, this tools combines the functionality of the libraries to firstly establish the global control `struct`, followed by retrieving of the file extents. Given that this tool only maps a single file, the extents do not need to be sorted, as they are traversed in logically increasing order. During iteration over the mapped extents, the starting and ending addresses of consecutive extents are analyzed, in order to identify if holes are present between the extent. If present, the distance between the addresses is calculated, identifying the size of the hole, and store this information for future statistics collection. This process is done on-the-fly during the printing of the report, which provides the mappings of files to extents and zones, as well as depicting the hole information.

4.6.4. zns.imap

Given that the majority of functionality for locating and mapping inodes is implemented by the libraries, this tool requires few lines of code to combine the remaining functionality that is required. After initializing all control functionality, the NAT is traversed for each block address of an inode. Note, `libf2fs` implements the function to retrieve a NAT entry for an inode number, however this may be data, while this tools needs to locate the inode. Therefore, all NAT entries that match the inode number for the file to be mapped are checked. These are traversed individually, with the control identifying the past NAT entry checked, such that the same entry is not iterated over again, but continue checking the next entry. In order to identify if the correct NAT entry is selected, the block for the returned NAT entry is read and checked if the node footer in the block identifies it as being a node. If it is a node, the inode is found, and its information is reported. Otherwise, the next NAT entry is checked. This functionality is implemented in `libf2fs`, where `zns.imap` only checks if a valid NAT entry has been found, and otherwise calls the library again to find the next NAT entry.

4.6.5. zns.segmap

This tool similarly relies on the existing functionality of the libraries, setting up the F2FS information and retrieving the extents. A difference to existing tools is that this tool provides the option to map multiple files and directories. Therefore, it relies on a recursive approach to extract the mapped extents of all the files. With extents for all files collected, the extents are sorted, which functionality is provided by the libraries. With sorted extents, each is mapped to the respective segments in F2FS by looping over all extents. When mapping the extents to F2FS segments, information is extracted for the segment from the F2FS statistics in procfs. During these operations the collected segment information and zone mappings are printed, to the user on-the-fly. Concurrently, per-file/directory statistics are collected, identifying the number of extents the directory/file maps to, the number of F2FS segments these extents are contained in, the total number of zones that are occupied by their respective extents, and the statistics on the segment classification type of the occupied F2FS segments.

4.6.6. zns.fpbench

With a focus on benchmarking the data placement of F2FS, this tool issues independent write requests to the file system, based on the provided configuration. The configuration allows depicting the concurrently run processes to issue independent I/O requests to different files, with a distinct file for each process, and characteristics of the I/O. The I/O configuration defines a block size of how large each I/O is, if `fscyn()` is to be called after a specific number of I/O requests, and integrate with msF2FS to map files to particular streams. This tool is similar to existing benchmarking frameworks, such as fio, however provides a smaller and easier to configure code base. The particular aim of this tool is to benchmark the data placement decisions of F2FS, with custom workloads, and utilizing the additional tools to identify the exact placement. Therefore, serving in the problem definition, by identifying the lack of concurrency in F2FS for files under the same write hint. Due to its small code footprint, this tool was readily integrated into the development of msF2FS by allowing to generate minimal write workloads to debug data placement of the file system. Using the developed libraries, the tool only requires to manage the workload and writing of files, where the libraries handle additional information retrieval, including segment and data placement summaries.

4.6.7. zns.trace

As this tracing tool relies on bpftrace, it is disjoint from the other tools. At the core, it relies on eBPF to insert two kernel probes, one for the setup of a NVMe command with `nvme_setup_cmd`, and a second to trace NVMe command completion with `nvme_complete_rq`. The structures of the function arguments are complex and contain a plethora of information. Several particular fields are of interest, indicating the target device, target LBA, opcode that indicates the type of operation, the command flags to retrieve the command type, and further I/O information. Both must be checked, the command type and the operation code, because in the case where the device is attached in passthrough mode (e.g., with *vfio-pci* passthrough from host to a VM), the command flags are set to `REQ_OP_DRV_OUT` for issued zone resets, indicating the host system driver (*vfio-pci*, outside the VM) handles the request. In such case, the opcode of the NVMe command is checked, which indicates `nvme_cmd_zone_mgmt_send`, based on which an additional structure in the NVMe command, `nvme_cmd->zms.zsa` indicating the zone management action, is checked. From this structure the type of operation is identified. In all other cases, the command flag provides information to identify the type of command (read/write/zone append).

During the setup function the respective I/O information is extracted and stored in different data maps. Data maps are indexed by the ZSLBA, which is retrieved with identical bit operation as utilized in the tool libraries (Figure 4.6). To extract the size of the request, the `data_len` field of the request is right bit shifted by the sector size of the device (e.g., bit shift of 9 for a sector size of 512, $2^9 = 512$), giving the number of

LBAs of the request, rather than the exact bytes. With the data maps being indexed by the ZSLBA, counters are maintained for each zone on the number of requests of each type issued to each zone. To the benefit of the Linux kernel relying on power of 2 values for addresses and device sizes, bit operations are used for all address calculations in the kernel, avoiding much complexity of division/multiplication operations. In order to retrieve the zone reset latency, during the setup the current system in nanoseconds is recorded, along with the command identifier that is provided in the command tag. The command identifier must be store with the respective time stamp in order to be able to identify the correct zone reset completion, as concurrent resets may be issued. Therefore, on completion of the request the command identifier is checked and the difference of the current time stamp and the recorded time stamp of command issuing is calculated and stored in a data map.

4.7. Summary

With the complexity of file systems and data allocation, and the lack of currently available tools for collecting data and visualizing these operation aspects, we utilize the opportunities of the new ZNS interface to develop a set of tools for understanding operation aspects of F2FS and ZNS. We make the tools publicly available at `https://github.com/nicktehrany/zns-tools`, which is actively continuing development for additional features and extensions to further file systems on ZNS.

5 Design and Implementation of msF2FS: A ZNS optimized F2FS Design

While F2FS is the de facto standard file system for flash storage, its integration of newly standardized ZNS devices fails to leverage much of the ZNS benefits and capabilities. The number of concurrently active zones in F2FS is limited to six, whereas current commercially available ZNS devices (e.g., the ones used during this thesis work) support a larger number of active zones [20, 94]. Leaving this potential utilization of a larger number of active resources fails to leverage the full capabilities of ZNS in terms of performance and optimality in data placement. Furthermore, the standardization of ZNS introduces the coordination of device and host software in storage device management by pushing GC responsibility to the host software. While this decreases the semantic gap of storage, a semantic gap between the application on file system remains, due to the lack of presenting new management possibility from the file system interface to the application. In this chapter we make the following contributions:

1. We detail changes made in the Linux kernel and F2FS to integrate ZNS into F2FS (Section 5.2).
2. We introduce msF2FS, an enhanced F2FS for ZNS specific integration allowing to fully leverage the parallelism capabilities of ZNS (Section 5.3).
3. We detail the design of msF2FS, and describe the implementation of its various features and data placement policies(Sections 5.4, 5.4.1 and 5.6).
4. We describe the integration of **application-guided data placement**, allowing applications to pass hints to msF2FS to coordinate the data placement decisions between msF2FS and applications (Section 5.4.2).
5. We depict the design of a proactive mechanism for managing concurrently active zones of ZNS in msF2FS with buffered I/O (Section 5.5).

5.1. Design Constraints

With msF2FS being built on the basis of F2FS, there are several constraints that apply to its design. (1) msF2FS is an in-kernel file system, running entirely in the Linux kernel. (2) By building on top of F2FS, it inherently utilizes the design of F2FS, with data structures, and operations. To support the stream integration and application-guided data placement, msF2FS makes additions to the operations and data structures of F2FS. With the utilization of F2FS as its base, msF2FS similarly relies on the standard Linux calls to interact with the ZNS device and additional parts of the Linux kernel. (3) With msF2FS being a prototype file system, aimed at evaluating the implications of enhanced ZNS integration, it lacks several features to be able to be used in production. Of particular importance are the recovery mechanisms, including the checkpoint, which msF2FS does not utilize for stream information. Therefore, in the case of a system failure, the stream information cannot be recovered. Furthermore, the application-guided data placement hints are relying on the `fcntl()` call to be set, storing application hints only in the VFS inode, and not on the persisted on-device inode, which is written to the ZNS device. (4) For consistency reasons, msF2FS only provides streams for data logs, and not

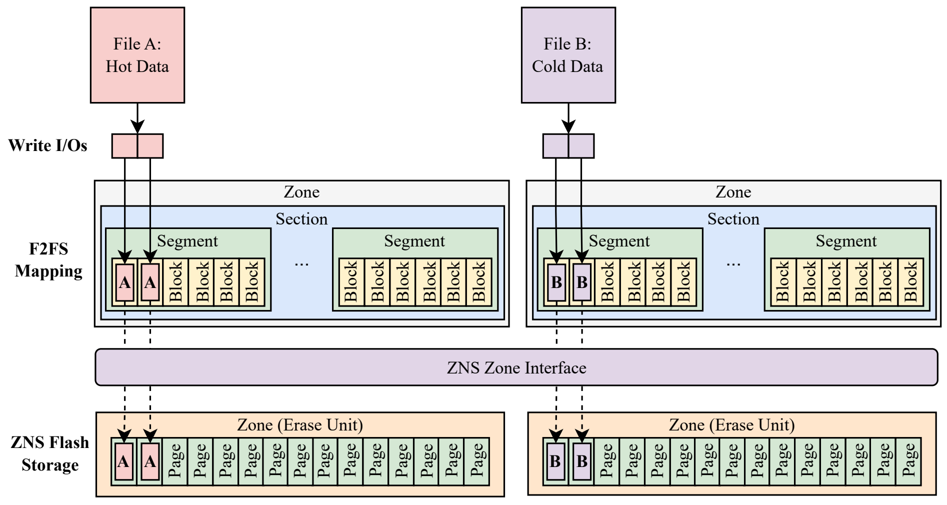


Figure 5.1: F2FS allocation aligns the F2FS zone to the ZNS zone, with a larger number of sections, ensuring adequate data separation.

for node logs. This is partially a design choice due to the increased fragmentation for file data, as opposed to node data. However, to build a functioning prototype of msF2FS we additionally disable node streams due to multiple node logs making the maintaining of consistency increasingly complex. For instance, with multiple node logs different versions of the same inode may reside in different logs, requiring versioning and other methods to locate the valid inode. We identify several of these limitations as possible future work (discussed in Section 8.2), furthermore warranting an extended evaluation on the additional features.

5.2. Existing ZNS Support in F2FS

We firstly summarize the work F2FS developers have done to integrate ZNS devices [20]. With F2FS being a LFS, the sequential write constraint is already fulfilled to an extent, minimizing the necessitated changes to the file system. The changes that are needed revolve around two key aspects.

5.2.1. Aligning F2FS Allocation to ZNS

Given that ZNS introduces a zone capacity, making several LBAs not usable, F2FS must similarly align its allocation to consider the zone capacity. F2FS segments are 2MiB in size, whereas the zone size remains a power of two value, the zone capacity may be a number that is less than the zone size. Therefore, the file system segments remain mapped up to the zone size, thus filling the entire address space, however any segment after the zone capacity is marked as unusable. Furthermore, the zone capacity implies that the last usable segment in a zone may extend past the zone capacity, and therefore is only marked as partially usable, up to the last writable LBA in the zone.

With ZNS exposing the erase unit (a single zone) of the storage device, F2FS can align the allocation unit to the erase unit of the device. Information about the erase unit is commonly hidden behind the FTL and requires reverse engineering tools, such as Queenie [182], to be estimated. Therefore, default F2FS is configured to contain a single segment in a section and a single section in a zone. With ZNS however the erase unit is known, and F2FS is configured at file system format time to have the number of segments in a section be equal to the number of segments that fit within a ZNS zone, and a F2FS zone consists of a single section. As a result, physical data separation is adequately configured, due to the knowledge of the device erase unit, as illustrated in Figure 5.1.

5.2.2. F2FS Zone Management

The active zone limit introduced by ZNS must similarly be enforced by F2FS to limit the concurrently written data logs, if the number of active zones is lower than the 6 default logs of F2FS. Therefore, for devices that support a fewer number of active zones, the number of active logs can be decreased depending on the number of active zones on the device. With less active zones, the possible configurations are either four concurrent logs, for which the cold metadata and data log is no longer used, or two active logs where only the hot metadata and data logs are used.

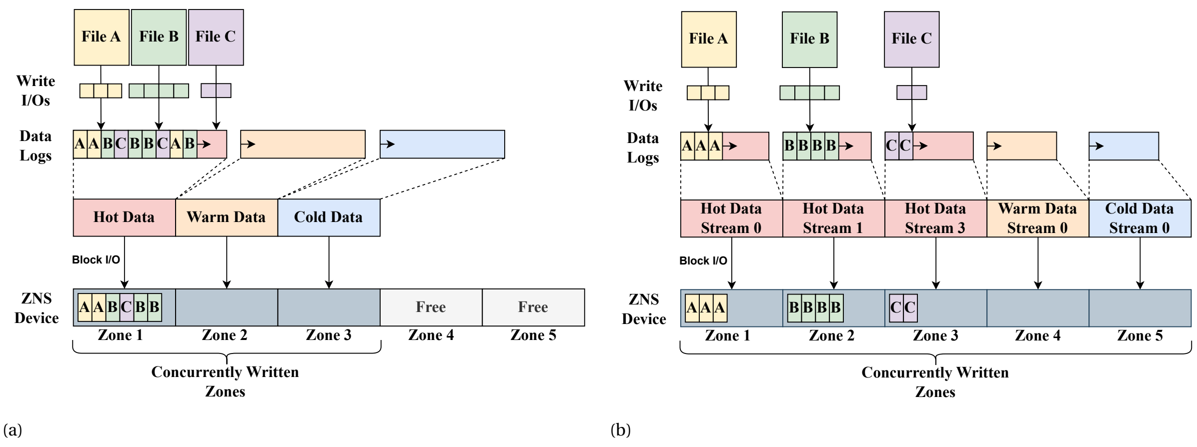


Figure 5.2: **Zone layout** for (a) F2FS mapping sections and containing segments to ZNS zones, and (b) our msF2FS extension of F2FS mapping 2 additional hot data streams to ZNS zones, providing data separation inside a zone, and leveraging ZNS parallelism capabilities.

Write Buffering. While F2FS is a LFS, with direct I/O enabled, bypassing page cache, write ordering cannot be guaranteed. Consequently, only a single outstanding I/O request per zone is possible with conventional writes. Therefore, F2FS currently does not support direct I/O and relies on buffered I/O in the page cache. Buffered I/O additionally allows the I/O scheduler to maximize the merging of I/O requests and achieve closer to NVMe *Maximum Data Transfer Size (MDTS)* sized I/O requests, which has been shown to be beneficial for performance and saturating the device bandwidth and leveraging the on-device parallelism [261, 281].

Garbage Collection. During GC in the file system segments are marked as no longer valid, if all usable segments in a section are marked as invalid, the zone on the ZNS device is reset. With the unit of GC being a section, which is equivalent to a ZNS zone, upon cleaning of a section, the zone no longer contains valid data and can be reset. However, similar to the consistency guarantees of discard commands, zone reset commands are delayed until the next checkpoint is written, such that in the case of a roll-backward recovery, the metadata in the latest checkpoint points to existing data in a zone that has not been reset.

5.3. msF2FS - Design of Multi-Streamed F2FS

While F2FS refers to the different logs for varying lifetime temperatures as *streams* or *multi-headed logs*, these are sequentialized. Particularly, data blocks must be persisted prior to node blocks in order to provide consistent recovery. In case of system failure during writes, recovery must be able to use the node block that points to valid data, hence the data blocks must be persisted before the node block pointing to this new data is persisted. Furthermore, node blocks must be persisted before directory data blocks are written, for the same consistency reasons. While such requirements are not needed for data logs, where different temperature lifetimes can be written concurrently, there only exists a single log for a particular lifetime classification. While the utilization of 3 logs allows for data separation, there is no data separation for concurrently written data of the same lifetime, resulting in all data of the lifetime contending for the single log. Figure 5.2a illustrates the contention of write requests from 3 hot data files, which F2FS maps into the same hot data log, as it only has a single log for this classification, resulting in intra-file fragmentation and write contention on the single resource.

Particularly, with the new ZNS interface providing the capabilities of enhanced data grouping through the separation of zones, F2FS fails to fully leverage the ZNS capabilities. To enhance the integration of ZNS, we design ***Multi-Streamed F2FS (msF2FS)***, a ZNS optimized LFS based on F2FS. msF2FS introduces a new abstraction, called a **Stream**, which leverages ZNS by mapping a stream to a zone, and allocating data logs of of the same lifetime classifications in concurrent streams. As a result, msF2FS provides numerous data logs for a single lifetime classification, increasing the data parallelism capabilities and providing enhanced data separation possibility. With msF2FS, the number of streams for the three lifetime classifications (hot/warm/cold) can be configured, allowing users to customize the number of msF2FS logs that can concurrently be written based on their workload characteristics.

With the *stream mapping* of msF2FS, it is capable of creating a higher number of concurrently writable

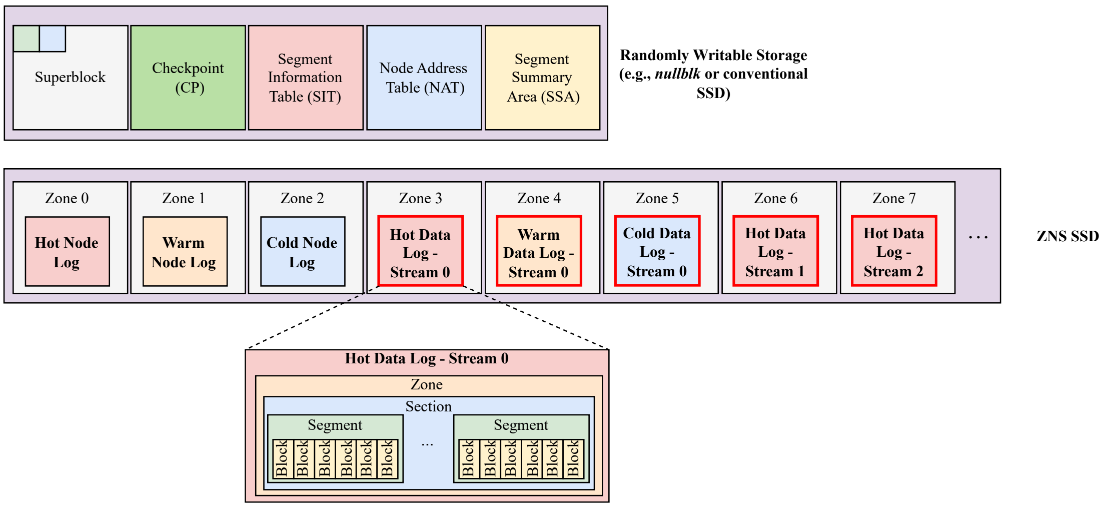


Figure 5.3: msF2FS mapping metadata to conventional randomly writable storage, and logs for different lifetime classifications to ZNS zones, with three hot data streams.

logs for lifetime classifications. Figure 5.2b depicts the same scenario as previously illustrated for F2FS, where concurrent write requests of 3 different files with the same hot data lifetime classification are mapped into different streams. In addition to benefiting from increased concurrency due to the elimination of resource contention, msF2FS furthermore decreases the intra-file fragmentation in the individual data logs, providing potential reduction in future GC overheads.

5.4. msF2FS Stream Layout

With msF2FS utilizing a larger number of concurrently writable data logs, there are subtle changes to data layout, from the conventional F2FS layout. Figure 5.3 depicts the on-device layout of msF2FS. The randomly writable metadata (Superblock, CP, SIT, NAT, and SSA) are stored on a conventional block device. msF2FS maps the node logs (hot, warm, and cold) to the first three zones, with each occupying a single section and therefore mapping to a single ZNS zone. Allocations for data logs are in the order of hot to cold for the first stream (stream 0), serving as the default allocation of streams. Next, the streams for the remaining data logs are allocated to the next available zones, in the order of hot to cold for the user-specified configuration of streams. The figure illustrates a stream configuration with 3 hot data streams, 1 warm data stream, and 1 cold data stream.

5.4.1. Stream Allocation

With msF2FS providing the ability to utilize multiple streams for a single lifetime classification, it must now decide on the block allocation among the different streams. Particularly, block allocation within streams must match the desired write patterns of files, identifying the tradeoff between allowing increased parallelism by utilizing all available streams, and the resulting intra-file fragmentation if block allocations in a single file are split across streams. Therefore, msF2FS employs several stream allocation policies, which we iteratively evaluate during development to avoid shortcomings of one policy in later policies.

During mount time of msF2FS the number of individual streams for each of the data lifetime classifications is configured by the user, along with the stream allocation policy to utilize for block allocations among the streams. If there is only a single stream specified for a particular lifetime classification, the policy does not matter, as all block allocations are mapped to this single stream. Therefore, such a configuration is nearly identical to the default implementation of F2FS. With the iterative development approach, we design two stream allocation policies. The policies have different feature sets, and block allocation that results in varying file-to-stream mappings. In this section we detail the design of each stream allocation policy, and specifically state the reasons for particular design, such as leveraging frequent application write characteristics we encountered during benchmarking, and being able to represent their patterns during block allocation.

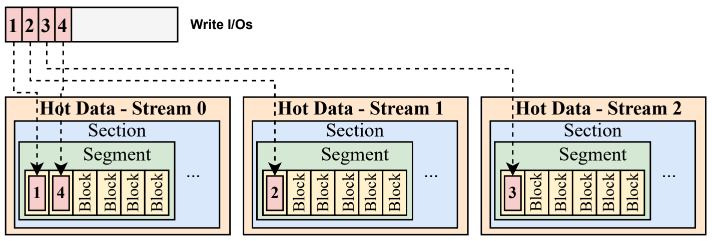


Figure 5.4: msF2FS block allocation with SRR stream allocation policy and write I/O requests from a single hot data file to three hot data streams.

Streamed Round-Robin (SRR)
This policy implements a naive approach at allocating blocks among the available streams for a lifetime classification. A particular goal of this policy is to maximize the data distribution across available resources by evenly allocating blocks across streams in a round-robin fashion. As a result, all available streams are utilized, even if just a single file is being written. The round-robin allocation, depicted in Figure 5.4, starts at stream 0, allocates a single block on this stream, and continues to the next stream on which it allocates the next block. Once all streams have a block allocated, the next block is allocated on stream 0 again. As a result, SRR provides a possible file-to-stream mapping that maps block allocations for a single file to all streams. While this policy allows to maximize the stream utilization, allocating data to all available resources, it suffers from increased fragmentation. As a result, the file blocks are fragmented into individual blocks that are spread across the available streams.

In order to avoid the significant intra-file fragmentation with round-robin based block allocation among streams, SRR furthermore adds the possibility to configure a stride value (termed *rr-stride*). This value is configured at mount time, and groups block allocations in a stream before moving to allocations in the next stream. For instance, a stride value of 512 allows the block allocations to fill an entire segment ($512 * 4\text{KiB} = 2\text{MiB}$) on one stream, before allocating on the next stream. Filling entire segments is beneficial for several reasons. Firstly, it decreases the fragmentation by allocating consecutive blocks in the same segment, always filling an entire segment before allocating in a different segment. Secondly, it maximizes the write unit. The NVMe *Maximum Data Transfer Size (MDTS)* of ZNS is commonly large (several hundred KiB to a few MiB), allowing the consecutive block allocations to be configured to maximally fill the outstanding requests to the particular zone the stream maps to. The aim of SRR with a stride value configuration is to allow maximizing the achievable bandwidth of each ZNS zone.

Stream Pinned Files (SPF)
Incorporating the shortcomings of the SRR policy, this policy adapts the implementation to avoid intra-file fragmentation. Instead of allocating blocks in round-robin manner, SPF maps files to a stream in round-robin, stores the assigned stream in the VFS inode of the file, and only uses the assigned stream for future block allocations of this file. The stream that a file is assigned to is selected in round-robin fashion among the different files, in order to maximize the inter-file parallelism by mapping files to different streams, and decrease fragmentation by avoid multiple files mapping to a single stream. The benefit of only a single file being mapped to a stream is only possible as long as the number of files is less than or equal to the number of available streams. As soon as there are more files than streams, block allocations must overlap in any of the streams.

While SPF allows to decrease the intra-file fragmentation, it requires multiple files to be written concurrently in order to utilize all available streams and maximize device utilization. Furthermore, pinning block allocations to files independently allows simplifying application management of streams, requiring no further configuration beyond the number of streams for lifetime classifications, unlike the stride configuration of SRR. Figure 5.5 illustrates the stream allocation for three hot data files and three hot data streams, where each file is mapped to an independent stream, allowing block allocations to result in no intra-file fragmentation, and utilizing all available resources.

5.4.2. Application-Guided Data Placement

While SRR and SPF stream allocation policies allow to leverage the parallelism and data grouping capabilities of msF2FS streams, we identify that naive policies lack the possibility to adequately map allocations to write patterns. For instance, when SPF is utilized with more concurrently written files than available streams,

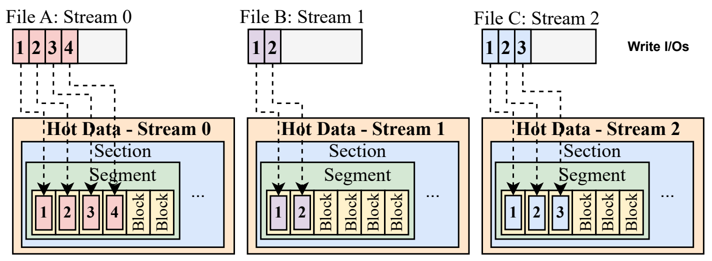


Figure 5.5: msF2FS block allocation with SPF stream allocation policy and write I/O requests from three hot data files to three hot data streams.

contention on the available streams results in block allocations for different files being mapped to the same stream. While mapping multiple files to the same stream is not avoidable, a more comprehensive integration of the block allocation decision is required, taking into consideration which files are optimal choices for interleaved block allocations in the same stream. Identifying the optimal files requires coordination between the application writing the file and msF2FS deciding on the stream allocation. For this purpose, msF2FS introduces **application-guided data placement**, with which the application passes hints to msF2FS, which are used during the block allocation decision-making. Emphasizing on the goal of closing the semantic gap between the storage device and its interface, we leverage the opportunity of ZNS to further the closing of the semantic gap to the file system level, providing enhanced coordination between applications and file system data placement decision-making. In this section we present two integrations of different hinting policies (Sections 5.4.2.1 and 5.4.2.2), and provide an example scenario of an application integrating with the hinting mechanisms (Section 5.4.2.3).

File Exclusive Streams

The first hinting mechanism is built on top of the SPF stream allocation policy and introduces **file exclusive streams**, where only a single file is possible to be mapped to particular stream. Using `fcntl()` on a file, an application sets a flag in the VFS inode, requesting an *exclusive stream* to be assigned to the file during the next block allocation. The setting of the inode must occur prior to the first block allocation (i.e., the first write I/O) for the respective file. As a result, the first block allocation of this file finds a stream that it can exclusively reserve, maps the file to this stream, and only allocates blocks for this file on this particular stream.

The aim of exclusively reserving streams for block allocations is to provide the particular file with no intra-file fragmentation, ensuring that its block allocations are not mixed with blocks of other files. As a result, exclusive streams provide application-guided data placement to ensure life- and death-time segregation of file data. After the file holding the exclusive stream is deleted, there are no more valid blocks in its segment(s), eliminating the need for possible GC for this region. In the scenario where a file that is requesting an exclusive stream is mapped to a stream that other files are also mapped to, the file requesting this particular stream exclusively locks the stream, and other files are migrated to a different stream, called *stream migration*. During stream migration, the other files are moved to the next available stream in round-robin manner, where the selected stream becomes the newly mapped stream for this particular file, resulting in future block allocations to remain on this stream. Exclusively claiming streams is only possible for a number of files that is less than the number of available streams for their lifetime classification. Particularly, because there must always be at least one stream for block allocations, in order to avoid starvation for block allocations if no streams are available, for which stream 0 is a *non-reservable stream*. In the worst case, all streams are reserved by files, and msF2FS falls back to any allocations on a single stream, equivalent to the default F2FS implementation. Therefore, if an exclusive stream is requested by a file, streams starting from stream 1 are checked for the first available stream.

However, if all streams are already reserved by other files, the exclusive stream request fails, resetting the inode exclusive stream request flag, notifying the host about the failure in the kernel log, mapping the file to stream 0, and allocating blocks on this stream. Note, that the application calls `fcntl()` before the first block allocation, and this only sets an inode flag to request an exclusive stream. Therefore, if the request for an exclusive stream fails, the inode flag is simply reset, and the call by the application returns successfully. Furthermore, when a file that is holding an exclusive stream has finished writing all data, it can release the exclusive stream, allowing other files to utilize the stream again. To release the stream the application must issue a `fcntl()` call with the flag to unset the exclusive stream request. Future block allocations for other files can then again map to this stream, and this particular stream can again be exclusively reserved by another file.

Application Managed File Streams
Similar to the prior SPF policy, giving the host application a degree of decision-making capabilities on stream allocation, this policy aims at furthering the degree of stream management by the application by introducing ***Application Managed File Streams (AMFS)***. Particularly, AMFS minimizes the allocation decisions made by the file system, requiring the application to provide the desired allocation information. By default, AMFS maps all block allocations (for every file) to stream 0, making its allocation policy identical to that of default F2FS. However, AMFS furthermore expects the application to provide a mapping for each file, depicting on which stream it shall allocate data blocks. This mapping information is passed from the application to msF2FS using `fcntl()` and providing a pointer to a bitmap. The bitmap indicates the streams to use for the block allocations of the file. The passed bitmap is copied to the respective inode of the file, which msF2FS checks on block allocations. With such a mapping, the application is in full control of what files are mapped to which streams, which blocks of files can be interleaved in the same segment, which files may not encounter any intra-file fragmentation, and which files require increased parallelism by allocating on multiple streams.

As a result, AMFS pushes the stream management responsibility to the application, decreasing the semantic gap by allowing the application to decide on data placement. Given there are a limited number of streams available, applications must carefully design mappings for particular files. Furthermore, this allocation avoids having to manage the release of resources (i.e., releasing an exclusive stream) by clearing inode flags. If a file is mapped to multiple streams, by multiple bits in the passed bitmap being set, AMFS aims to minimize the intra-file fragmentation similar to the rr-stride of SRR, where it firstly fills the current segment of a stream, before allocating in the next stream indicated by the bitmap. As a result, allocations fill the segments, decreasing intra-file fragmentation and similarly allow maximizing bandwidth by increasing stream allocations to get closer to the device MDTS.

Example Workload: RocksDB with msF2FS and Application-Guided Data Placement
In this section we illustrate the block allocations for the two different hinting mechanisms for application-guided data placement. These examples utilize RocksDB files, as we evaluate the implications of application-guided data placement integration in RocksDB in Section 6.6. However, the illustrations can be adapted for any particular file.

File Exclusive Streams. Figure 5.6 illustrates the stream migration for a file with an example scenario with two RocksDB files. The scenario depicts that the first RocksDB file (`00012.sst`) is mapped to stream 1, which is a result of file `A` being mapped to stream 0 before file `00012.sst` (round-robin stream assignment). Hence, the first allocation of `00012.sst` maps to stream 1. A single block is written for file `00012.sst`, while an additional two files `B` and `C` are mapped to streams 2 and 0, respectively. As a result, the first allocation of a different RocksDB file (`00055.sst`) ends up being mapped to the same stream as the `00012.sst` file, and it furthermore reserves this stream exclusively. As a result, the prior `00012.sst` file is being migrated to the next available stream 2, where all future block allocations occur, resulting in the inode being modified to depict the updated stream mapping.

Application Managed File Streams. Figure 5.7 illustrates an example scenario of RocksDB utilizing AMFS in the application-guided data placement for a SSTable file (`00012.sst`). The file is mapped to streams 1 and 2, indicated by the respective bits being set in the bitmap (`streammap` field of the inode). The bitmap in the inode indicates the stream to use based on the index of the set bits in the bitmap, where the rightmost bit being set indicates to use stream 0. Similarly, setting the second to the rightmost bit sets block allocations to stream 1. Therefore, the bitmap of `0000000000000110` contains 16 bits (as 16 is the maximum number of total streams), indicating to use streams 1 and 2. The block allocations for `00012.sst` therefore fill the first segment in stream 1, followed by the first segment in stream 2, and continuing allocations in the second segment of stream 1.

5.5. Active Zone Management

msF2FS allows to utilize a maximum of sixteen total streams, of which six are used for the default msF2FS allocation of a single log for each data and node type, leaving a remaining ten streams to be configured. However, devices may not have support for sixteen active zones and require less to be configured. Furthermore, since it uses buffered I/O, block allocations in msF2FS may remain in page cache for some time. If a stream finishes allocating blocks in a section it must allocate a new section in the next free zone. However, the block

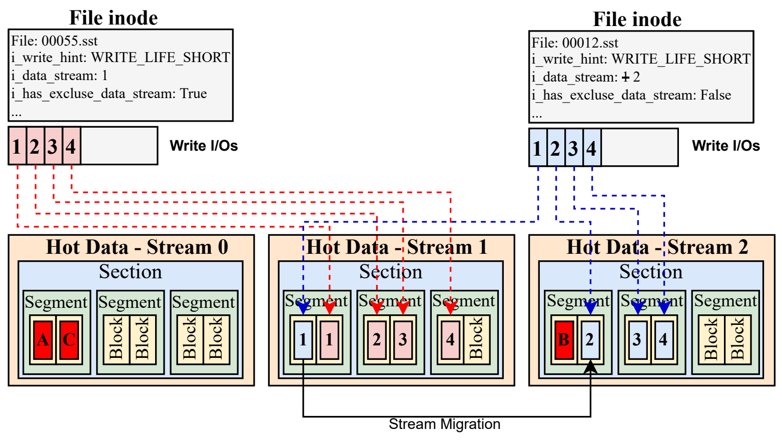


Figure 5.6: Files `00012.sst` and `00055.sst` are mapped to the same stream (`i_data_stream`: hot data - stream 1), however `00055.sst` requests an exclusive stream, resulting in block allocations for `00012.sst` being migrated to the next available stream (hot data - stream 2). Red boxes indicate block allocations for different files (files `A`, `B`, and `C`).

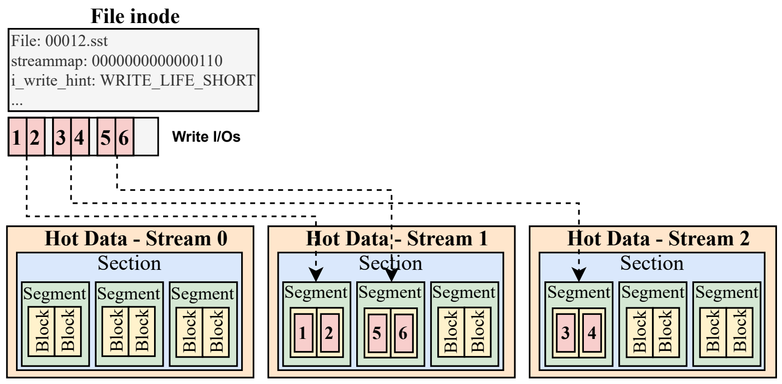


Figure 5.7: Block allocation for a RocksDB SSTable file (`00012.sst`) with AMFS stream allocation policy, and the file having a stream mapping bitmap in the inode indicating to map allocations to stream 1 and stream 2. Allocations fill an entire segment in a stream before allocating in the next mapped stream.

allocations in the prior section may still be in page cache. During writebacks all the outstanding writes in page cache may be written back to the device. If the streams are configured to use the maximum number of active zones, a stream allocating a new section while it still has outstanding writes in page cache exceeds the number of active zones, resulting in I/O errors on the device. In order to solve this effectively, direct I/O is required or write barriers must be introduced. Direct I/O requires significant redesign of the msF2FS I/O management, with for instance the modification of writes to use the ZNS append command. Write barriers, flushing any outstanding writes prior to a new section allocation, similarly introduce complexity for flushing outstanding I/Os in a zone before allocating a new zone.

msF2FS on the other hand uses a proactive approach during section allocation, called *Stream Section Allocation Barrier (SSAB)*. If a stream finishes a section, it checks the total number of active zones on the entire device and only allocates a new section if the number of active zones is not greater than the maximum number of active zones for the device. Therefore, if a stream cannot allocate a new section because the number of active zones has reached the limit, the block allocation uses a first-fit policy starting from stream 0, to find the first stream that has space to allocate the block. This allocation is repeated, starting at stream 0 again on the next block allocation, until the number of active zones decreases, which is checked during each block allocation for the stream. Periodic writebacks and explicit `fsync()` calls result in the outstanding writes to be issued to the device and the zones being filled, resulting in the device releasing the resources for active zones.

At least one zone must become inactive in the future as a stream is attempting to allocate a new section, it must have finished allocating in its current section. These writes may still be outstanding in page cache, but must be written to the device at some later point, resulting in at least one zone becoming inactive and allowing the stream to allocate a new section. SSAB however cannot guarantee that the number of active zones is not exceeded. If a stream encounters a SSAB, it attempts to allocate the block in one of the other streams in a round-robin manner starting from stream 0. If however, all streams are full, allocation falls back to the last stream, which if it is also full, allocates a new section regardless. While this particular scenario is highly unlikely, due to requiring all streams being full at the exact same time, this can be controlled by the user. If the user configures the total number of streams to be less than the maximum number of active zones, the possibility of exceeding the active zones is decreased. Based on evaluations, we recommend configuring the total number of streams to be at least 1-2 less than the maximum active zones, allowing for more safety on section allocations.

5.6. msF2FS Implementation

In this section we describe the implementation details of the varying aspects to integrate msF2FS based on F2FS. For the implementation we base the source code on Linux kernel version 5.19. We describe only the modifications we make on top of F2FS, and we do not detail the entire conventional F2FS implementation. Furthermore, as kernel development can become complex, we detail our setup and process of debugging and verifying code correctness in Appendix C. We aim for such information to be useful for programmers starting with Linux kernel development, illustrating how to set up, configure, and debug Linux kernel source code, and to showcase the integration of the developed `zns-tools` into our msF2FS development cycle. We furthermore provide several helpful commands and setup instructions we utilize during the development of msF2FS in Appendix C.

5.6.1. Stream Management

A common structure for all file systems is the superblock (`struct super_block`), storing all required information for the kernel to be able to mount the file system. When a file system is mounted, an object for its superblock is constructed by the file system reading its own superblock from persistent storage, and assigning respective values to the fields of the constructed superblock. Within this superblock is a pointer to file system specific information that is kept in memory during runtime, and allows the file system to manage its data structures. In F2FS this data structure is called the *sbi*, which msF2FS extends to maintain several data structures for stream management. Of particular importance are a bitmap that is kept for each of the data and node lifetime classifications, for which msF2FS sets respective bits in this bitmap on which particular streams are configured to be active. Configuration is provided by the user at mount time, with specific values for the number of hot/warm/cold data streams to use, at which point msF2FS initializes the bitmaps. During runtime, msF2FS checks possible block allocations on a stream by verifying the respective bit for the stream in the bitmap is set. The additional configuration options that can be passed by the user at mount time are similarly stored in the *sbi*. These include the type of stream allocation policy that is being used and the stride for SRR allocation. Additional structures are required for each of the different stream allocation policies, which we describe in their respective section.

Concurrency Control.
With the addition of data structures to the *sbi*, we ensure concurrency control with several mechanism. All allocations to the same segment require concurrency control, which the segment manager implements through semaphores, which readers and writes must lock in order to read or modify the segment information. Semaphores allow multiple readers concurrently, as they all read valid data, however only a single writer is allowed at a time, additionally blocking all readers. Similarly, the bitmaps in msF2FS are protected with spinlocks to ensure concurrency control. The utilization of the atomic type interface of the kernel allows ensuring concurrency control on data that relies on a single value. As a result, msF2FS ensures concurrency control on all global data structures using these particular interfaces provided by the Linux kernel. In order to avoid locking overheads, the values that are only initialized at mount time, and are never changed, have no locking. As these are only ever read, there is no possibility for race conditions or invalid data reads.

Garbage Collection.
During the process of GC, valid blocks are moved to the cold data streams, possibly resulting in a reclassification for a file (e.g., moving hot data blocks to cold), making the prior set stream information in the inode no longer valid. Particularly, because the cold data streams may be configured differently than other data streams, it can have fewer streams, requiring the inode information on the pinned stream for a file to be reset. Furthermore, msF2FS pins all GC to stream 0 of the cold data, resulting in any blocks moved during GC to receive blocks allocated on stream 0 of the cold data. As the data was infrequently modified, resulting in it being garbage collected, it is likely to not be modified in the future, allowing to pin these files to the same

stream. If more cold data streams are configured, these can be utilized by direct cold data writes, issued by the host and not the GC process, by mapping the files to these streams with AMFS or SPF exclusive streams.

NAT and SIT Caches.
In order to increase the lookup performance of F2FS, it maintains a NAT cache in the hot data segment information, inside the SSA. Upon lookups of a node, this cache is checked for the presence of the block addresses for file data of the node. msF2FS extends the NAT cache to span across the segment information for the utilized hot data streams, which are all read during the lookup process of a node. With multiple configured hot streams, the NAT cache can therefore be increased in size, allowing for a higher probability of cache hits. The SSA summary defines the block for the cache to either be for NAT or SIT entries. Therefore, msF2FS similarly extends the SIT cache, which is indexed by the cold data streams. By configuring a higher number of cold data streams, is the SIT cache in the SSA increasing in size. As a result, msF2FS utilizes the spare area of the SSA to enhance performance with caching.

Extracting Information on msF2FS Stream Utilization
In order to evaluate the utilization of streams in msF2FS, the SIT entries contain the stream number that the segment is on. This change only implies for already allocated segments, as free sections and their containing segments can be allocated to any of the streams and types. Using this information, the `debugfs` entry for msF2FS is adapted to depict this information on the streams. The entry furthermore contains all default statistics of F2FS utilization, including block allocation statistics, extent information, and more. Figure 5.8 depicts the additional fields of msF2FS, showing the statistics of the file system during a macro-benchmarking workload. The particular example is configured with 2 hot data streams, 4 warm data streams, and 2 cold data streams, under SPF allocation policy. Of the depicted information, we can identify the information kept in the *sbi* for the stream management, and the utilization of exclusive streams, which are not used in this example. The main statistics for the different streams depicts their current segment, its section number, zone number, and the stream information on allocation statistics. With the number of dirty and full segments, we are able to identify the utilization of a stream. Particularly, if a stream has little dirty segments, this implies that either no data on the stream is deleted, or entire sections in the stream can be reclaimed without GC. In this particular workload, files are frequently deleted, with the most common characteristic of files being written once and then only being deleted. In the warm data stream, we can identify that data placement is adequate because it has little dirty segments, implying that when data of a file is deleted, the section does not contain still valid data from other files, and therefore does not need to be garbage collected. This is particularly a result of decreased fragmentation and enhanced data grouping, where files that are mapped to the same stream are deleted within close proximity. On the other hand, the cold data streams have numerous dirty segments, implying that they contain a higher degree of fragmentation, which likely can be decreased with a configuration that utilizes more cold data streams.

5.6.2. msF2FS Application-Guided Data Placement Hinting Mechanisms

Similar to SPF pinning files to streams by setting fields in the inode, the application-guided data placement can be achieved in combination with the stream management data structures in the inode. For setting of exclusive data streams with SPF, a call by the host application to `fcntl()` is required, passing the `F_SET_EXCLUSIVE_DATA_STREAM` flag to request the exclusive data stream, which sets a respective flag in the inode. The `fcntl()` functionality is implemented by adding the required additional flag to the possible options and parsing the user provided flag. If the exclusive stream flag is set in the inode, during the next block allocation msF2FS checks if it can reserve an exclusive stream. If a stream is available, it similarly to SPF sets the pinned stream information, and additionally sets a map inside the *sbi*, which tracks the inode numbers that are holding an exclusive stream. As a result msF2FS is able to check which inode is holding which exclusive stream. To release the exclusive stream resources, the application calls `fcntl()` with the `F_UNSET_EXCLUSIVE_DATA_STREAM` flag, which clears the flag in the inode for an exclusive stream request.

The allocation for AMFS, providing the application-guided data placement for enhanced data grouping, similarly requires the host application to issue `fcntl()` calls with the `F_SET_DATA_STREAM_MAP` flag, and passing a bitmap of the streams to use for the particular file. The bitmap is verified and stored in the inode of the file. During block allocations, each inode is checked if this bitmap is set. If the bitmap is present, the respective streams that are specified are used, if the streams are active. If the bitmap contains a set bit for a stream that is not active, the bitmap is disabled and the allocation falls back to stream 0. Similarly, if no bitmap is provided, all block allocations are on stream 0. If multiple bits are set in the bitmap, allocation happens in round-robin fashion, with a stride that fills a segment. By maintaining the number of the segment

```
root@stosys:/home/user# cat /sys/kernel/debug/f2fs/status | grep -A50 -i "multi"
    Multi-Stream INFO:
  - Stream Allocation:
         Policy: SPF
  - Maximum Streams: 11
  - Active Streams: 11
  - STREAMS:
        Maximum:  [  2  4  2  1  1  1  ]
         Active:  [  2  4  2  1  1  1  ]
      EXCLUSIVE:  [  0  0  0  0  0  0  ]
  - ACTIVE STREAM BITMAPS:
       HOT_DATA   [ 1 1 0 0 0 0 ]
       WARM_DATA  [ 1 1 1 1 0 0 ]
       COLD_DATA  [ 1 1 0 0 0 0 ]
  - EXCLUSIVE STREAM BITMAPS:
       HOT_DATA   [ - 0 ]
       WARM_DATA  [ - 0 0 0 ]
       COLD_DATA  [ - 0 ]
  - EXCLUSIVE STREAM INO-MAP:
       HOT_DATA   [ - 0 ]
       WARM_DATA  [ - 0 0 0 ]
       COLD_DATA  [ - 0 ]

    TYPE       STREAM    segno    secno   zoneno  dirty_seg   full_seg  valid_blk
  - HOT  data:      0   144450      141      141        766        919     524788
  - HOT  data:      1   315518      308      308       6783          4     101203
  - WARM data:      0    69768       68       68         16       4644    2382956
  - WARM data:      1    82441       80       80         18       2891    1482928
  - WARM data:      2  2837918     2771     2771         15       3768    1933478
  - WARM data:      3    36301       35       35          9       3079    1579274
  - COLD data:      0   153601      150      150      18900     311492  164337597
  - COLD data:      1   112872      110      110      20077     309479  163609213
  - HOT  node:      0        9        0        0          2          0          2
  - WARM node:      0   244862      239      239       2513          4     331014
  - COLD node:      0     2096        2        2         27          0        713
```

Figure 5.8: Example output, depicting the file system statistics information with 2, 4, and 2 hot, warm, and cold data streams, respectively, under SPF allocation policy.

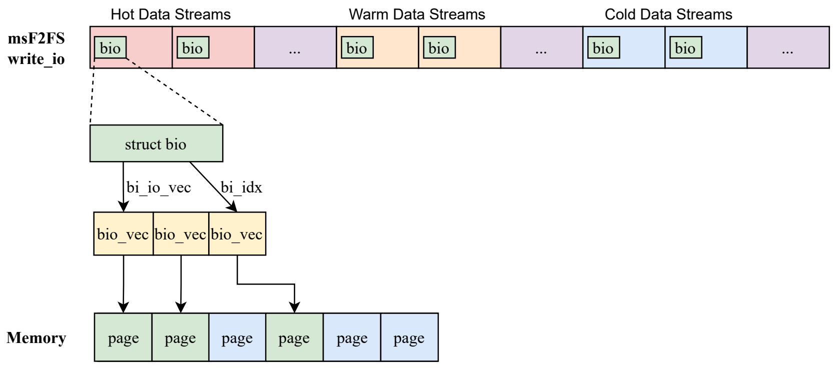


Figure 5.9: I/O structure in msF2FS with a `struct bio` for each lifetime classification and type to increase merging of in-flight I/Os.

that is currently being allocated to, if the next allocation occurs at this segment + 1, meaning the current segment is full, allocation moves to the next specified stream. The aim of this is to decrease fragmentation, and allow increasing the outstanding write I/O requests to the MDTS. Similar to all prior policies, any provided hints are ignored if msF2FS has no section to allocate for a stream or the active zone limit is reached (SSAB is encountered).

5.6.3. I/O Management

With buffered I/O in msF2FS and ZNS devices, it is important to manage the write I/O requests in such a way that not only aligns with the sequential write requirement within zones, but maximizes possible merging in order to reduce the number of I/Os to issue and maximize intra-zone write throughput. For this msF2FS maintains outstanding write I/Os for each stream independently. Figure 5.9 depicts the I/O management for data streams in msF2FS, where each stream of each lifetime classification has one `struct bio`, which tracks the outstanding write I/O requests. The `write_io` array contains a structure for each of the lifetime classification types for the data and node type, we only depict the data streams present. Within this structure, the outstanding I/O request are commonly also referred to as "in-flight" requests. Within each `struct bio` is a vector of `bio_vec`, indexed by the `bi_idx`, which point to the memory pages that contain the data to write to the device in a single I/O request. With such a structure, consecutive write I/Os to increasing LBAs can be merged into fewer requests. Particularly, by msF2FS maintaining the in-flight requests for each stream separately, it is able to maximize the merging within a single stream, and therefore within each of the ZNS zones. The sequential write requirement is fulfilled due to msF2FS inserting write I/O requests in each of the structures in consecutive order, as is a result of the log-based block allocation.

5.6.4. Development Cycle

To develop msF2FS, we utilize a full system emulator, QEMU, with which we boot the compiled Linux kernel with msF2FS, and through which we utilize debugging tools. In addition to debugging, to ensure correctness of the msF2FS, we utilize the developed `zns-tools` to generate workloads, and provide detailed file mappings on the ZNS zones. Of particular importance is the allocation of blocks, under the varying stream allocation policies. Therefore, we commonly issue micro-workloads with `zns.fpbench`, that only write a few MiB of data, followed by identifying the detailed zone mappings with `zns.fiemap` and `zns.segmap`. With the mapping information we are able to identify if particular block allocations are correctly placed under the specific stream allocation policy.

5.7. Summary

Leveraging the possibility of the ZNS interface, we present msF2FS, a F2FS based file system with enhanced ZNS support. The streams of msF2FS allow it to map data logs to different ZNS zones, increasing the parallelism and data grouping capabilities with multiple data logs on the streams for each lifetime classification. Furthermore, building on the foundational shift of ZNS decreasing the semantic gap between flash storage and the storage software, msF2FS provides application-guided data placement, leveraging the coordination between applications and the file system for enhanced data placement decision-making. All source code of msF2FS is made publicly available at `https://github.com/nicktehrany/msF2FS`.

6
Experimental Evaluation

In this chapter we evaluate the performance of msF2FS, with its different stream allocation policies, compared to F2FS, and a domain specific ZNS file system using respective micro- and macro-level benchmarking frameworks and workloads. For structuring of this evaluation, we devise five key *Evaluation Question (EQ)* to evaluate msF2FS by.

EQ1. With the possibility of emulating ZNS devices, does such emulation scale with real ZNS devices, and how should ZNS be configured for quantitative performance evaluations?

EQ2. What is the quantitative micro-level performance impact of leveraging a larger number of concurrent zones in msF2FS through streams?

EQ3. What is the quantitative impact of file fragmentation on GC, and therefore the file system performance?

EQ4. What is the quantitative impact of enhanced data grouping in msF2FS on the zone management activity with zone reset commands?

EQ5. What is the quantitative macro-level impact of msF2FS utilizing concurrent streams with application-guided data placement through hints?

For each of the respective research questions, we devise micro- or macro-level benchmarks to evaluate them. We make all source code of benchmarking scrips and collected data publicly available at `https://github.com/nicktehrany/msF2FS-bench`. To furthermore enhance reproducibility of this evaluation, we provide detailed setup instructions and benchmarking commands in Appendix A.

6.1. System Setup

This section details the hardware and software configurations used during the evaluation. All experiments are run inside a QEMU VM, with the host system and QEMU VM configurations depicted in Table 6.1. The CPU configuration enables hyper-threading on the host system in order to provide the QEMU VM the option to have 34 cores from the 40 available threads (2 threads per core from the enabled hyper-threading). Several threads on the host system are left available, to handle ssh sessions and VM monitoring. The memory is configured to be 25GiB for the QEMU VM particularly as msF2FS requires a randomly writable conventional block device for F2FS and msF2FS metadata, which is emulated using a *nullblk* device of 19GiB. This device is backed by the available RAM, allowing reading and writing the F2FS metadata to this device. The remaining 6GiB aim at minimizing the available system memory in order to increase the memory pressure for msF2FS and F2FS and enforce more frequent writebacks of file data in page cache, allowing to maximize the generated device traffic to the ZNS device. The configuration of a ZNS device firstly follows an evaluation comparing emulated ZNS devices to real ZNS hardware (presented in Section 6.2), based on which the final environmental setup configuration for this evaluation is extended. Lastly, we depict all software versions used during this evaluation, and the respective git commit that is checked out.

CPU	Host: Dual socket Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 10 cores/socket, hyper-threading enabled, with Spectre and Meltdown patches QEMU VM: 34 cores
DRAM	Host: 256 GiB, DDR4 QEMU VM: 25GiB
Software	Ubuntu 22.04, kernel 5.19 (built from source for F2FS and msF2FS), QEMU (7.1.0), fio (3.32, git commit: 6b6f52b), nvme-cli tools (2.2.1, git commit: b7ac8e4), f2fs-tools (1.15.0), RocksDB and db_bench (7.3, git commit: 01fdec2), ZenFS (2.1.2, git commit: b04ca0c), bpftrace (0.16, git commit: ed06d87)

Table 6.1: Benchmarking environment.

ZNS Device 1	Western Digital PCIe 3.0 Ultrastar DC 8,192 GB ZN540 (product link), zone size: 2,048 MiB, capacity: 1,077 MiB, number of zones: 3,688, Max. active zones: 14
ZNS Device 2	Western Digital PCIe 3.0 Ultrastar DC 1,024 GB ZN540 (product link), zone size: 2,048 MiB, capacity: 1,077 MiB, number of zones: 904, Max. active zones: 14

Table 6.2: ZNS devices evaluated.

6.2. ZNS Evaluation Environment

While ZNS devices are now commercially available, frameworks such as FEMU [180], provide functionality to emulate ZNS devices, begging the question on the performance validity of FEMU, compared to real ZNS hardware. Therefore, we compare the performance scalability of FEMU to two real types of ZNS devices, in Section 6.2.1. Furthermore, the configuration options of ZNS devices, allowing different *LBA Format (LBAF)* (512B or 4KiB), requires investigating the intra- and inter-zone performance capabilities of the respective LBAFs. Such investigation is conducted in Section 6.2.2

6.2.1. Performance Scalability of Emulated ZNS

With our evaluation being on Linux kernel 5.19 (base kernel of msF2FS), we encountered issues with FEMU emulating the ZNS device. Therefore, we provide a patch to port FEMU to the 5.19 kernel, fixing a bug in the NVMe controller startup, which has been merged into the mainline FEMU [292]. To compare the performance of FEMU to real ZNS devices, we make use of two different ZNS devices, whose characteristics are depicted in Table 6.2. We evaluate the intra-zone (inside a single zone) and inter-zone (across multiple zones) scalability of each. The different devices are configured with a LBAF of 4KiB, for which FEMU requires minor code modifications, which are detailed in Appendix A.3. Similarly, the instructions to modify the LBAF on the ZNS devices are depicted in Appendix A.2. The ZNS devices are configured to expose the maximum available conventional space of 4.29GB (ZNS-1) and 2.15GB (ZNS-2), which is not utilized and left empty during the benchmarks. The remaining capacity of the ZNS device is exposed in a single namespace for all sequentially written zones. On both devices, both of the namespaces are created and attached to the device controller 0. The device is passed from the host to the QEMU VM through VFIO-PCI, to minimize virtualization overheads.

The achievable throughput of the ZNS devices is measure using fio, configured to issue 4KiB sequential write I/O requests with `io_uring`, setup with submission and queue polling, following its recommended configuration [60]. To measure the intra-zone scalability, the number of outstanding I/O requests (`iodepth` in fio) is increased from 1 to 1,024, identifying the saturation throughput of a single zone. The devices are configured to be setup with the `mq-deadline` scheduler, resulting in merging of I/O requests in a single zone. The inter-zone scalability is measured by maintaining an `iodepth` of 1, and increasing the number of jobs that concurrently write to different zones. The number of concurrent jobs ranges from 1 to 14, equal to the maximum number of active zones, where each job writes a total of 7% of the available capacity, and starts at an offset of 7%. This ensures that jobs do not issue write requests to the same zone, and that no reset

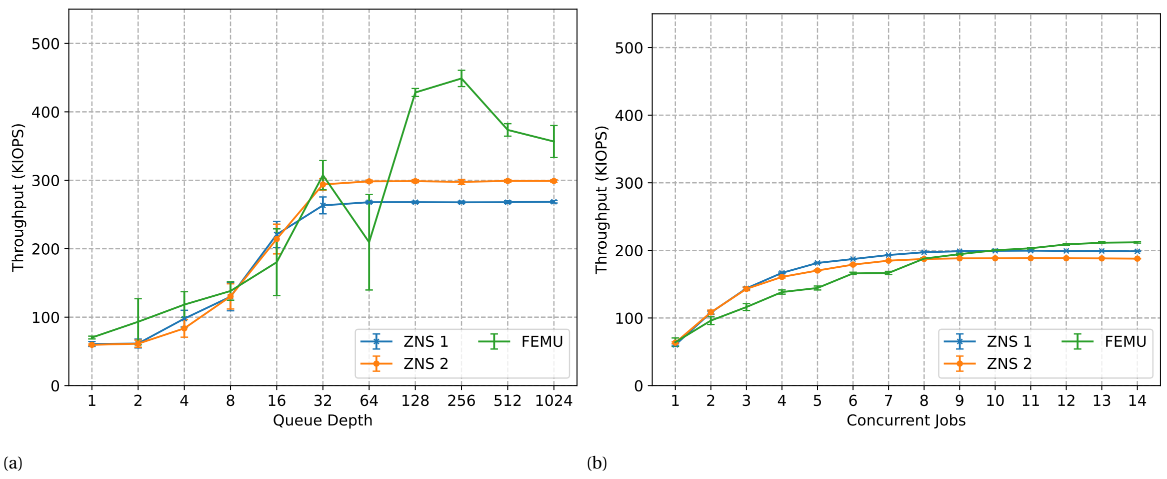


Figure 6.1: Comparing achievable write throughput of FEMU ZNS emulation to real ZNS devices for (a) intra-zone scalability, and (b) inter-zone scalability.

commands are issued during the benchmark, avoiding reset overheads. Furthermore, between benchmarks all zones on the ZNS device are reset.

The resulting throughput is depicted in Figure 6.1, with (a) the intra-zone scalability, and (b) the inter-zone scalability with errors bars representing the standard deviation. Intra-zone scalability of FEMU, achieves a peak throughput that is between 50.23% and 67.45% higher than the ZNS devices. However, throughput of FEMU is less stable, with higher standard deviations, and the inter-zone throughput (Figure 6.1b) of FEMU scales slower than both of the ZNS devices. The current device emulation of ZNS does not provide configuration options for the zone mapping, such as configuring the chips/channels/planes to which zones map. Such interface is available for OCSSD emulation with FEMU, which we envision to be valuable future work to integrate similar configuration options into the ZNS emulation. However, due to the lower throughput stability and the decreased inter-zone scalability, we currently recommend to utilize real ZNS devices. Furthermore, given the highest inter-zone scalability is with ZNS device 1, we utilize it for all evaluations in this section. Particularly, as msF2FS aims to increase the ZNS zone parallelism, high inter-zone scalability is required.

Summary. The increased inter-zone scalability of real ZNS devices and the increased stability in achievable throughput makes it a more suitable device to evaluate msF2FS on. However, the intra-zone scalability of FEMU emulated ZNS is capable of achieving an up to 67.45% higher throughput than real ZNS hardware.

6.2.2. ZNS Device Setup

With the selected ZNS device, we extend its evaluation of intra- and inter-zone scalability under its possible LBAF configurations, to identify the desired LBAF (512B or 4KiB) for the selected ZNS device. The benchmark is configured to issue micro workloads with fio that submit sequential write I/Os of 4KiB with the `io_uring` engine, with submission and queue polling that follows its recommended configuration [60]. In order to measure the intra-zone (inside a single zone), the number of outstanding write requests (`iodepth` in fio) is increased in that single zone. The device is configured to be setup with the `mq-deadline` scheduler, resulting in merging of I/O requests in a single zone. To measure the scalability of concurrently writing multiple zones, the number of concurrent jobs is increased, where each job writes zones at a stride of 100 zones. Jobs have identical configuration, increasing the iodepth in their respective zone. Each job writes a maximum of 100 zones, or a runtime of 30 seconds with a ramp time of 10 seconds. The ramp simply runs the workload without collecting metrics, which are only collected for the runtime duration. This serves the purpose to avoid initial fluctuations in performance, such as from initialization overheads, and reach a more stable performance. The configuration of writing a maximum of 100 zones, with jobs at a stride of 100 zones, avoids overlapping the zones being written for the different jobs, and eliminates the overheads of zone resets affecting the throughput. Furthermore, between each benchmark, all zones on the device are reset, ensuring that no zones are reset during the benchmark.

Figure 6.2 shows the resulting throughput for the scalability with (a) depicting the 512B LBAF and (b) showing the 4KiB LBAF. The x-axis represents the number of outstanding write I/O requests (iodepth), with the various lines depicting the number of concurrent jobs being run, with the error bars indicating the stan-

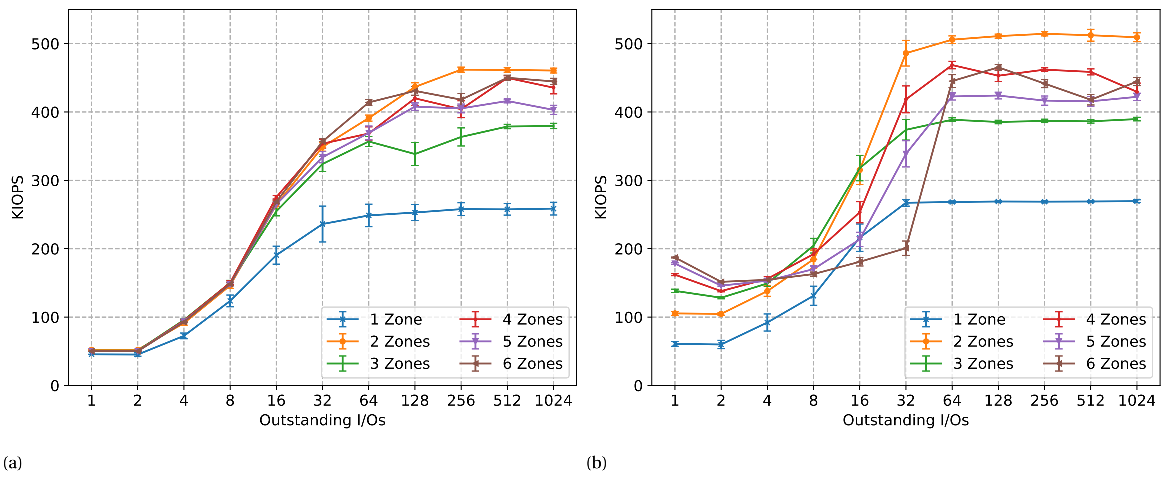


Figure 6.2: Maximum achievable throughput of the ZNS device under increasing I/O depth and concurrent jobs, with (a) the ZNS device having a 512B LBAF and (b) the ZNS device having a 4KiB LBAF.

dard deviation. Results with the 512B LBAF provide a more stable throughput with inter-zone scaling, as opposed to the 4KiB LBAF, where increasing concurrently written zones can have a detrimental effect on the throughput. Particularly with the 4KiB LBAF, 2 concurrent zones maximizes the achievable throughput, where 3 concurrently written zones results in performance drops of 32.93% at 256 outstanding I/Os. 4 concurrently written zones increases the performance again by 19.37%, however with 5 concurrently written zones the performance drops again. While a drop in performance for 3 concurrently written zones is similarly present with the 512B LBAF in Figure 6.2a, it overall provides similar throughput for varying levels of intra-zone concurrency. The maximum throughput with 512B LBAF is limited 11.36% lower than with 4KiB LBAF, however its increased stability for inter-zone utilization with 512B LBAF makes it a preferred configuration for the evaluation of msF2FS.

Summary. Evaluations identify that on the particular ZNS device used during this evaluation the 512B LBAF provides a more stable inter-zone scalability at higher queue depths, albeit achieving a slightly lower maximum throughput than the 4KiB LBAF. Therefore, for the duration of this evaluation the ZNS device is configured with the 512B LBAF.

6.2.3. Summary

Findings of the micro-level benchmarks identify that the inter-zone performance of current ZNS emulation fails to scale with the capabilities of real ZNS hardware. Furthermore, the ZNS device utilized during this evaluation, provides more throughput stability with write I/O requests to concurrent zones when configured with 512B LBAF. However, the 512B LBAF achieves a peak throughput 11.36% lower than the 4KiB LBAF. Due to the increased stability for inter-zone benchmarks, the 512B LBAF is used as the ZNS device configuration for the remainder of this evaluation.

6.3. Performance Implications of Concurrent msF2FS Streams

With the established ZNS device configuration, we proceed with evaluating the performance implications of msF2FS concurrent streams, and compare it to F2FS with micro-level benchmarking workloads. Firstly, the achievable throughput of msF2FS is measured, with a configuration that allows it to utilize the maximum number of active zones with concurrently writable streams. For this fio is configured with the `psync` engine to issue 4KiB synchronous write I/Os to a hot data file. The hot data file is configured to have the write hint marking it as hot (`write_hint=short` in fio). The current release of fio utilizes a different hint that is no longer supported in 5.17+ kernels, which we patch and provide instructions on reproducing the patch in Appendix A.4. We find that the use of synchronous I/O provides increased and more stable performance, compared to asynchronous I/O engines (i.e., `io_uring`), and therefore utilize it throughput this evaluation. As F2FS and msF2FS enforce buffered I/O with ZNS devices, fio ensures more frequent page cache writeback by issuing explicit `fsync()` calls for every 128 write I/Os submitted. Particularly, as `fsync()` calls flush the data of all streams, a value of 128 aims at increasing the outstanding requests on all streams, while maintain-

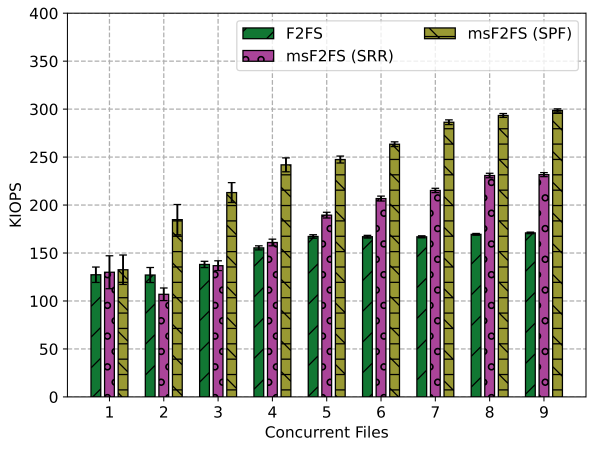


Figure 6.3: Achievable throughput for concurrently written hot data files with F2FS and msF2FS under the two stream allocation policies, where msF2FS with SPF maps each file to a separate hot data stream.

ing memory pressure to flush page cache. Similar to the ZNS baseline, the concurrency is scaled by increasing the number of concurrently run jobs, where each job replicates the same configuration but utilizes a unique file for write requests. F2FS requires no configuration, as it only has a single hot data log. However, msF2FS is configured to have the number of hot data streams equal to the number of files present (e.g., with 4 hot data files msF2FS has 4 hot data streams). Therefore, msF2FS with the SPF policy allows mapping each file to its own distinct hot data stream.

The resulting throughput, depicted in Figure 6.3, showcases the scalability for msF2FS over F2FS. The x-axis shows the number of concurrently written hot files, configured with the number of jobs in fio, and where msF2FS is configured with the number of hot data streams being equal to the number of files being written. Error bars indicate the standard deviation of KIOPS. The results show that F2FS scales by 34.25% with increasing the number of concurrently written files, whereas msF2FS with SRR policy scales 78.28%, and with the SPF policy scales a total of 125.29%. The resulting maximum throughput of msF2FS with SPF provides a 74.68% increase over F2FS with 9 concurrently written hot data files. While the ZNS baseline experiments shows a saturation of the device throughput at 2 concurrently written zones, msF2FS continues scaling with up to 9 concurrent streams. While, the maximum throughput of msF2FS of 299.65KIOPS at 9 streams surpasses the maximum throughput of a single written zone (Figure 6.2a), it is below the maximum achievable throughput for 2 or more concurrently written zones, resulting in msF2FS leveraging a higher degree of utilization of each zone, however failing to saturate the zones. The increased performance is particularly a result of merging I/O requests and issuing larger I/Os for each data stream.

Comparing the SRR policy, allocating blocks in round-robin fashion among the streams, to SPF, assigning a stream to a file and allocating blocks for the file only on the assigned stream, shows increased scaling benefits of SPF as a result of two key reasons. (1) The decrease in the file fragmentation, and (2) increasing the outstanding write I/Os in a single zone and issuing larger I/O requests. Firstly, if allocation follows round-robin fashion for individual blocks, the resulting data allocation will break all files into 4KiB extents that are spread across the available streams. While this allows to evenly distribute the outstanding write I/Os across the available streams, the increasing fragmentation introduces additional overheads for the file system. msF2FS manages the extents in a red-black tree, where insertions into the tree become a bottleneck as the tree grows exponentially in size.

Secondly, as the workload issues `fsync()` calls every 128 write requests on each file, a single stream may not have 128 outstanding write requests at a time. When `fsync()` is issued, due to consistency reasons, msF2FS and F2FS write all data logs to the storage device. Due to the file data blocks being allocated in round-robin, and files independently flushing write requests, the performance of zones cannot be saturated enough. With SPF, when a job issues a `fsync()` call for its file, it has written all 128 write requests into its hot data log, allowing to maximize the intra-zone throughput. While files are flushing the streams of other files, resulting in possibly not all 128 outstanding requests to be present in a stream when that file issues its `fsync()` call, it allows increasing the number of outstanding in a single zone, which scales better than evenly spreading the I/Os across all available zones.

Summary. The achievable throughput of msF2FS outperforms F2FS by 74.68%, where SPF allocation policy provides the most substantial gains, scaling by 125.29% with the increase of concurrently written hot data files. Furthermore, results showcase the shortcomings of SRR allocation policy incurring substantial file fragmentation, which SPF avoids by mapping files to distinct data streams.

6.4. Performance Impact of Fragmentation

A goal of msF2FS is to decrease the file fragmentation, by providing a larger number of data logs that can be written concurrently for each lifetime classification. With F2FS all files for a single lifetime classification contend to be written on the same data log, resulting in block allocations to interleave the different file data. The file fragmentation results in increased GC overheads, due to the inadequate data grouping. In this section, we evaluate the implications of file fragmentation in F2FS, compared to msF2FS, using micro-workloads generated with fio.

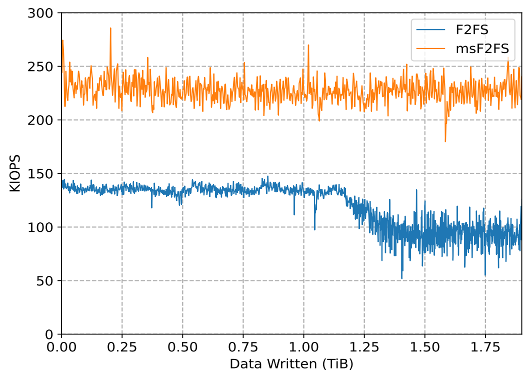
6.4.1. Fragmentation Impact of Deleting Files

In order to measure the GC overhead on write performance, fio is configured to issue 4KiB sequential write I/O requests for each file, with a total of 7 jobs writing their own distinct file. Each file is writing a total of 500GiB, filling 94% of the file system capacity. F2FS and msF2FS allocate blocks only when file data is being written from the page cache to the storage device (i.e., kernel inode writeback or `fsync()` call). Relying on the kernel to periodically flush page cache decreases the fragmentation, as this results in allocating a larger number of consecutive blocks to each file. In order to increase the file fragmentation, fio issues a `fsync()` call for each file (from each job) for every 1 outstanding write request, resulting in file fragments for the 7 files to be interleaved every block. All files are written under the same hot lifetime classification, where msF2FS is configured with 7 hot data streams and the SPF allocation policy, as it has shown to provide the highest performance gains with msF2FS. After writing all file data, over half of the files are deleted (4 out of the 7 files), to generate valid and invalid data in each of the segments, as is a result of the caused fragmentation. The file deletions serve the purpose of freeing space in the file system, however requiring GC to be available for new data writes. To generate GC traffic, fio generates a write workload for 4 new files, issuing 4KiB sequential write requests with the `psync` engine , and calling `fsync()` every 128 requests, as is the configuration of the prior msF2FS throughput benchmark (Section 6.3).

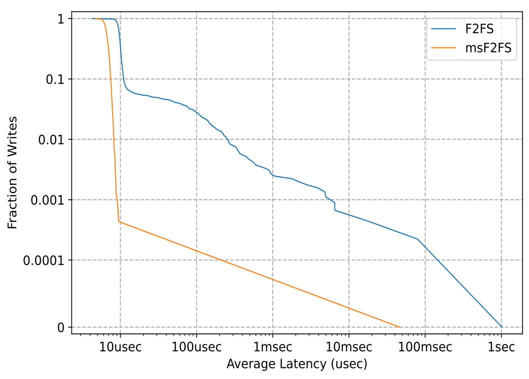
Achievable Throughput. The resulting throughput (y-axis) over the total amount of data written (x-axis) is depicted in in Figure 6.4a, where the throughput depicts the average throughput for 1 second with samples collected every 1 second. The significant drop in performance for F2FS at 1.2TiB written showcases the performance overheads of GC, resulting in a significant drop of average throughput. With msF2FS and SPF allocation policy, each file is mapped to an independent stream, avoiding all file fragmentation and eliminating the need for GC. As a result the performance of msF2FS remains stable throughout the benchmark, without significant drops. The fluctuation of performance in msF2FS is slightly larger, due to jobs issuing `fsync()` calls, resulting in flushing of all data streams, containing data from the other files. msF2FS utilizing concurrent data streams allows having an average throughput 94.77% higher than F2FS.

Average I/O Latency. In addition to the throughput, Figure 6.4b shows the resulting reverse *Cumulative Distribution Function (CDF)* [217]. With a simple CDF, the significance of the tail latency is lost, where focus emphasizes the mean latency [217]. Therefore, with a reverse CDF, the plot depicts the fraction of writes (y-axis) that have a latency higher than the value depicted on the x-axis (on log-scale). Note, due to the sampling of every 1 second, values represent the average latency during the sample duration, and not the latency of individual I/O requests. As a result, the latency between 1 and 0.1, presenting 100% to 10% of the writes, showcases a 3.06 microseconds lower latency for msF2FS. However, the tail latencies for the remaining 10% increase significantly for F2FS, as a result of GC, where the P100 tail latency is one order of magnitude higher for F2FS. High tail latencies in msF2FS are a result of writing checkpoints, as msF2FS utilization of multiple streams enhances the data grouping, eliminating possible GC during this benchmark. The GC triggered in F2FS however overshadows the latency of checkpointing, resulting in latency in the scale of seconds.

Tail Latency. As the logging of latency is sampled for every 1 second, with the latency of all I/O during the sample being averaged, tail latency that exceeds 1 second cannot be sampled. Particularly, as the tail latency of F2FS in Figure 6.4b shows 1 second to be the highest latency, equal to the sampling rate. Therefore, Figure 6.5 furthermore shows the tail latency of individual write I/O requests. Note, these values are the tail latency as reported by fio, measured for each individual I/O request. Results show a further increase for the tail latency beyond the average latency of Figure 6.4b. F2FS reaches a P100 latency of 7.28 seconds, over 2 orders of magnitude higher than the P100 of msF2FS with 0.016 seconds. The tail latency for msF2FS is solely



(a)



(b)

Figure 6.4: Write benchmark showcasing the (a) throughput over the amount of data written with GC being triggered in F2FS, and (b) the reverse CDF depicting the average I/O latency every 1 second.

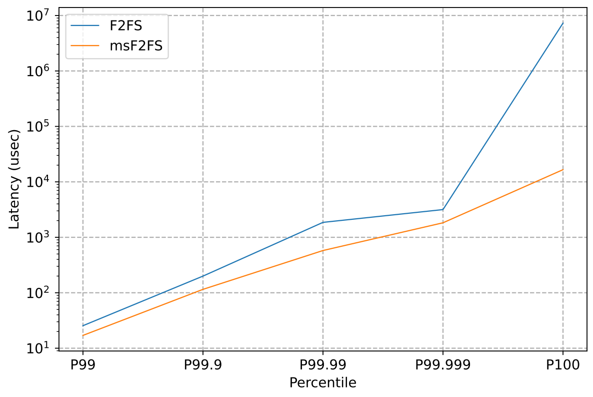


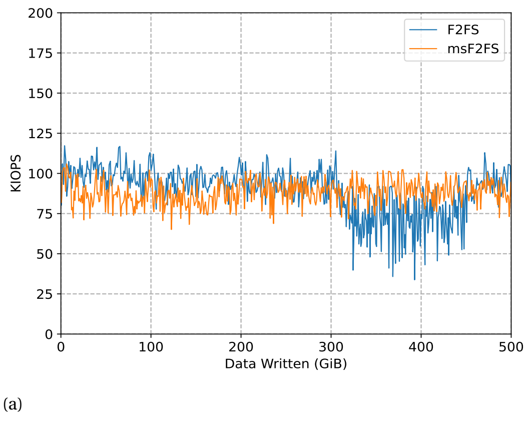
Figure 6.5: Latency percentiles of the write I/O requests during the writing of file data.

a result of checkpoints, whereas the GC overhead of F2FS attributes to its high tail latency.

6.4.2. Fragmentation Impact of Overwriting Files

To evaluate the effects of overwriting existing files, an identical workload is repeated, which instead of deleting 4 files overwrites a single file. A single job selects one file to overwrite with fio issuing 4KiB sequential write I/O requests and issuing `fsync()` calls every 32 requests, as now only a single file is written instead of the previously configured 4 files. Figure 6.6a shows the resulting throughput (y-axis) over the total amount of data overwritten (x-axis). As now both file systems overwrite just a single file, the concurrent data streams in msF2FS provides no performance gains as these are not utilized. However, the resulting throughput showcases the drop in performance for F2FS when it must run GC to free space, whereas msF2FS does not run any GC and maintains stable throughput. Since the workload is configured to write sequentially, msF2FS is invalidating all data blocks as they are being overwritten. Therefore, msF2FS can directly free the space of a section, as these no longer contain valid data once the benchmark has overwritten the data blocks. With F2FS the interleaving of prior written data with the data of the file being overwritten results in valid blocks remaining in the sections, triggering GC.

The reverse CDF for the overwrite workload, depicted in Figure 6.6b, shows the average I/O latency (x-axis), during samples of 1 second, on a log-scale, and the fraction of writes (y-axis) with a higher average latency than the corresponding value on the x-axis. Similarly, F2FS has a 29.87 microsecond higher latency, however due to only overwriting a single file, less GC is required and the tail latency is substantially lower, compared to the prior benchmark. Evaluating the tail latency of individual I/O requests, Figure 6.7 depicts the percentiles of individual requests, showing an order or magnitude higher tail latency with F2FS, as a result of the GC. The decrease in GC run by F2FS, compared to the prior benchmark which writes 4x the amount of data, provides it with lower overall latencies, showing the increasing GC overhead over time, where continued fragmentation significantly decreases performance.



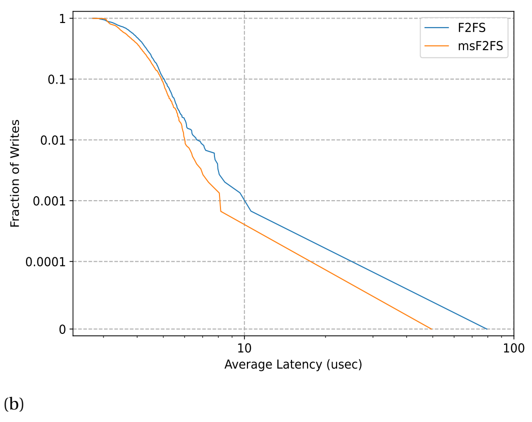


Figure 6.6: Overwrite benchmark for a single file showcasing the (a) throughput over the amount of data written with GC being triggered in F2FS, and (b) the reverse CDF depicting the average I/O latency every 1 second.

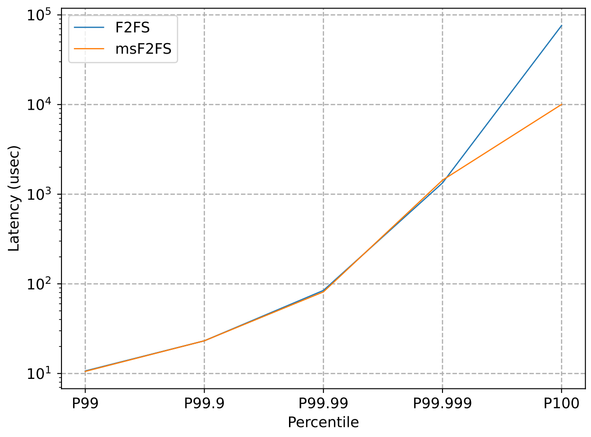


Figure 6.7: Latency percentiles of the write I/O requests during the sequential overwriting of a single file.

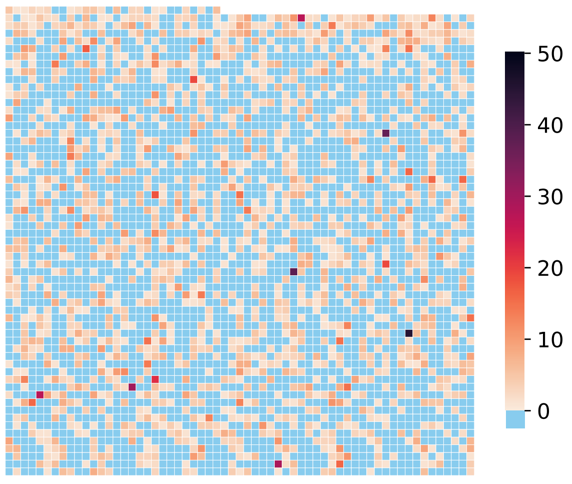
6.4.3. Summary

The increased possibility of data grouping with msF2FS, due to the multiple data streams, reduces the file fragmentation, which in turn avoids GC overheads. As a result, msF2FS is able to achieve 94.77% higher average throughput, and tail latency 2 orders of magnitude lower than F2FS.

6.5. Zone Management Activity

To overwrite the ZNS device, F2FS and msF2FS must first issue zone reset commands to reset zones that no longer contain valid data blocks. The utilization of zone resets is similar to block discard commands (`blkdiscard`) issued on conventional SSDs to discard a particular range of blocks. While there are additional zone management commands, such as finishing of a zone that fills the remaining space, F2FS and msF2FS only utilize the zone reset command, due to the sequential writing in the logs. When a section, mapping to a single ZNS zone, no longer contains any valid data blocks, it can be reset. However, the reset must be issued after the next checkpoint, such that in the case of recovery, the prior checkpoint points to the old data, causing it to become valid again. As zone resets have a significant performance impact with latency of ~10 milliseconds [63], zone reset commands should be minimized to avoid excessive zone reset overheads.

The on-device implementation of zone resets commonly only updates metadata to indicate the LBAs associated to the zone are no longer valid. However, the device must unmap the LBAs from the mapping tables, resulting in the zone reset overhead. In addition to introduced zone reset latency, depending on the ZNS firmware, if the device does not implement wear leveling, relying on a static zone-to-flash mapping that always maps the LBAs in a zone to the same PBAs, even wear across the zones must be ensured by the host. For reference, the device used during this evaluation does not utilize a static mapping and handles the wear leveling, however this remains a device specific implementation.



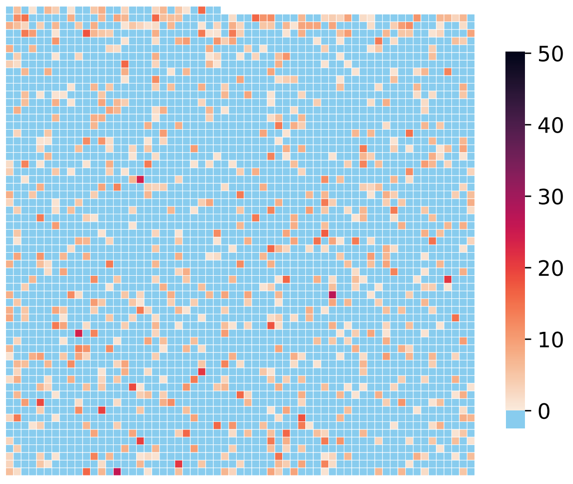


Figure 6.8: Heat maps indicating the zone reset commands issued to each zone on the ZNS device by (a) F2FS and (b) msF2FS. A blue square indicates a zone to which 0 reset commands were issued.

To quantify the zone management of msF2FS compared to F2FS, fio is configured to generate a workload that fills 98% of the available space on the file system with 7 hot data files, each 520GiB in size, followed by sequentially overwriting 3 of the files for a total of 3 times, resulting in a total of 4.57TiB overwrite traffic. The writing and overwriting is configured to issue 4KiB write I/Os to each file by an independent job, with each job issuing a `fsync()` call for every 32 request. Therefore, firstly filling the file system, followed by overwriting existing data generates GC in F2FS, as it co-locates the data blocks of all files on the same log, where the overwrite benchmark invalidates the old data. Consequently, during the overwrite workload F2FS co-locates the new data blocks of all files being overwritten, resulting in continuous GC. msF2FS is configured with the SPF stream allocation policy and 7 hot data streams, mapping each file to a distinct stream, eliminating file fragmentation and GC.

Tracing of zone management operations is achieved by utilizing the `zns.trace` tool from the `zns-tools`, which traces the zone reset commands issued by to the ZNS device. `zns.trace` collects the counter of the zone reset commands issued over the entire write and overwrite benchmark duration. Figure 6.8 shows the resulting heat map of the zone resets issued to each zone for (a) F2FS and (b) msF2FS. Zones are represented in increasing numbers by each row, starting at point (0,0), and a blue square indicates the zone has 0 reset commands issued to it. The GC in F2FS being triggered during the overwrite workload results in a more uniform distribution across the available zones, as it selects the zone with the most garbage (greedy GC policy), freeing up the most amount of space. As a result, a larger number of zones are reset only few times.

While there are zones that are reset several times, these are the result of greedy GC selecting the optimal choice. A uniform distribution can be achieved with the *Age-Threshold (AT)* GC policy (background GC), that selects the best *Cost-Benefit (CB)* option, with a minimum time requirement to have elapsed in order to be selected as the victim, eliminating the possibility to continuously select the same zone. With msF2FS eliminating all GC in the workload scenario, it resets a smaller number of zones more times. 14.34% of the zones that are reset by msF2FS are reset 10 or more times, whereas only 5.48% of the zones reset by F2FS are reset 10 or more times. Particularly as it avoids fragmentation, the data blocks of the file being overwritten are invalidated sequentially, resulting in the zone not containing more data, and resetting it. Therefore, the iterations of overwriting result in the same zones becoming available. While msF2FS provides a decrease in wear leveling, it is able to reduce the total number of zone resets issued by F2FS from 6,499 to 4,454, a reduction of 31.04%.

6.5.1. msF2FS Carbon Footprint

The manufacturing of flash storage generates CO2 emissions [278], requiring the file systems to optimize the flash usage. Based on the prior conducted zone management tracing benchmark, identifying the number of reset commands issued by F2FS and msF2FS, we estimate the carbon footprint for msF2FS and F2FS. Firstly, with the total number of zones on the ZNS device (3,688), and its endurance, being written 3.5 times a day for a total of 5 years (retrieved from the specification sheet of the ZNS device [61]), the total number of resets that can be issued is given as.

$$\text{Total\_Resest} = 3,688 * 3.5 * 365 * 5 = 23,557,100$$

Next, the collected data on the number of issued reset commands during the benchmark is used to calculate the number of times the benchmark can be repeated.

$$\text{F2FS\_Bench\_Cycles} = 23,557,100/6499 = 3,624 \text{ cycles}$$
$$\text{msF2FS\_Bench\_Cycles} = 23,557,100/4454 = 5,288 \text{ cycles}$$

Using the total amount of data written by the benchmark, which writes 7 files of 520GiB each followed by overwriting 3 files fully for 3 times, giving a total GiB per benchmark cycle of $7 * 520\text{GiB} + 3 * 3 * 520\text{GiB} = 8,320\text{GiB}$, the total GiB of data written over all benchmark cycles is given as.

$$\text{F2FS\_data\_written} = 3,624 * 8,320 = 30,151,680 \text{ GiB}$$
$$\text{msF2FS\_data\_written} = 5,288 * 8,320 = 43,996,160 \text{ GiB}$$

Next, we calculate the number of times each 1GiB of flash storage is written during all benchmark cycles. For this the prior calculated total GiB of data written is divided by the total capacity of the ZNS device. The *zone_cap* depicts the zone capacity of the ZNS device (1,077MiB, note the unit is converted to GiB in the calculation by dividing by 1,024) and the *nr_zones* as the number of zones on the device (3,688).

$$\begin{aligned}
\text{F2FS\_flash\_writes} &= 30,151,680/(zone\_cap * nr\_zones) \\
&= 30,151,680 \ (1077\text{MiB} * 3688/1024) \\
&= 7773.28974797 \text{ times} \\
\text{msF2FS\_flash\_writes} &= 43,996,160/(zone\_cap * nr\_zones) \\
&= 43,996,160 \ (1077\text{MiB} * 3688/1024) \\
&= 11342.4823916 \text{ times}
\end{aligned}$$

With the cost of producing 1GiB of flash storage being 0.16KG of CO2 [278], and the number of times each 1GiB of flash is written, the carbon footprint of F2FS and msF2FS are calculate as follows.

$$\begin{aligned}
\text{F2FS\_CO2} &= 0.16\text{KG CO2 per 1GiB}/7773.28974797 \\
&= 205833 * 10^{-10} \text{ KG of CO2 per 1GiB} \\
\text{msF2FS\_CO2} &= 0.16\text{KG CO2 per 1GiB}11342.4823916 \\
&= 141062.5 * 10^{-10} \text{ KG of CO2 per 1GiB}
\end{aligned}$$

As a result, msF2FS has a carbon footprint, which is 31.47% lower than F2FS. The reduction in fragmentation of msF2FS eliminating GC scales its longevity by reducing the amount of data written, decreasing the consumption of flash storage, and in turn leading to reduced CO2 emission for flash manufacturing costs.

6.5.2. Summary

The reduction in fragmentation, resulting in a decrease of GC, decreases the required zone reset commands by 31.04%, which furthermore benefits msF2FS in prolonging the longevity of the flash storage. Increasing the longevity in turn provides a reduced carbon footprint, as a result of flash storage manufacturing costs, where msF2FS has 31.47% lower CO2 emissions, compared to F2FS.

6.6. Macro-Level Impact of msF2FS Streams and Application-Guided Data Placement

To measure the performance of msF2FS with macro-workloads, we utilize RocksDB [73] and its benchmarking tool db_bench. RocksDB is an optimized key-value store for flash-based storage devices. Its design characteristic is based on the LSM-Tree data structure, illustrated in Figure 6.9, which is organized in a number of *Sorted String Table (SSTable)* files at particular levels. Upon a write operation within RocksDB, keys are written into an in-memory *memtable*, which is flushed to persistent storage when full or periodically. Written to

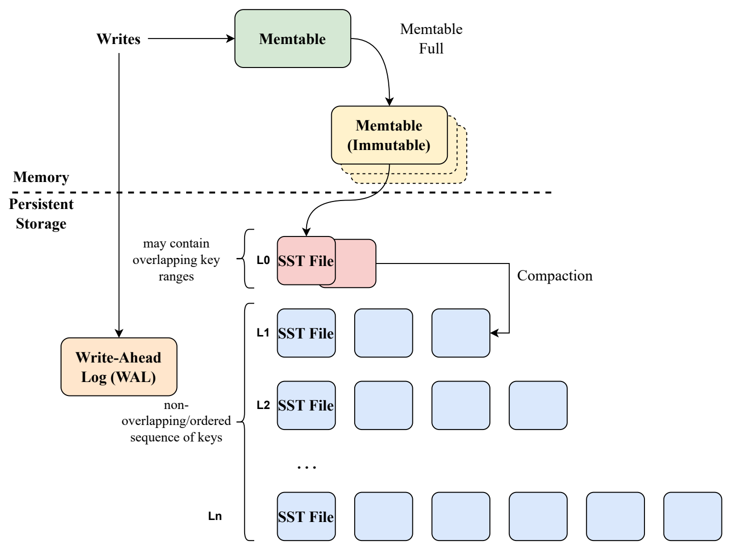


Figure 6.9: Illustration of RocksDB files maintained in memory, and flushed to persistent storage through *compaction* with level-based SSD files. Figure adapted from RocksDB Wiki [74].

level 0 (abbreviated as L0), the SSTable file may contain overlapping key ranges at this level. All SSTable files are stored on persistent storage (e.g., flash SSD), whereas the *memtable* resides in memory. When the SSTable files at L0 are full, particular files are selected to be *compacted* to the next level (i.e., L1 from L0). The process of compaction combines two SSTable files and generates a new SSTable file with the merged contents. Starting from L1, files contain non-overlapping keys and the keys are ordered. With the increase in level, the size of the SSTable files similarly increases. For failure tolerance, RocksDB utilizes a *Write-Ahead Log (WAL)*, to write each update into. A new WAL file is created after each compaction, such that recovery is possible based on the information in the WAL.

The particular characteristic of RocksDB that is optimized for flash storage is the utilization of immutable SSTable files. These are written sequentially, are immutable, and are erased in one unit, matching the characteristics of flash storage. For the remainder of this evaluation, we modify the mapping of WAL and SSTable files to msF2FS streams by integrating the msF2FS application-guided data placement hint mechanisms. Therefore, a high level understanding of RocksDB levels and files for SSTable and WAL is required.

Particularly the choice for RocksDB as the case study for msF2FS hint integration is due to its write characteristics of sequentially writing in an LSM-Tree. Its characteristics match the characteristics of flash storage and allow evaluating the hint integration of msF2FS, in addition to evaluating the impact of the increased parallelism and reduced fragmentation in msF2FS. Firstly, in Section 6.6.1 we describe the modifications required in RocksDB to integrate the two msF2FS hints, exclusive streams on SPF and full stream management by RocksDB with AMFS stream allocation policy. Next, the evaluation is conducted for the msF2FS integration in RocksDB with a series of benchmarks in Sections 6.6.3 and 6.6.4. In order to depict the designed stream mappings of the RocksDB integration with AMFS, we firstly explain the setup of msF2FS for all RocksDB benchmarks.

File classification in RocksDB are as follows; (1) the WAL is classified as hot data, (2) the L0 - L1 SSTable files are classified as warm, and the remaining levels are classified as cold. Based on the write characteristics and lifetime classifications, msF2FS is configured with 2 hot data streams, 3 warm data streams, and 4 cold data streams. The 2 hot data streams serve the purpose to be able to map the WAL files and additional frequently written files, such as the `LOG` file. As the L0 and L1 SSTable files are most frequently written, and are the smallest, utilizing 3 different warm data streams aims to concurrently write the different files, and to adequately provide data separation such that fragmentation can be minimized. Lastly, the cold data streams configuration is based on the remaining available streams on the device, which are not fully utilize in order to avoid SSAB performance overheads and its resulting suboptimal data mappings. Therefore, 2 zones are left without streams, and configure the cold data streams to 4, providing a total utilization of 12 streams (9 data streams and 3 default node streams). As there are the most files for different levels in cold data, the 4 cold data streams aim to provide data separation of the L2 and higher SSTable files. A broader exploration of hotness-level streams to zones is left as future work, as we use a 2/3/4 (hot/warm/cold) ratio for the streams

Hint Integration	Lines of Code
SPF exclusive streams	46
AMFS-1	88
AMFS-2	87
Total	221

Table 6.3: Lines of code required for integrating the respective hinting policy into RocksDB.

configuration.

6.6.1. RocksDB msF2FS Hint Integration

To integrate the added hint mechanisms of msF2FS into RocksDB, we firstly define the mapping of files to establish. Integration follows the practices of existing RocksDB enhancements, such as presented by SpanDB [38], which separate the WAL and top low-level RocksDB LSM-Tree files (L0) from the higher-level files (L1+). msF2FS hint integration is achieved with a similar mapping for the SPF exclusive stream hint and the AMFS stream allocation control. For SPF exclusive streams we evaluate one integration (Section 6.6.1.1), and for AMFS we evaluate two separate integrations utilizing the stream mapping differently (Section 6.6.1.2), to which we refer to as AMFS-1 and AMFS-2 throughput this evaluation. Table 6.3 indicates the required modifications for integrating the two hinting policies, with the lines of code added to RocksDB. We make all source code changes for the hint integration publicly available at `https://github.com/nicktehrany/rocksdb`, on the `amfs` and `amfs-2` branches for the AMFS policy (for AMFS-1 and AMFS-2 respectively), and the `spf_exclusive_streams` branch for the SPF policy with hints.

SPF Exclusive Streams

The goal of the first integration is to separate only the WAL files by requesting an exclusive stream for the WAL files. Based on the SPF policy, the remaining files are mapped to the remaining streams in round-robin fashion. As the exclusive stream hint with SPF requires to be set and unset during block allocations, the inode flag of the respective file must be cleared to release the exclusive stream. This becomes particularly challenging with asynchronous I/O and buffered I/O, as the application does not directly know when the last block allocation occurs, in order to unset the inode flag at the appropriate time. The unsetting of the inode flag occurs during the period where RocksDB synchronizes the outstanding write requests. While this is not optimal, as this may release the exclusive stream before all blocks have been allocated on the stream, it decreases the possible fragmentation caused by files being mapped into the WAL files before the synchronization call by RocksDB.

Application Managed File Streams

AMFS-1. The first AMFS integration aims at statically separating WAL and SSTable files by assigning the WAL to a distinct stream, and mapping each file to a particular stream based on its level. Table 6.4 provides the lifetime classification and mapping of each SSTable level. As there are 2 hot data streams, the WAL is mapped to the last stream, in order to avoid other hot data files to be mapped into the same stream as the WAL. Similarly, L0 is mapped to the last warm data stream in order to avoid files that have no lifetime classification, which are allocated on the warm data log stream 0, to fragment the L0 files. Therefore, by mapping only L0 files to the last warm data stream, the L0 file is not fragmented by any other file. The remaining warm data streams are mapped to L1 and L2. Remaining levels after L2 are mapped to the cold data stream, where the highest level L6 is mapped to the cold data stream 0. This particularly aims at co-locating the coldest SSTable files with the data that is moved during GC. As the GC process reclassifies moved data to cold data, and writes the data blocks on cold data stream 0. By grouping it with the data of L6 files, the coldest data SSTable files reside are grouped with the cold data moved by GC. L3-L5 are mapped to the remaining cold data streams, with each occupying an independent stream.

The benefit of such a mapping allows to be deterministic, as opposed to SPF possibly not applying deterministic mappings. If files are flushed from page cache in different order, the round-robin based stream allocation, pinning a file to a stream, depends on the exact order of the flushed files. This can vary depending on available memory and the writeback calls issued by the kernel. Therefore, utilizing AMFS, deterministically ensures that files of particular levels are always mapped to the same stream, and are co-located with files from the same level, providing lifetime-aware file classification.

File/Level	Lifetime Classification	Stream
WAL	HOT	Stream 1
L0	WARM	Stream 2
L1	WARM	Stream 0
L2	WARM	Stream 1
L3	COLD	Stream 1
L4	COLD	Stream 2
L5	COLD	Stream 3
L6	COLD	Stream 0

Table 6.4: Static stream mapping of RocksDB files for msF2FS configured with AMFS-1 hint integration.

AMFS-2. The second integration of AMFS aims at utilizing the existing hinting mechanism within RocksDB, based on which it assigns lifetime classifications to SSTable files, and avoid static stream assignments by utilizing compaction information in the stream assignment. The default mapping of RocksDB is for the WAL to be classified as hot data, L0 to be warm data, and depending on the `base_level` higher level classifications occur. The base level depicts the level that the L0 SSTable file is being compacted to. Therefore, (1) if the difference between the current level and the `base_level` is greater than 2, the data is classified is cold. Otherwise, (2) if the current level is smaller than the `base_level`, it is classified as warm data. To integrate the AMFS stream allocation mapping, the WAL is mapped to the hot data stream 1, leaving hot data stream 0 for other hot data files to not fragment the WAL. L0 is statically mapped to the warm data stream 2, separating it from all other warm data files, where further streams are assigned based on the `base_level`. Based on the `base_level` being compacted from, higher levels are assigned to respective streams of their lifetime classification.

6.6.2. RocksDB Benchmark Configuration

Evaluating the performance of RocksDB on msF2FS with the additional hint integration, is achieved with RocksDB workloads generated by its `db_bench` benchmark, which is configured using the `zbdbench` benchmarking suite [53], presented in [20]. The suite contains a benchmark that evaluates the performance of numerous RocksDB workloads on file systems, particularly aimed at ZNS devices and the ZenFS backend for RocksDB on ZNS devices. ZenFS provides a file system backend that directly ZNS device for directly writing RocksDB generated files to the zones, without additional intermediary file systems. Utilizing a *Lifetime-based Zone Allocation (LIZA)* policy for file allocations to zones, it provides data grouping based on the lifetime hints that RocksDB sets for files. To distinguish the different msF2FS configurations, throughout this section we utilize the following abbreviations for their configuration; (1) msF2FS without utilization of hints (abbreviated as SPF), SPF with the exclusive stream hint integration (abbreviated as SPF-E), and msF2FS with AMFS (abbreviated as AMFS-X, with a number indicative of the version of integration).

Before configuring the `zbdbench` framework to evaluate the performance of msF2FS, we firstly discuss the default RocksDB workload, as presented in the original paper [20], followed by the modifications made to integrate the msF2FS evaluation for this thesis. The suite defines a workload that firstly fills (*fillrandom* benchmark) the entire database with 3.8B keys, each of which is 20B in size with an 800B value that is compressed down to 400B. The configuration of the SSTable files is aligned to the zone size (2GiB), with the memtable similarly configured to match the zone size. As the actual SSTable file size can vary slightly, depending on the compression and contents, the file size is 1.99GiB, to allow for variations in the file size to fit within the zone size. This size is specific for L0 SSTable files, where higher levels apply this base size with a multiplier of 4. This serves the purpose of aligning the larger higher level SSTable files, and reduce the overlap of SSTable files during compaction to maintain write activity for numerous SSTable files. The workload furthermore enables concurrent compactions, and deletes obsolete files every 30 seconds. After filling the database with the *fillrandom* workload, an *overwrite* workload is run for the same number of entries, with identical configuration, followed by a random reading workload (*readrandom* benchmark), a read while writing workload (*readwhilewriting* benchmark), running 32 read threads with 1 thread issuing write requests. Lastly, a workload that repeats the read while writing workload is run, with the write thread being limited to 20MiB/second. The latter two benchmarks allow evaluating the concurrency capabilities of RocksDB and the file systems, where ongoing GC caused by the write thread trigger possible read delays. All the reading workloads are configured to run for a total of 30 minutes.

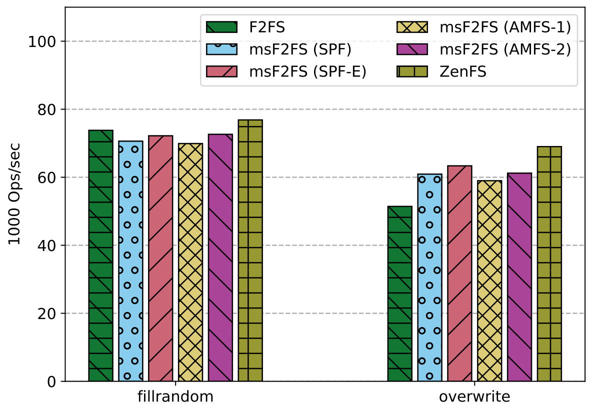


Figure 6.10: Write throughput by the different file system configurations for the (a) *fillrandom* workload, and (b) the *overwrite* workload.

In order to scale the workload with the ZNS device used during this evaluation, we make two main modifications. Firstly, the suite creates a `nullblk` device for the F2FS metadata, which is 250GiB in size. We reduce this size to the prior established 19GiB (Section 6.1), to avoid data segments residing on the `nullblk` device, and only utilizing it for metadata. Secondly, we increase the total number of keys written to 6B, to scale the workload with the larger size of the used ZNS device. Furthermore, we increase the available memory of the QEMU VM from 25GiB to 64GiB, as RocksDB requires substantially more memory than the micro-workloads. The particular choice for 64GiB aims to provide enough memory for RocksDB to run the workload successfully, and maintain enough memory pressure the increase the flushing of file data, and therefore result in possible larger fragmentation. The available memory to RocksDB is additionally limited by the 19GiB required for the `nullblk` device. As described in Section 6.6.1, the msF2FS configuration utilizes 2 hot data streams, 3 warm data streams, and 4 cold data streams.

6.6.3. RocksDB Write Performance on msF2FS

Throughout this section we evaluate the performance of the write workloads for RocksDB (*fillrandom* and *overwrite*), followed by the respective read workloads in Section 6.6.4

Write Throughput. Firstly, we evaluate the write throughput of both write workloads. Figure 6.10 depicts the throughput of the *fillrandom* and *overwrite* benchmarks. The write throughput of F2FS for *fillrandom* is up to 5.26% higher than msF2FS, as a result of F2FS being able to merge more requests to the single data log. With msF2FS the requests are split up across more available zones, resulting in less possible merging of I/O requests. While all msF2FS configurations achieve slightly lower throughput than F2FS for *fillrandom*, the hint integrations SPF-E and AMFS-2 scale the *fillrandom* throughput within 97.93% of F2FS, showcasing the benefit of the application managing the stream allocation. Particularly, the benefits of hint integration outperform the default msF2FS stream allocation policy by 2.84% for *fillrandom*. With ZenFS being a custom built file system for RocksDB and ZNS device, it provides superior performance over F2FS and msF2FS, outperforming msF2FS by 9.93%, and F2FS by 4.14%.

The *overwrite* benchmark shows a significant decrease for F2FS in throughput of 30.31%, whereas the throughput drop for msF2FS ranges between 12.24% (SPF-E) and 15.71% (AMFS-2) for the various integrations. The substantial performance drop for F2FS is a result of GC activity arising from the file fragmentation. With msF2FS the enhanced data mapping reduces the required GC overheads by limiting the fragmentation, allowing it to maintain higher throughput. Specifically, the benefit of the application-guided data placement with hints, allows throughput to scale up to 23.19% higher than F2FS. As a result, msF2FS with hint integration is capable of scaling within 91.08% of the custom ZenFS backend.

Tail Latency. Evaluating the tail latency of the write benchmarks, shown in Figure 6.11a, F2FS achieves lower P95 and P99 during the *fillrandom* benchmark, due to its increase in merging, whereas msF2FS without hints, and the suboptimal AMFS-1 integration result in higher tail latency. However, the hint integration of AMFS-2 decreases the tail latency to that of F2FS. With ZenFS achieving the highest write throughput, it similarly provides the lowest tail latency, with a P95 of up to 49.47% lower than msF2FS, and a P99 of up to 44.17% lower than msF2FS. The penalty of increased fragmentation and GC, resulting in the significant throughput loss for F2FS on the *overwrite* benchmark, depicted in Figure 6.11b, is similarly evident in the tail latency, where the P95 and P99 are 11.84% and 17.16% higher than msF2FS.

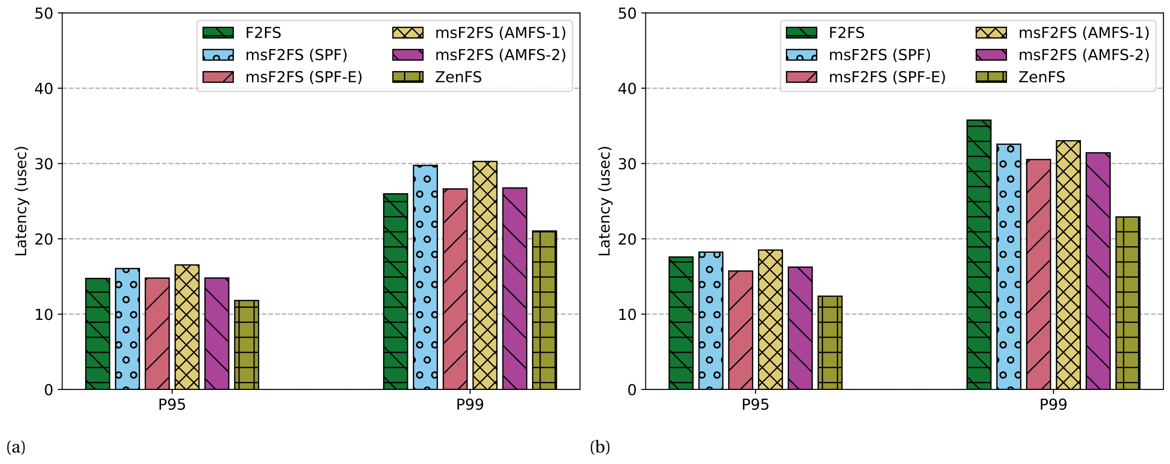


Figure 6.11: Tail latency of RocksDB writes during the (a) *fillrandom* workload, and (b) *overwrite* workload.

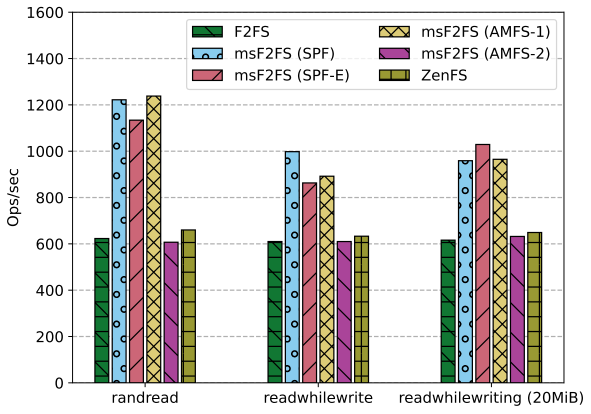


Figure 6.12: Achievable throughput of RocksDB random read operations during the various read workloads, with *readrandom*, *readrandom* and writes, and *readrandom* with writes being limited to 20MiB/s.

Summary. With the RocksDB workload generating a substantial amount of files, the reduction of fragmentation and GC overheads in msF2FS with the aid of application hints achieves a 23.19% higher overwrite throughput than F2FS, scaling within 91.08% of ZenFS. The GC overheads similarly result in F2FS having a 17.16% higher P99 tail latency than msF2FS.

6.6.4. RocksDB Read Performance on msF2FS

Following the *fillrandom* and *overwrite* workloads, `zbdbench` issues several read workloads, which we discuss now.

Read Throughput. Firstly, we evaluate the achieved throughput of the various read workloads, depicted in Figure 6.12. The overall throughput of all benchmarks is substantially lower than that of the write workloads, which is a result of the high memory pressure, particularly as all benchmarks utilize random reading operations, reading of SSTable files becomes particularly slow with the limited memory availability. The resulting performance showcases that msF2FS is capable of achieving up to 98.72% higher throughput than F2FS for random reading, and up to 67.05% for random reading while writing workloads. Furthermore, msF2FS similarly outperforms ZenFS by 87.58% for reading and 58.55% for read while writing workloads. However, given that AMFS-2 hint integration performs similar to F2FS and ZenFS, and that random read operations should not be heavily affected by fragmentation in the file system, we are still investigating the reason for the higher performance of msF2FS.

Summary. While msF2FS is capable of outperforming F2FS and ZenFS by nearly 2x the read performance for particular msF2FS configuration and hint integration, the performance of AMFS-2 hint integration fails to reach such performance, begging the question if the high memory pressure applied is a result of the decreased read performance, or are a result of the file system. Investigation into the implications of the random read results is ongoing.

6.6.5. Summary

The reduction of fragmentation in msF2FS, resulting in a reduction in GC overheads, and the integration of msF2FS hints for application-guided data placement, result in a 23.19% higher overwrite throughput for msF2FS, compared to F2FS. Furthermore, msF2FS is capable of scaling within 91.08% of custom RocksDB file system backend ZenFS. Similarly, P99 tail latency for F2FS is up to 17.16% higher than msF2FS as a result of the increased GC.

7 Related Work

With ZNS having only been introduced and standardized in 2021 [20, 301], there are only a limited number of integrations into storage system software. Therefore, we divide the fields of related work into respective groups, depicting ZNS-specific integrations and flash-based storage evaluations, file system (particularly F2FS) optimized flash-integration, and additional application enhancements. Much of the related work is furthermore discussed in the presented literature study (Chapter 3).

ZNS Integration. Bjørling et al. [20] pave the way of ZNS integration with the proposals of the ZNS interface, and providing its first integration into storage software with F2FS and ZenFS for RocksDB. The modifications the authors make are merged into the mainline Linux kernel, and are the foundation for ZNS support on which we base the design and implementation of msF2FS. With ZNS attracting significant research attention, Purundare et al. [239] discuss the importance of Log-based data structures on ZNS with file systems, databases, and key-value stores. The authors particularly focus on the newly introduced append operation for ZNS, leveraging grouped append operations that buffer small I/Os (few bytes in size) in device memory before appending to a zone, in order to enhance micro write performance caused by small I/O, such as metadata updates. The benefits of ZNS devices are furthermore discussed by Stavrinos et al. [268], arguing the standardization of ZNS as a fundamental shift to the storage paradigm, making conventional SSD research obsolete.

With ZNS providing a new interface for flash management, current integration in system software is limited. ZNSwap [13] implements swap management on ZNS with host-side garbage collection. Another ZNS based storage application is the development of TropoDB [64], providing a LSM-Tree based key-value store for ZNS with particular *Write-Ahead Log (WAL)* integration with ZNS append operations. Similarly, Lee et al. [169] enhance the RocksDB integration with ZNS by utilizing the available zone mapping in death-time based data grouping for SSTable files, with compaction-aware zone allocation. The drawbacks of unaligned SSTable allocation on ZNS zones has similarly been discussed by Jung and Shin [130], showcasing the increased fragmentation and resulting delay in zone reset, requiring effective lifetime-based SSTable grouping. The mechanisms of data grouping, commonly focusing on decreased fragmentation are similarly a fundamental goal of our msF2FS design. The LFS-based data management is furthermore extended by Choi et al. [42], who identify that the possibly large size (seveal GiB) of ZNS zones are suboptimal for GC, for which the authors propose a segment-based (several MiB in size) LFS data management and GC policy for ZNS.

Flash-Based File System Optimizations. While there are a plethora of file systems that provide optimized integration for flash-based storage (discussed in the literature study presented in Chapter 3), there are numerous file system optimizations that are particularly related to the work we present with ZNS. While not being based on ZNS integration, Lia et al. [185] present Max (multicore-accelerated file system), that enhances F2FS to scale with the parallelism capabilities of flash SSD by scaling with the number of cores available on the system. Particularly, the authors modify F2FS to enhance the locking granularity of in-memory file system data structures, eliminating severe concurrency bottlenecks, and present the *Minor Log (mlog)* abstraction, dividing F2FS logs multiple concurrently writable logs. The implementation of the mlogs are similar to msF2FS, where we however map the different logs to different F2FS sections for better data grouping. Similar to the new *stream* abstraction we present for F2FS, Rho et al. [246] provide a related implementation of files being mapped to distinct resources with streams. Their implementation adapts the ext4 and xfs file systems to allow mapping of different data to separate streams, including the mapping of particular files to particular streams, and certain metadata (e.g., inode) to specific streams. Therefore, the proposed system is very much alike msF2FS, where we however furthermore focus on leveraging hardware parallelism capabilities, and enhance data placement with application-guided data placement.

Of particular importance to optimizing integration of flash-based storage are effective data grouping mechanisms allow reducing GC overheads of flash storage and LFS systems [32, 98, 187, 259, 313] (see Section 3.6.2 for detailed discussion on data grouping mechanisms). Such data grouping can furthermore be enhanced by improved coordination in the storage software stack layers on data placement decisions. Qiu et al. [242] present such integration with coordinated GC between FTL and the file system, which is similarly implemented in several other file systems [302, 320]. Other efforts in reducing the semantic gap with flash-based storage aim at adapting the block interface to provide a higher degree of flash management to the host [174, 175], much alike the ZNS interface.

I/O Hint Integrations. The utilization of passing hints to enhance data grouping on the storage device has previously been presented by numerous efforts. FStream [246] utilizes multi-stream SSD to pass hints to the storage device, indicating data lifetime and enhance data grouping. Similarly, Ge et al. [82] provide a framework with an application interface to pass hints on the data classification, I/O scheduling, and prefetching to the storage device. The application of utilizing hints to storage devices has similarly been largely adopted by related work. Mandal et al. [200] present an interface for hint passing to enhance prefetching and reduce deduplication of data. While msF2FS aims at utilizing hints for enhanced data grouping, its stream management and hinting interface can be extended to support similar prefetching and deduplication hints.

8 Conclusion and Future Work

In this thesis we take the unique opportunity of enhancing the ZNS integration, and present msF2FS, a ZNS optimized file system. The goal of msF2FS is to leverage the capabilities of the newly standardized ZNS interface to increase (1) the parallelism capabilities of F2FS, (2) enhance the data grouping to reduce GC overheads, and (3) shrink the semantic gap between the file system and applications with **application-guided data placement**. Through the introduced **stream** abstraction, msF2FS maps concurrently written logs to different ZNS zones, providing increased utilization of the ZNS zones, and expanding the data grouping possibilities. Data grouping with msF2FS is furthermore enhanced by coordinating the stream management with applications through hint passing.

8.1. Answering Research Questions

RQ1: How has flash storage influenced design choices, challenges, and opportunities in file systems development?
The architecture of flash-based storage introduces several key challenges for file system and storage software developers to account for in the design and development of storage software. A particularly important challenge is the management of garbage collection overheads. The penalties of GC furthermore extend into increased I/O amplification and wear of the flash storage. A multitude of effective algorithms and data structures have been integrated into file systems, in an effort to reduce and manage the GC overheads with flash storage. Apart from the challenges arising from flash storage, the parallelism capabilities that flash storage provides present a great opportunity for file systems to leverage. A plethora of methods, including data striping, clustered allocation, and high concurrency are widely adopted in the design of file systems to leverage the potentials of flash storage.

RQ2: How to identify and leverage application-provided hints to improve performance of flash file systems for data management?
The opportunity presented by the ZNS zone interface, gives file systems the capability to ensure data separation with the zone interface. With msF2FS, we present **application-guided data placement**, where the data placement decision-making is coordinated between the file system and the application. The coordination relies on hints, containing an application provided bitmap on the file-to-stream mapping in msF2FS, to be passed from the application to msF2FS. Using the hint passed from the application, msF2FS is able to leverage the bitmap from the hint information in the allocation of the file data, enhancing the data grouping based on the knowledge conveyed from the application.

RQ3: What is the quantitative impact of leveraging a larger number of concurrent zones in msF2FS with multiple streams?
The ZNS interface offers a unique opportunity to leverage the parallelism capabilities through its *zone* abstraction. In Chapter 5, we detail the design and implementation of msF2FS, introducing the **stream** abstraction, mapping each concurrently writable log to a distinct ZNS zone. The concurrently writable streams of msF2FS outperform the single data log in F2FS, utilizing a single zone, by up to 74.56%, showcasing the benefits of msF2FS leveraging the parallelism capabilities of ZNS. Furthermore, the scalability of msF2FS utilizing

a larger number of zones results in a 125.29% throughput gain, compared to utilizing a single zone. The increased data grouping capabilities of msF2FS reduces the necessary zone rest commands issued by 31.04%, decreasing the overall amount of data written by GC, furthermore decreasing the msF2FS CO2 footprint, arising from flash manufacturing costs, by 31.47% compared to F2FS.

RQ4: How to simplify and visualize operational data in the context of F2FS and ZNS devices as currently there do not exist any tools?
The complexity of F2FS makes understanding its operational aspects a challenging task. Furthermore, there currently is a lack of tools and frameworks to collect and analyze operational data of F2FS. Therefore, with the arrival of ZNS, we leverage the opportunity to provide a set of tools that collect operational data of F2FS and the ZNS device utilization. Using the collected data, several reports and visuals on the data placement of F2FS and the I/O activity of the ZNS device are generated. The ZNS zone interface simplifies the reports by providing mapping information on data placement, and I/O activity, on the basis of individual zones. Therefore, visualizations are represented in comprehensible heat maps, showcasing the activity of individual zones.

8.2. Future Work

With the several contributions of this thesis work, there exists a broad spectrum of possible future work. We divide this work into respective categories, representative of the contributions of this thesis.

msF2FS. Our efforts in enhancing ZNS integration in F2FS pave the foundation for future additions and improved ZNS utilization. Firstly, we propose the addition of full recovery of streams and data, which currently is not supported, by extending checkpointing to persist stream information, and the addition of the application-provided inode flags to be persisted to the on-device inode. Currently, all this information resides in the VFS inode, which is lost during system reboot. Secondly, we suggest extending the stream allocation policy to adaptively allocate streams based on workload and file characteristics, such as the size, file extension, file owner, and file creation time. We furthermore suggest the addition of including more hints in the stream allocation, such as not mixing particular files in streams based on the file extension and the update type flag, indicating if it is randomly or sequentially accessed. Fourthly, we recommend the exploration of dynamic stream allocation and deallocation based on workload characteristics, such as periodic hot-data write heavy traffic. Such policy can be based on application-provided hints when it requires more focus on a particular temperature classification, or can be dynamically determined by the file system. Lastly, the penalty of foreground GC can present a significant bottleneck due to the global locking structure within F2FS, which we propose to enhance with fine-grained locking and multi-streamed GC that concurrently writes out garbage collected data into multiple streams, enhancing throughput. Similarly, we envision integration of file/stream-aware GC, to maintain file data locality and reduce fragmentation, to provide substantial benefits in reducing foreground GC overheads.

F2FS. In addition to msF2FS, modifications to the base F2FS file system are needed to further enhance its ZNS integration. Any of the future additions we propose will also be applicable to msF2FS, however are not focused on the introduced concepts of msF2FS. Any such additions will also be ported into msF2FS. We suggest the addition of direct I/O with ZNS, where currently only buffered I/O is supported. This is achievable by modifying the I/O architecture in F2FS to utilize appends instead of regular writes, allowing to maintain compliance with the sequential write requirement of LFS and ZNS. However, such an addition requires significant refactoring of the I/O architecture in F2FS and is therefore a larger undertaking.

zns-tools. A particular bottleneck of the current implementation of several `zns-tools` is that they are single threaded. Retrieval of extents for multiple files is a sequential approach that can be parallelized by a single thread for each file, or multiple threads in the case of larger files. The current implementation suffers from high overheads for mapping of large files that occupy tens to hundreds of zones, and particularly files that are severely fragmented with millions of extents penalize runtime significantly. Future work will continue active development on the `zns-tools` to enhance runtime and memory consumption by adapting distributed systems and HPC algorithms and data structures to distribute work across available system resources.

Acronyms

AHCI Advanced Host Controller Interface.

AMF Application Managed Flash.

AMFS Application Managed File Streams.

API Application Programming Interface.

AR Address Remapping.

ARS Adaptive Reserved Space.

AT Age-Threshold.

BIOS Basic I/O System.

CAFTL Content Aware FTL.

CAT Cost-Age-Time.

CB Cost-Benefit.

CDF Cumulative Distribution Function.

CFLRU Clean First Least Recently Used.

CFQ Completely Fair Queuing.

CMOS Complementary Metal-Oxide Semiconductor.

CoW Copy on Write.

CP Checkpoint.

CPU Central Processing Unit.

DAC Dynamic Data Clustering.

DMA Direct Memory Access.

DN-Chain Data Node Chain.

DRAM Dynamic RAM.

eBPF extended Berkeley Packet Filter.

ECC Error Correction Codes.

EQ Evaluation Question.

F2FS Flash-Friendly File System.

FAB Flash Aware Buffer.

FaGC File-aware Garbage Collection.

FIC Flash Integration Challenge.

FIFO	First in First out.
FPC	File Access Pattern-Guided Compression.
FSP	File System as Processes.
FTL	Flash Translation Layer.
FUA	Forced Unit Access.
FUSE	Filesystem in USErspace.
GC	Garbage Collection.
HDD	Hard Disk Drive.
HPC	High Performance Computing.
I/O	Input/Output.
ICT	Information and Communications Technology.
inode	index node.
IOPS	I/O Operations per Second.
IoT	Internet of Things.
IPC	Inter-Process Communication.
L2P	Logical-to-Physical.
LBA	Logical Block Address.
LBADS	LBA Data Size.
LBAF	LBA Format.
LFS	Log-Structured File System.
LIZA	Lifetime-based Zone Allocation.
LRU	Least Recently Used.
LSM-Tree	Log-Structured Merge Tree.
LZ	Lempel-Ziv.
MBF	Multiple Bloom Filters.
MDTS	Maximum Data Transfer Size.
MLC	Multi-Level Cell.
mlog	Minor Log.
MRU	Most Recently Used.
msF2FS	Multi-Streamed F2FS.
NAT	Node Address Table.
NVM	Non-Volatile Memory.
NVMe	Non-Volatile Memory Express.

OCSSD Open-Channel SSD.

OOB Out-Of-Band.

OPS Over-Provisioning Space.

P2L Physical-to-Logical.

PBA Page Boundary Alignment.

PBA Physical Block Address.

PCI Peripheral Component Interconnect.

PCIe PCI Express.

PEB Physical Erase Block.

PID Process Identifier.

PIO B-Tree Parallel I/O B-Tree.

PLC Penta-Level Cell.

QPSC Quasi Preemptive Segment Cleaning.

RA Read Amplification.

RAM Random Access Memory.

RFS Refactored File System.

RL Range Locking.

RLE Run Length Encoding.

RLSQ Related Literature Studies Query.

RM Research Methodology.

RM-IPU Remap-Based In-Place-Updates.

RPS Reader Pass-through Semaphore.

RQ Research Question.

RQs Research Questions.

RSQ Relevant Studies Query.

SA Space Amplification.

SAC Suspend Aware Cleaning.

SATA Serial Advanced Technology Attachment.

SCJ Segment Cleaning Journal.

SCM Storage Class Memory.

SDF Software-Defined Flash.

SIT Segment Information Table.

SLC Single-Level Cell.

SLR Systematic Literature Review.

SMR Shingled Magnetic Recording.

SPDK Storage Performance Development Kit.

SPF Stream Pinned Files.

SRQ Survey Research Question.

SRR Streamed Round-Robin.

SSA Segment Summary Area.

SSAB Stream Section Allocation Barrier.

SSD Solid State Drive.

SSTable Sorted String Table.

TLB Translation Lookaside Buffer.

UFT Update Frequency Table.

V2P Virtual-to-Physical.

VBA Virtual Block Address.

VBQueue Valid Block Queue.

VFIO Virtual Function I/O.

VFIO-PCI VFIO-PCI.

VFS Virtual File System.

VM Virtual Machine.

WA Write Amplification.

WAL Write-Ahead Log.

WL Wear Leveling.

WODS Write Optimized Data Structure.

WP Write Pointer.

ZAC Zoned Block Device ATA Command Set.

ZBC Zoned Block Command.

ZNS Zoned Namespace.

ZSLBA Zone Start LBA.

Bibliography

[1] SFS: Random Write Considered Harmful in Solid State Drives. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, 2012. USENIX Association. URL `https://www.usenix.org/conference/fast12/sfs-random-write-considered-harmful-solid-state-drives`.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 57–70, USA, 2008. USENIX Association.

[3] Miklós Ajtai. The complexity of the pigeonhole principle. *Combinatorica*, 14(4):417–433, 1994.

[4] One Aleph. YAFFS: Yet another Flash file system. *http://www. yaffs. net*, 2001.

[5] Chen Luo an. LSM-based Storage Techniques: A Survey. *ArXiv preprint*, abs/1812.07527, 2018. URL `https://arxiv.org/abs/1812.07527`.

[6] Nick Antonopoulos and Lee Gillam. *Cloud computing*. Springer, 2010.

[7] Remzi H. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, volume 42. 2017. URL `https://www.usenix.org/publications/login/spring2017/arpaci-dusseau`.

[8] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. What you can't forget: exploiting parallelism for zoned namespaces. In Ali Anwar, Dimitris Skourtis, Sudarsun Kannan, and Xiaosong Ma, editors, *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*, pages 79–85. ACM, 2022. doi: 10.1145/3538643.3539744. URL `https://doi.org/10.1145/3538643.3539744`.

[9] Maria Ijaz Baig, Liyana Shuib, and Elaheh Yadegaridehkordi. Big data adoption: State of the art and research challenges. *Information Processing & Management*, 56(6):102095, 2019. ISSN 0306-4573. doi: https://doi.org/10.1016/j.ipm.2019.102095. URL `https://www.sciencedirect.com/science/article/pii/S0306457319301773`.

[10] Felix Beierle, Vinh Thuy Tran, Mathias Allemand, Patrick Neff, Winfried Schlee, Thomas Probst, Johannes Zimmermann, and Rüdiger Pryss. What data are smartphone users willing to share with researchers? designing and evaluating a privacy model for mobile data collection apps. *Journal of Ambient Intelligence and Humanized Computing*, 11:2277–2289, 2020.

[11] Hanmant P Belgal, Nick Righos, Ivan Kalastirsky, Jeff J Peterson, Robert Shiner, and Neal Mielke. A new reliability model for post-cycling charge retention of flash memories. In *2002 IEEE International Reliability Physics Symposium. Proceedings. 40th Annual (Cat. No. 02CH37320)*, pages 7–20. IEEE, 2002.

[12] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. An introduction to B-trees and write-optimization. *login; magazine*, 40(5), 2015.

[13] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. Znswap: un-block your swap. In Jiri Schindler and Noa Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 1–18. USENIX Association, 2022. URL `https://www.usenix.org/conference/atc22/presentation/bergman`.

[14] Roberto Bez, Emilio Camerlenghi, Alberto Modelli, and Angelo Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[15] Sonja Bezjak, April Clyburne-Sherin, Philipp Conzett, Pedro Fernandes, Edit Görögh, Kerstin Helbig, Bianca Kramer, Ignasi Labastida, Kyle Niemeyer, Fotis Psomopoulos, Tony Ross-Hellauer, René Schneider, Jon Tennant, Ellen Verbakel, Helene Brinken, and Lambert Heller. *Open Science Training Handbook*. Zenodo, April 2018. doi: 10.5281/zenodo.1212496. URL `https://doi.org/10.5281/zenodo.1212496`.

[16] Janki Bhimani, Jingpei Yang, Zhengyu Yang, Ningfang Mi, NHV 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)*, pages 1–2. IEEE, 2017.

[17] Artem B Bityutskiy. JFFS3 design issues, 2005.

[18] Matias Bjørling. Zone Append: A New Way of Writing to Zoned Storage. `https://www.usenix.org/conference/vault20/presentation/bjorling`, Feb 2020. Accessed: 2023-Jan-02.

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

[20] 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)*, pages 689–703. USENIX Association, 2021. ISBN 978-1-939133-23-6. URL `https://www.usenix.org/conference/atc21/presentation/bjorling`.

[21] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *FAST*, pages 115–128, 2010.

[22] Adam Brand, Ken Wu, Sam Pan, and David Chin. Novel read disturb failure mechanism induced by FLASH cycling. In *31st Annual Proceedings Reliability Physics 1993*, pages 127–132. IEEE, 1993.

[23] Solution Brief. Internet of things. 2019.

[24] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, volume 3, pages 546–554. Citeseer, 2003.

[25] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The Next Generation of Ext2/3 Filesystem. In *LSF*, 2007.

[26] Adrian M Caulfield, Laura M Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *ACM Sigplan Notices*, 44(3):217–228, 2009.

[27] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, 2010. doi: 10.1109/MICRO.2010.33.

[28] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In Tim Harris and Michael L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 387–400. ACM, 2012. doi: 10.1145/2150976.2151017. URL `https://doi.org/10.1145/2150976.2151017`.

[29] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 387–400, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450307598. doi: 10.1145/2150976.2151017. URL `https://doi.org/10.1145/2150976.2151017`.

[30] Chandranil Chakraborttii and Heiner Litz. Reducing write amplification in flash by death-time prediction of logical block addresses. In Bruno Wassermann, Michal Malka, Vijay Chidambaram, and Danny Raz, editors, *SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*, pages 11:1–11:12. ACM, 2021. doi: 10.1145/3456727.3463784. URL `https://doi.org/10.1145/3456727.3463784`.

[31] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1126–1130, 2007.

[32] Li-Pin Chang. Hybrid solid-state disks: Combining heterogeneous NAND flash in large SSDs. In *2008 Asia and South Pacific Design Automation Conference*, pages 428–433. IEEE, 2008.

[33] Li-Pin Chang and Tei-Wei Kuo. An adaptive striping architecture for flash memory storage systems of embedded systems. In *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 187–196. IEEE, 2002.

[34] Feng Chen, Song Jiang, and Xiaodong Zhang. Smartsaver: turning flash drive into a disk energy saver for mobile computers. In Wolfgang Nebel, Mircea R. Stan, Anand Raghunathan, Jörg Henkel, and Diana Marculescu, editors, *Proceedings of the 2006 International Symposium on Low Power Electronics and Design, 2006, Tegernsee, Bavaria, Germany, October 4-6, 2006*, pages 412–417. ACM, 2006. doi: 10.1145/1165573.1165674. URL `https://doi.org/10.1145/1165573.1165674`.

[35] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. pages 181–192, 2009. doi: 10.1145/1555349.1555371. URL `https://doi.org/10.1145/1555349.1555371`.

[36] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277. IEEE, 2011.

[37] Feng Chen, Tian Luo, and Xiaodong Zhang. $CAFTL$: A $Content-Aware$ flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[38] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based kv store on hybrid storage. In *FAST*, volume 21, pages 17–32, 2021.

[39] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142, 2007.

[40] Seungyong Cheon and Youjip Won. Exploiting multi-block atomic write in SQLite transaction. In *Proceedings of the International Conference on High Performance Compilation, Computing and Communications*, pages 23–27, 2017.

[41] M-L Chiang and R-C Chang. Cleaning policies in mobile computers using flash memory. *Journal of Systems and Software*, 48(3):213–231, 1999.

[42] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. A new lsm-style garbage collection scheme for zns ssds. In *HotStorage*, pages 1–6, 2020.

[43] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. JFTL: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage (TOS)*, 4(4):1–22, 2009.

[44] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5-6):332–343, 2009.

[45] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: transaction support for next-generation, solid-state drives. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 197–212. ACM, 2013. doi: 10.1145/2517349.2522724. URL `https://doi.org/10.1145/2517349.2522724`.

[46] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[47] INCITS T10 Technical Committee. Information technology - Zoned Block Commands (ZBC). Standard, American National Standards Institute, 2014. Available from: `https://www.t10.org/`.

[48] INCITS T13 Technical Committee. Information technology – Zoned Device ATA Command Set (ZAC). Standard, American National Standards Institute, 2015. Available from: `https://www.t13.org/`.

[49] Kernel Development Community. Cramfs - cram a filesystem onto a small ROM. `https://www.kernel.org/doc/html/latest/filesystems/cramfs.html`. Accessed: 2022-07-10.

[50] Christian Monzio Compagnoni, Akira Goda, Alessandro S Spinelli, Peter Feeley, Andrea L Lacaita, and Angelo Visconti. Reviewing the evolution of the NAND flash technology. *Proceedings of the IEEE*, 105 (9):1609–1633, 2017.

[51] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald Porter, and Martin Farach-Colton. Filesystem Aging: It's more Usage than Fullness. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, 2019. USENIX Association. URL `https://www.usenix.org/conference/hotstorage19/presentation/conway`.

[52] Western Digital Corporation. ZenFS: RocksDB Storage Backend for ZNS SSDs and SMR HDDs. `https://github.com/westerndigitalcorporation/zenfs`. Accessed: 2022-01-21.

[53] Western Digital Corporation. zbdbench. `https://github.com/westerndigitalcorporation/zbdbench`, Accessed: 2023-Feb-01.

[54] Tugrul U Daim, Pattravadee Ploykitikoon, Elizabeth Kennedy, and Woraruthai Choothian. Forecasting the future of data storage: case of hard disk drive and flash memory. *Foresight*, 2008.

[55] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013. doi: 10.1145/2408776.2408794. URL `https://doi.org/10.1145/2408776.2408794`.

[56] Yuhui Deng. What is the future of disk drives, death or rebirth? *ACM Comput. Surv.*, 43(3):23:1–23:27, 2011. doi: 10.1145/1922649.1922660. URL `https://doi.org/10.1145/1922649.1922660`.

[57] Peter Desnoyers. Analytic models of SSD write performance. *ACM Trans. Storage*, 10(2):8:1–8:25, 2014. doi: 10.1145/2577384. URL `https://doi.org/10.1145/2577384`.

[58] Peter Deutsch. GZIP file format specification version 4.3. Technical report, 1996.

[59] Stefano Di Carlo, M Cramia, Paolo Prinetto, and Michele Fabiano. Chapter Design Issues and Challenges of File Systems for Flash Memories. 2011.

[60] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, 2022.

[61] Western Digital. Ultrastar® DC ZN540 Data Sheet. Available from: `https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/ultrastar-dc-zn540-ssd/data-sheet-ultrastar-dc-zn540.pdf`.

[62] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *USENIX Annual Technical Conference*, volume 7, pages 261–274, 2007.

[63] Krijn Doekemeijer. Zone reset performance. In *TropoDB: Design, Implementation and Evaluation of an Optimised KV-Store for NVMe Zoned Namespace Devices*, chapter 6.3.3, pages 145–146. 2022.

[64] Krijn Doekemeijer. TropoDB. `https://github.com/atlarge-research/tropodb`, Accessed: 2023-Feb-01.

[65] Krijn Doekemeijer and Animesh Trivedi. Key-Value Stores on Flash Storage Devices: A Survey. *ArXiv preprint*, abs/2205.07975, 2022. URL `https://arxiv.org/abs/2205.07975`.

[66] Fred Douglis and Arun Iyengar. Application-specific Delta-encoding via Resemblance Detection. In *USENIX annual technical conference, general track*, pages 113–126. San Antonio, TX, USA, 2003.

[67] Viacheslav Dubeyko. SSDFS: Towards LFS Flash-Friendly File System without GC operation. *ArXiv preprint*, abs/1907.11825, 2019. URL `https://arxiv.org/abs/1907.11825`.

[68] Adanma Cecilia Eberendu et al. Unstructured data: an overview of the data of big data. *International Journal of Computer Trends and Technology*, 38(1):46–50, 2016.

[69] Christian Egger. File systems for flash devices, 2010.

[70] Emery D. Berger, Stephen M. Blackburn, Matthias Hauswirth, Michael W. Hicks. A Checklist Manifesto for Empirical Evaluation: A Preemptive Strike Against a Replication Crisis in Computer Science. `https://blog.sigplan.org/2019/08/28/a-checklist-manifesto-for-empirical-evaluation-a-preemptive-strike-against-a-replication-crisis-in-computer-science/`, August 2019.

[71] Jörn Engel and Robert Mertens. LogFS-finally a scalable flash file system. In *12th International Linux System Technology Conference*, 2005.

[72] K. Eshghi and R. Micheloni. *SSD Architecture and PCI Express Interface*, pages 19–45. Springer Netherlands, Dordrecht, 2013. ISBN 978-94-007-5146-0. doi: 10.1007/978-94-007-5146-0_2. URL `https://doi.org/10.1007/978-94-007-5146-0_2`.

[73] Facebook. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. `https://github.com/facebook/rocksdb`, . Accessed: 2023-03-01.

[74] Facebook. RocksDB Wiki: RocksDB Overview. `https://github.com/facebook/rocksdb/wiki/RocksDB-Overview`, . Accessed: 2023-03-01.

[75] Timothy R. Feldman and Garth A. Gibson. Shingled Magnetic Recording: Areal Density Increase Requires New Data Management. *login Usenix Mag.*, 38, 2013.

[76] Werner Fischer and Georg Schönberger. Linux Storage Stack Diagram. *Online] http://www. thomas-krenn. com/en/wiki/Linux_Storage_Stack_Diagram (License: CC-BY-SA 3.0, modified by Ingu Kang)*, 2017.

[77] Annie P Foong, Bryan Veal, and Frank T Hady. Towards SSD-Ready Enterprise Platforms. In *ADMS@ VLDB*, pages 15–21, 2010.

[78] Agostino Forestiero, Clara Pizzuti, and Giandomenico Spezzano. A single pass algorithm for clustering evolving data streams based on swarm intelligence. *Data Mining and Knowledge Discovery*, 26(1):1–26, 2013.

[79] Yinjin Fu, Hong Jiang, Nong Xiao, Lei Tian, and Fang Liu. Aa-dedupe: An application-aware source deduplication approach for cloud backup services in the personal computing environment. In *2011 IEEE International Conference on Cluster Computing*, pages 112–120. IEEE, 2011.

[80] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)*, 37(2):138–163, 2005.

[81] Om Rameshwar Gatla, Mai Zheng, Muhammad Hameed, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojevic, Cyril Guyot, and Robert Mateescu. Towards robust file system checkers. *ACM Transactions on Storage (TOS)*, 14(4):1–25, 2018.

[82] Xiongzi Ge, Zhichao Cao, David H. C. Du, Pradeep Ganesan, and Dennis Hahn. Hintstor: A framework to study I/O hints in heterogeneous storage. *ACM Trans. Storage*, 18(2):18:1–18:24, 2022. doi: 10.1145/3489143. URL `https://doi.org/10.1145/3489143`.

[83] Norjihan Abdul Ghani, Suraya Hamid, Ibrahim Abaker Targio Hashem, and Ejaz Ahmed. Social media big data analytics: A survey. *Computers in Human Behavior*, 101:417–428, 2019.

[84] Garth Gibson and Greg Ganger. Principles of Operation for Shingled Disk Devices. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)*, Portland, OR, 2011. USENIX Association. URL `https://www.usenix.org/conference/hotstorage11/principles-operation-shingled-disk-devices`.

[85] Emmanuel Goossaert. Coding for ssds–part 2: Architecture of an ssd and benchmarking, feb 2014.

[86] Nitesh Goyal and Rabi Mahapatra. Energy characterization of cramfs for embedded systems. In *International Workshop on Software Support for Portable Storage (IWSSPS) held in conjunction with the IEEE Real-Time and Embedded Systems and Applications Symposium (RTAS 2005)*, 2005.

[87] Goetz Graefe et al. Modern B-tree techniques. *Foundations and Trends® in Databases*, 3(4):203–402, 2011.

[88] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. pages 229–240, 2009. doi: 10.1145/1508244.1508271. URL `https://doi.org/10.1145/1508244.1508271`.

[89] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke. DRPM: dynamic speed control for power management in server class disks. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 169–179. IEEE, 2003.

[90] Hyunho Gwak and Dongkun Shin. SCJ: Segment Cleaning Journaling for Log-Structured File Systems. *IEEE Access*, 9:142437–142448, 2021. doi: 10.1109/ACCESS.2021.3121423.

[91] Jin-Yong Ha, Young-Sik Lee, and Jin-Soo Kim. Deduplication with block-level content-aware chunking for solid state drives (SSDs). In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1982–1989. IEEE, 2013.

[92] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 759–771, 2017.

[93] Richard R Hamming. *Art of doing science and engineering: Learning to learn.* CRC Press, 1997.

[94] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: advanced zoned namespace interface for supporting in-storage zone compaction. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 147–162. USENIX Association, 2021. URL `https://www.usenix.org/conference/osdi21/presentation/han`.

[95] Boaz Harrosh. zuf: ZUFS Zero-copy User-mode FileSystem. `https://lwn.net/Articles/795996/`. Accessed: 2022-07-10.

[96] John A Hartigan and Manchek A Wong. Algorithm AS 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

[97] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information Systems*, 47:98–115, 2015. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2014.07.006. URL `https://www.sciencedirect.com/science/article/pii/S0306437914001288`.

[98] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The unwritten contract of solid state drives. In Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic, editors, *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 127–144. ACM, 2017. doi: 10.1145/3064176.3064187. URL `https://doi.org/10.1145/3064176.3064187`.

[99] Tim Hegeman and Alexandru Iosup. Survey of Graph Analysis Applications. *ArXiv preprint*, abs/1807.00382, 2018. URL `https://arxiv.org/abs/1807.00382`.

[100] Dominique A Heger and Richard Quinn. Linux 2.6 IO Performance Analysis, Quantification, and Optimization. In *Int. CMG Conference*, 2010.

[101] Gernot Heiser. Systems benchmarking crimes, 2018.

[102] Jen-Wei Hsieh, Li-Pin Chang, and Tei-Wei Kuo. Efficient on-line identification of hot data for flash-memory management. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 838–842, 2005.

[103] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman A. Pletka. Write amplification analysis in flash-based solid state drives. In Miriam Allalouf, Michael Factor, and Dror G. Feitelson, editors, *Proceedings of of SYSTOR 2009: The Israeli Experimental Systems Conference 2009, Haifa, Israel, May 4-6, 2009*, ACM International Conference Proceeding Series, page 10. ACM, 2009. doi: 10.1145/1534530.1534544. URL `https://doi.org/10.1145/1534530.1534544`.

[104] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. BVSSD: Build built-in versioning flash-based solid state drives. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, 2012.

[105] Ping Huang, Guangping Wan, Ke Zhou, Miaoqing Huang, Chunhua Li, and Hua Wang. Improve effective capacity and lifetime of solid state drives. In *2013 IEEE Eighth International Conference on Networking, Architecture and Storage*, pages 50–59. IEEE, 2013.

[106] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[107] Adrian Hunter. A brief introduction to the design of UBIFS, 2008.

[108] Seunghwan Hyun, Hyokyung Bahn, and Kern Koh. LeCramFS: an efficient compressed file system for flash-based portable consumer devices. *IEEE Transactions on Consumer Electronics*, 53(2):481–488, 2007. doi: 10.1109/TCE.2007.381719.

[109] Intel. Intel® SSD D7-P5600 Series. `https://ark.intel.com/content/www/us/en/ark/products/202708/intel-ssd-d7p5600-series-6-4tb-2-5in-pcie-4-0-x4-3d3-tlc.html`, Accessed: 2022-05-02.

[110] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent Van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. The atlarge vision on the design of distributed systems and ecosystems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776. IEEE, 2019.

[111] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. Future computer systems and networking research in the netherlands: A manifesto, 2022. URL `https://arxiv.org/abs/2206.03259`.

[112] Charlie Isaksson, Margaret H Dunham, and Michael Hahsler. SOStream: Self organizing density-based clustering over data stream. In *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pages 264–278. Springer, 2012.

[113] Jeremy Iverson, Chandrika Kamath, and George Karypis. Fast and effective lossy compression algorithms for scientific datasets. In *European Conference on Parallel Processing*, pages 843–856. Springer, 2012.

[114] Shehbaz Jaffer. Evolution of File System design for Solid State Drives.

[115] Shehbaz Jaffer, Stathis Maneas, Andy A. Hwang, and Bianca Schroeder. The reliability of modern file systems in the face of SSD errors. *ACM Trans. Storage*, 16(1):2:1–2:28, 2020. doi: 10.1145/3375553. URL `https://doi.org/10.1145/3375553`.

[116] Ashish Jagmohan, Michele Franceschini, and Luis Lastras. Write amplification reduction in NAND flash through multi-write coding. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6. IEEE, 2010.

[117] Raj Jain. *The art of computer systems performance analysis.* john wiley & sons, 2008.

[118] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. BetrFS: A Right-Optimized Write-Optimized File System. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 301–315, 2015.

[119] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. An Empirical Study of File-System Fragmentation in Mobile Storage Systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, 2016. USENIX Association. URL `https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/ji`.

[120] Cheng Ji, Li-Pin Chang, Riwei Pan, Chao Wu, Congming Gao, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Pattern-Guided File Compression with User-Experience Enhancement for Log-Structured File System on Mobile Devices. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 127–140. USENIX Association, 2021. ISBN 978-1-939133-20-5. URL `https://www.usenix.org/conference/fast21/presentation/ji`.

[121] Chen Jia, ChengYu Tan, and Ai Yong. A grid and density-based clustering algorithm for processing data stream. In *2008 Second International Conference on Genetic and Evolutionary Computing*, pages 517–521. IEEE, 2008.

[122] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4, pages 8–8, 2005.

[123] Yizheng Jiao, Simon Bertron, Sagar Patel, Luke Zeller, Rory Bennett, Nirjhar Mukherjee, Michael A. Bender, Michael Condict, Alex Conway, Martin Farach-Colton, Xiongzi Ge, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. Betrfs: a compleat file system for commodity ssds. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 610–627. ACM, 2022. doi: 10.1145/3492321.3519571. URL `https://doi.org/10.1145/3492321.3519571`.

[124] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Improving SSD Lifetime with Byte-Addressable Metadata. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '17, page 374–384, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450353359. doi: 10.1145/3132402.3132420. URL `https://doi.org/10.1145/3132402.3132420`.

[125] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. FAB: Flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006.

[126] William K Josephson. *A Direct-Access File System for a New Generation of Flash Memory.* PhD thesis, Princeton University, 2011.

[127] William K. Josephson, Lars Ailo Bongo, Kai Li, and David Flynn. DFS: A file system for virtualized flash storage. *ACM Trans. Storage*, 6(3):14:1–14:25, 2010. doi: 10.1145/1837915.1837922. URL `https://doi.org/10.1145/1837915.1837922`.

[128] Dawoon Jung, Jaegeuk Kim, Jinsoo Kim, and Joonwon Lee. Scaleffs: A scalable log-structured flash file system for mobile multimedia systems. *ACM Trans. Multim. Comput. Commun. Appl.*, 5(1):9:1–9:18, 2008. doi: 10.1145/1404880.1404889. URL `https://doi.org/10.1145/1404880.1404889`.

[129] Jaemin Jung, Youjip Won, Eunki Kim, Hyungjong Shin, and Byeonggil Jeon. FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage. *ACM Trans. Storage*, 6(1), 2010. ISSN 1553-3077. doi: 10.1145/1714454.1714457. URL `https://doi.org/10.1145/1714454.1714457`.

[130] Jeeyoon Jung and Dongkun Shin. Lifetime-leveling lsm-tree compaction for zns ssd. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 100–105, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393997. doi: 10.1145/3538643.3539741. URL `https://doi.org/10.1145/3538643.3539741`.

[131] Sanghyuk Jung, Yangsup Lee, and Yong Ho Song. A process-aware hot/cold identification scheme for flash memory storage systems. *IEEE Transactions on Consumer Electronics*, 56(2):339–347, 2010.

[132] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging what you see and what you don't see. A file system aging approach for modern storage systems. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 691–704, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/kadekodi.

[133] Dong Hyun Kang, Gihwan Oh, Dongki Kim, In Hwan Doh, Changwoo Min, Sang-Won Lee, and Young Ik Eom. When address remapping techniques meet consistency guarantee mechanisms. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[134] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed $Solid-State$ Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[135] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261, Santa Clara, CA, 2015. USENIX Association. ISBN 978-1-931971-225. URL https://www.usenix.org/conference/atc15/technical-session/presentation/kang.

[136] Mincheol Kang, Wonyoung Lee, Jinkwon Kim, and Soontae Kim. PR-SSD: Maximizing Partial Read Potential By Exploiting Compression and Channel-Level Parallelism. *IEEE Transactions on Computers*, 2022.

[137] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: transactional FTL for SQLite databases. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 97–108. ACM, 2013. doi: 10.1145/2463676.2465326. URL https://doi.org/10.1145/2463676.2465326.

[138] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-Access File System with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 241–256, Oakland, CA, 2018. USENIX Association. ISBN 978-1-931971-42-3. URL https://www.usenix.org/conference/fast18/presentation/kannan.

[139] S Kapoor and A Chopra. A review of Lempel Ziv compression techniques. *IJCST*, 4(2), 2013.

[140] P Kavitha. A survey on lossless and lossy data compression methods. *International Journal of Computer Science & Engineering Technology*, 7(03):110–114, 2016.

[141] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX*, pages 155–164, 1995.

[142] Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, ver. 2.3 ebse technical report. ebse, 2007.

[143] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Storage Gardening: Using a Virtualization Layer for Efficient Defragmentation in the WAFL File System. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 65–78, Boston, MA, 2019. USENIX Association. ISBN 978-1-939133-09-0. URL https://www.usenix.org/conference/fast19/presentation/kesavan.

[144] Ram Kesavan, Matthew Curtis-Maury, Vinay Devadas, and Kesari Mishra. Countering fragmentation in an enterprise storage system. *ACM Trans. Storage*, 15(4):25:1–25:35, 2020. doi: 10.1145/3366173. URL https://doi.org/10.1145/3366173.

[145] Naim Kheir. *Systems modeling and computer simulation.* Routledge, 2018.

[146] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, 2016. USENIX Association. URL `https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim`.

[147] Hyojun Kim and Youjip Won. MNFS: mobile multimedia file system for NAND flash based storage device. In *CCNC 2006. 2006 3rd IEEE Consumer Communications and Networking Conference, 2006.*, volume 1, pages 208–212. IEEE, 2006.

[148] Jaegeuk Kim. [PATCH 2/4] f2fs: support atomic_write feature for database. `https://lkml.org/lkml/2014/9/26/19`, 9 2014.

[149] Jaegeuk Kim. DEFRAG.F2FS. `https://manpages.debian.org/testing/f2fs-tools/defrag.f2fs.8.en.html`, 2021.

[150] Jaegeuk Kim, Heeseung Jo, Hyotaek Shim, Jin-Soo Kim, and Seungryoul Maeng. Efficient Metadata Management for Flash File Systems. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 535–540, 2008. doi: 10.1109/ISORC.2008.34.

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

[152] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-aware I/O management for solid state disks (SSDs). *IEEE Transactions on Computers*, 61(5):636–649, 2011.

[153] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-Aware I/O Management for Solid State Disks (SSDs). *IEEE Transactions on Computers*, 61(5):636–649, 2012. doi: 10.1109/TC.2011.76.

[154] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in SSDs: Model and quantitative analysis. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2012.

[155] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007. URL `https://www.elsevier.com/__data/promis_misc/525444systematicreviewsguide.pdf`.

[156] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In *2010 10th IEEE/ACM international conference on cluster, cloud and grid computing*, pages 398–407. IEEE, 2010.

[157] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Oper. Syst. Rev.*, 40(3):102–107, 2006. doi: 10.1145/1151374.1151375. URL `https://doi.org/10.1145/1151374.1151375`.

[158] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019.

[159] Tei-Wei Kuo, Jen-Wei Hsieh, Li-Pin Chang, and Yuan-Hao Chang. Configurability of performance and overheads in flash management. In *Asia and South Pacific Conference on Design Automation, 2006.*, pages 8–pp. IEEE, 2006.

[160] Se Jin Kwon, Arun Ranjitkar, Young-Bae Ko, and Tae-Sun Chung. FTL algorithms for NAND-type flash memories. *Design Automation for Embedded Systems*, 15(3):191–224, 2011.

[161] Stefan K. Lai. Brief History of ETOX NOR Flash Memory. *Journal of Nanoscience and Nanotechnology*, 12(10):7597–7603, 2012. ISSN 1533-4880. doi: doi:10.1166/jnn.2012.6649. URL `https://www.ingentaconnect.com/content/asp/jnn/2012/00000012/00000010/art00002`.

[162] Butler Lampson. Principles for computer system design. In *ACM Turing award lectures*, page 1992. 1993.

[163] Dave Landsman and D Walker. AHCI and NVMe as interfaces for SATA Express™ Devices, 2013.

[164] Tomer Lange, Joseph (Seffi) Naor, and Gala Yadgar. Offline and online algorithms for ssd management. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(3), dec 2021. doi: 10.1145/3491045. URL `https://doi.org/10.1145/3491045`.

[165] Kristian Valur Laursen Olason, Alexandru Uta, Alexandru Iosup, Paul Melis, Damian Podareanu, and Valeriu Codreanu. Beneath the SURFace: An MRI-like View into the Life of a 21st Century Datacenter, June 2020. URL `https://doi.org/10.5281/zenodo.3878143`.

[166] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write optimization of log-structured flash file system for parallel I/O on manycore servers. In Moshik Hershcovitch, Ashvin Goel, and Adam Morrison, editors, *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 21–32. ACM, 2019. doi: 10.1145/3319647.3325828. URL `https://doi.org/10.1145/3319647.3325828`.

[167] 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)*, pages 273–286, Santa Clara, CA, 2015. USENIX Association. ISBN 978-1-931971-201. URL `https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee`.

[168] Chul Lee, Sung Hoon Baek, and Kyu Ho Park. A Hybrid Flash File System Based on NOR and NAND Flash Memories for Embedded Devices. *IEEE Transactions on Computers*, 57(7):1002–1008, 2008. doi: 10.1109/TC.2008.14.

[169] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. Compaction-aware zone allocation for lsm based key-value store on zns ssds. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 93–99, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393997. doi: 10.1145/3538643.3539743. URL `https://doi.org/10.1145/3538643.3539743`.

[170] Jongsung Lee and Jin-Soo Kim. An Empirical Study of Hot/Cold Data Separation Policies in Solid State Drives (SSDs). In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321167. doi: 10.1145/2485732.2485745. URL `https://doi.org/10.1145/2485732.2485745`.

[171] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18–es, 2007.

[172] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1075–1086. ACM, 2008. doi: 10.1145/1376616.1376723. URL `https://doi.org/10.1145/1376616.1376723`.

[173] Sungjin Lee, Keonsoo Ha, Kangwon Zhang, Jihong Kim, and Junghwan Kim. FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. In *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA, 2009. USENIX Association. URL `https://www.usenix.org/conference/usenix-09/flexfs-flexible-flash-file-system-mlc-nand-flash-memory`.

[174] Sungjin Lee, Jihong Kim, and Arvind Mithal. Refactored design of i/o architecture for flash storage. *IEEE Computer Architecture Letters*, 14(1):70–74, 2014.

[175] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 339–353, Santa Clara, CA, 2016. USENIX Association. ISBN 978-1-931971-28-7. URL `https://www.usenix.org/conference/fast16/technical-sessions/presentation/lee`.

[176] Yongmyung Lee, Jong-Hyeok Park, Jonggyu Park, Hyunho Gwak, Dongkun Shin, Young Ik Eom, and Sang-Won Lee. When F2FS Meets Address Remapping. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 31–36, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393997. doi: 10.1145/3538643.3539755. URL `https://doi.org/10.1145/3538643.3539755`.

[177] Yair Levy and Timothy J Ellis. A systems approach to conduct an effective literature review in support of information systems research. *Informing Science*, 9, 2006.

[178] DeRen Li, JianJun Cao, and Yuan Yao. Big data in smart cities. *Sci. China Inf. Sci.*, 58(10):1–12, 2015.

[179] Hongyan Li. Flash Saver: Save the Flash-Based Solid State Drives through Deduplication and Delta-encoding. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 436–441. IEEE, 2012.

[180] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: cheap, accurate, scalable and extensible flash emulator. In Nitin Agrawal and Raju Rangaswami, editors, *16th USENIX Conference on File and Storage Technologies, FAST 2018, Oakland, CA, USA, February 12-15, 2018*, pages 83–90. USENIX Association, 2018. URL `https://www.usenix.org/conference/fast18/presentation/li`.

[181] Jiangpeng Li, Kai Zhao, Xuebin Zhang, Jun Ma, Ming Zhao, and Tong Zhang. How Much Can Data Compressibility Help to Improve NAND Flash Memory Lifetime? In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 227–240, Santa Clara, CA, 2015. USENIX Association. ISBN 978-1-931971-201. URL `https://www.usenix.org/conference/fast15/technical-sessions/presentation/li`.

[182] Nanqinqin Li, Mingzhe Hao, Huaicheng Li, Xing Lin, Tim Emami, and Haryadi S Gunawi. Fantastic ssd internals and how to learn and use them. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 72–84, 2022.

[183] Shuai Li, Wei Tong, Jingning Liu, Bing Wu, and Yazhi Feng. Accelerating garbage collection for 3D MLC flash memory with SLC blocks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[184] Zheng Li, Shuangwu Zhang, Jingning Liu, Wei Tong, Yu Hua, Dan Feng, and Chenye Yu. A software-defined fusion storage system for PCM and NAND flash. In *2015 IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, pages 1–6, 2015. doi: 10.1109/NVMSA.2015.7304361.

[185] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated File System for Flash Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 877–891. USENIX Association, 2021. ISBN 978-1-939133-23-6. URL `https://www.usenix.org/conference/atc21/presentation/liao`.

[186] Seung-Ho Lim. DeFFS: Duplication-eliminated flash file system. *Computers & Electrical Engineering*, 37(6):1122–1136, 2011. ISSN 0045-7906. doi: https://doi.org/10.1016/j.compeleceng.2011.06.007. URL `https://www.sciencedirect.com/science/article/pii/S0045790611000917`.

[187] Seung-Ho Lim and Kyu-Ho Park. An efficient NAND flash file system for flash memory storage. *IEEE Transactions on Computers*, 55(7):906–912, 2006. doi: 10.1109/TC.2006.96.

[188] Yoohyuk Lim, Jaemin Lee, Cassiano Campes, and Euiseong Seo. Parity-Stream separation and SLC/MLC convertible programming for life span and performance improvement of SSD RAIDs. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association. URL `https://www.usenix.org/conference/hotstorage17/program/presentation/lim`.

[189] Chun-Yi Liu, Yu-Ming Chang, and Yuan-Hao Chang. Read leveling for flash storage systems. In Dalit Naor, Gernot Heiser, and Idit Keidar, editors, *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, Haifa, Israel, May 26-28, 2015*, pages 5:1–5:10. ACM, 2015. doi: 10.1145/2757667.2757679. URL `https://doi.org/10.1145/2757667.2757679`.

[190] Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Sudarsun Kannan. File Systems as Processes. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, 2019. USENIX Association. URL `https://www.usenix.org/conference/hotstorage19/presentation/liu`.

[191] Chih-Yuan Lu, Kuang-Yeu Hsieh, and Rich Liu. Future challenges of flash memory technologies. *Microelectronic engineering*, 86(3):283–286, 2009.

[192] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 115–122, 2013. doi: 10.1109/ICCD.2013.6657033.

[193] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, San Jose, CA, 2013. USENIX Association. ISBN 978-1-931971-99-7. URL `https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou`.

[194] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, San Jose, CA, 2013. USENIX Association. ISBN 978-1-931971-99-7. URL `https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou`.

[195] Youyou Lu, Jiwu Shu, and Wei Wang. ReconFS: A Reconstructable File System on Flash Storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 75–88, Santa Clara, CA, 2014. USENIX Association. ISBN ISBN 978-1-931971-08-9. URL `https://www.usenix.org/conference/fast14/technical-sessions/presentation/lu`.

[196] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. High-performance and lightweight transaction support in flash-based SSDs. *IEEE Transactions on Computers*, 64(10):2819–2832, 2015.

[197] Youyou Lu, Jiwu Shu, and Jiacheng Zhang. Mitigating Synchronous I/O Overhead in File Systems on Open-Channel SSDs. *ACM Trans. Storage*, 15(3), 2019. ISSN 1553-3077. doi: 10.1145/3319369. URL `https://doi.org/10.1145/3319369`.

[198] Kainan Ma, Ming Liu, Tao Li, Yibo Yin, and Hongda Chen. A Low-Cost Improved Method of Raw Bit Error Rate Estimation for NAND Flash Memory of High Storage Density, 2020. ISSN 2079-9292. URL `https://www.mdpi.com/2079-9292/9/11/1900`.

[199] Umesh Maheshwari. From blocks to rocks: A natural extension of zoned namespaces. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '21, page 21–27, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385503. doi: 10.1145/3465332.3470870. URL `https://doi.org/10.1145/3465332.3470870`.

[200] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, pages 315–322. USENIX Association, 2016. URL `https://www.usenix.org/conference/fast16/technical-sessions/presentation/mandal`.

[201] Charles Manning. YAFFS: the NAND-specific flash file system. *Linuxdevices. org*, 2002.

[202] Charles Manning. How YAFFS works. *Retrieved April*, 6:2011, 2010.

[203] Marí. Understanding and taming SSD read performance variability: HDFS. *ArXiv preprint*, abs/1903.09347, 2019. URL `https://arxiv.org/abs/1903.09347`.

[204] Biswajit Mazumder and Jason O. Hallstrom. A Fast, Lightweight, and Reliable File System for Wireless Sensor Networks. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344852. doi: 10.1145/2968478.2968486. URL `https://doi.org/10.1145/2968478.2968486`.

[205] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[206] Ralph C Merkle. One way hash functions and DES. In *Conference on the Theory and Application of Cryptology*, pages 428–446. Springer, 1989.

[207] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. *ACM SIGMETRICS Performance Evaluation Review*, 43(1):177–190, 2015.

[208] Pratik Mishra and Arun Somani. Host managed contention avoidance storage solutions for Big Data. *Journal of Big Data*, 4:18, 2017. doi: 10.1186/s40537-017-0080-9.

[209] Mahsa Moallem. *A study on the performance evaluation of Linux I/O schedulers.* 2008.

[210] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea R Stan. How I learned to stop worrying and love flash endurance. In *2nd Workshop on Hot Topics in Storage and File Systems (HotStorage 10)*, 2010.

[211] Steve Morgan. The 2020 Data Attack Surface Report. `https://1c7fab3im83f5gqiow2qqs2k-wpengine.netdna-ssl.com/wp-content/uploads/2020/12/ArcserveDataReport2020.pdf`, 2020. Accessed: 2022-04-28.

[212] Keshava Munegowda, GT Raju, and Veera Manikandan Raju. Evaluation of file systems for solid state drives. In *Proceedings of the Second International Conference on Emerging Research in Computing, Information, Communication and Applications*, pages 342–348, 2014.

[213] Benjamin Nahill and Zeljko Zilic. FLogFS: A lightweight flash log file system. In *2015 IEEE 12th International Conference on Wearable and Implantable Body Sensor Networks (BSN)*, pages 1–6. IEEE, 2015.

[214] FEI Ning, Yi Zhuang, Chun-Ling Chen, and YANG Liang. Design, implementation and evaluation of write-enabled CramFS. *The Journal of China Universities of Posts and Telecommunications*, 18(3):124–128, 2011.

[215] MFXJ Oberhumer. LZO-a real-time data compression library. *http://www.oberhumer.com/opensource/lzo/*, 2008.

[216] Joontaek Oh, Sion Ji, Yongjin Kim, and Youjip Won. exF2FS: Transaction Support in Log-Structured Filesystem. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 345–362, Santa Clara, CA, 2022. USENIX Association. ISBN 978-1-939133-26-7. URL `https://www.usenix.org/conference/fast22/presentation/oh`.

[217] John Ousterhout. Always measure one level deeper. *Communications of the ACM*, 61(7):74–83, 2018.

[218] Jian Ouyang, Shiding Lin, Jiang Song, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, pages 471–484. ACM, 2014. doi: 10.1145/2541940.2541959. URL `https://doi.org/10.1145/2541940.2541959`.

[219] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K Panda. Beyond block I/O: Rethinking traditional storage primitives. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 301–311. IEEE, 2011.

[220] Krishna Parat and Chuck Dennison. A floating gate based 3D NAND technology with CMOS under array. In *2015 IEEE International Electron Devices Meeting (IEDM)*, pages 3–3. IEEE, 2015.

[221] Dongchul Park and David HC Du. Hot data identification for flash-based storage systems using multiple bloom filters. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2011.

[222] Dongil Park, Seungyong Cheon, and Youjip Won. Suspend-aware Segment Cleaning in Log-structured File System. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara, CA, 2015. USENIX Association. URL `https://www.usenix.org/conference/hotstorage15/workshop-program/presentation/park`.

[223] Hyunchan Park, Sam H. Noh, and Chuck Yoo. O1FS: Flash file system with O(1) crash recovery time. *Journal of Systems and Software*, 97:86–96, 2014. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2014.07.008. URL `https://www.sciencedirect.com/science/article/pii/S0164121214001514`.

[224] Jonggyu Park and Young Ik Eom. Fragpicker: A new defragmentation tool for modern storage devices. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 280–294, 2021.

[225] Jonggyu Park, Dong Hyun Kang, and Young Ik Eom. File Defragmentation Scheme for a Log-Structured File System. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342650. doi: 10.1145/2967360.2967622. URL `https://doi.org/10.1145/2967360.2967622`.

[226] Sang Oh Park and Sung Jo Kim. An efficient multimedia file system for NAND flash memory storage. *IEEE Transactions on Consumer Electronics*, 55(1):139–145, 2009. doi: 10.1109/TCE.2009.4814426.

[227] Sang Oh Park and Sung Jo Kim. An Efficient Array File System for Multiple Small-Capacity NAND Flash Memories. In *2011 14th International Conference on Network-Based Information Systems*, pages 569–572, 2011. doi: 10.1109/NBiS.2011.94.

[228] Sang Oh Park and Sung Jo Kim. ENFFiS: an enhanced NAND flash memory file system for mobile embedded multimedia system. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2):1–13, 2013.

[229] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, 2006.

[230] Song-Hwa Park, Tae-Hoon Lee, and Ki-Dong Chung. A Flash file system to support fast mounting for NAND Flash memory based embedded systems. In *International Workshop on Embedded Computer Systems*, pages 415–424. Springer, 2006.

[231] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync() a simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 225–238, 2013.

[232] Youngwoo Park, Seung-Ho Lim, Chul Lee, and Kyu Ho Park. PFFS: a scalable flash memory file system for the hybrid architecture of phase-change RAM and NAND flash. In Roger L. Wainwright and Hisham Haddad, editors, *Proceedings of the 2008 ACM Symposium on Applied Computing (SAC), Fortaleza, Ceara, Brazil, March 16-20, 2008*, pages 1498–1503. ACM, 2008. doi: 10.1145/1363686.1364038. URL `https://doi.org/10.1145/1363686.1364038`.

[233] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3): 45–77, 2007.

[234] percona. Tokudb. `https://github.com/percona/tokudb-engine`. Accessed: 2022-07-10.

[235] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-Channel SSD (What is it Good For). In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL `http://cidrdb.org/cidr2020/papers/p17-picoli-cidr20.pdf`.

[236] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Application crash consistency and performance with CCFS. *ACM Transactions on Storage (TOS)*, 13(3):1–29, 2017.

[237] Vijayan Prabhakaran, Thomas L Rodeheffer, and Lidong Zhou. Transactional Flash. In *OSDI*, volume 8, 2008.

[238] Stephen Pratt and Dominique A Heger. Workload dependent performance evaluation of the linux 2.6 i/o schedulers. In *Proceedings of the Linux symposium*, volume 2, pages 425–448, 2004.

[239] Devashish R Purandare, Peter Wilcox, Heiner Litz, and Shel Finkelstein. Append is near: Log-based data management on zns ssds. In *12th Annual Conference on Innovative Data Systems Research (CIDR'22).*, 2022.

[240] Hongwei Qin, Dan Feng, Wei Tong, Yutong Zhao, Sheng Qiu, Fei Liu, and Shu Li. Better Atomic Writes by Exposing the Flash Out-of-Band Area to File Systems. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2021, page 12–23, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384728. doi: 10.1145/3461648.3463843. URL `https://doi.org/10.1145/3461648.3463843`.

[241] Sheng Qiu and A. L. Narasimha Reddy. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, 2013. doi: 10.1109/MSST.2013.6558434.

[242] Wenwei Qiu, Xiang Chen, Nong Xiao, Fang Liu, and Zhiguang Chen. A New Exploration to Build Flash-Based Storage Systems by Co-designing File System and FTL. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 925–932, 2013. doi: 10.1109/CSE.2013.138.

[243] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 206–213, 2010.

[244] Arul Selvan Ramasamy and Porkumaran Karantharaj. File system and storage array design challenges for flash memory. In *2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE)*, pages 1–8, 2014. doi: 10.1109/ICGCCEE.2014.6922453.

[245] David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical. don't focus on big data; focus on the data that's big. *International Data Corporation (IDC) White Paper*, 2017.

[246] 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)*, pages 257–264, Oakland, CA, 2018. USENIX Association. ISBN 978-1-931971-42-3. URL `https://www.usenix.org/conference/fast18/presentation/rho`.

[247] Bhaskar Prasad Rimal, Admela Jukan, Dimitrios Katsaros, and Yves Goeleven. Architectural requirements for cloud computing systems: an enterprise cloud approach. *Journal of Grid Computing*, 9(1): 3–26, 2011.

[248] Ronald Rivest. The MD5 message-digest algorithm. Technical report, 1992.

[249] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[250] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-Based Solid State Drives. *Proc. VLDB Endow.*, 5(4): 286–297, 2011. ISSN 2150-8097. doi: 10.14778/2095686.2095688. URL `https://doi.org/10.14778/2095686.2095688`.

[251] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992. ISSN 0734-2071. doi: 10.1145/146941.146943. URL `https://doi.org/10.1145/146941.146943`.

[252] Samsung. Ultra-Low Latency with Samsung Z-NAND SSD. `https://semiconductor.samsung.com/resources/brochure/Ultra-Low%20Latency%20with%20Samsung%20Z-NAND%20SSD.pdf`, Accessed: 2022-05-02.

[253] Takashi Sato. ext4 online defragmentation. In *Proceedings of the Linux Symposium*, pages 179–186, 2007.

[254] Sebastian Schildt, Wolf-Bastian Pottner, and Lars Wolf. Contiki ring file system for real-time applications. In *2012 IEEE 8th International Conference on Distributed Computing in Sensor Systems*, pages 364–371. IEEE, 2012.

[255] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 67–80, Santa Clara, CA, 2016. USENIX Association. ISBN 978-1-931971-28-7. URL `https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder`.

[256] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An Implementation of a Log-Structured File System for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[257] Eric Seppanen, Matthew T O'Keefe, and David J Lilja. High performance solid state storage under linux. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2010.

[258] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/seshadri`.

[259] Mansour Shafaei, Peter Desnoyers, and Jim Fitzpatrick. Write Amplification Reduction in Flash-Based SSDs Through Extent-Based Temperature Identification. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, 2016. USENIX Association. URL `https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/shafaei`.

[260] Mamta Sharma et al. Compression using Huffman coding. *IJCSNS International Journal of Computer Science and Network Security*, 10(5):133–141, 2010.

[261] Hojin Shin, Myounghoon Oh, Gunhee Choi, and Jongmoo Choi. Exploring performance characteristics of zns ssds: Observation and implication. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–5. IEEE, 2020.

[262] Hojin Shin, Myunghoon Oh, Gunhee Choi, and Jongmoo Choi. Exploring performance characteristics of ZNS ssds: Observation and implication. In *9th Non-Volatile Memory Systems and Applications Symposium, NVMSA 2020, Seoul, South Korea, August 19-21, 2020*, pages 1–5. IEEE, 2020. doi: 10.1109/NVMSA51238.2020.9188086. URL `https://doi.org/10.1109/NVMSA51238.2020.9188086`.

[263] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. OS I/O Path Optimizations for Flash Solid-state Drives. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 483–488, Philadelphia, PA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL `https://www.usenix.org/conference/atc14/technical-sessions/presentation/shin`.

[264] YunSeung Shin. Non-volatile memory technologies for beyond 2010. In *Digest of Technical Papers. 2005 Symposium on VLSI Circuits, 2005.*, pages 156–159. IEEE, 2005.

[265] Keith A Smith and Margo I Seltzer. File system aging—increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 203–213, 1997.

[266] Livio Soares and Michael Stumm. $FlexSC$: Flexible System Call Scheduling with $Exception-Less$ System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[267] Yongseok Son, Hyuck Han, and Heon Young Yeom. Optimizing file systems for fast storage devices. In Dalit Naor, Gernot Heiser, and Idit Keidar, editors, *Proceedings of the 8th ACM International Systems and Storage Conference, SYSTOR 2015, Haifa, Israel, May 26-28, 2015*, pages 8:1–8:6. ACM, 2015. doi: 10.1145/2757667.2757670. URL `https://doi.org/10.1145/2757667.2757670`.

[268] 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 Sebastian Angel, Baris Kasikci, and Eddie Kohler, editors, *HotOS '21: Workshop on Hot Topics in Operating Systems, Ann Arbor, Michigan, USA, June, 1-3, 2021*, pages 144–151. ACM, 2021. doi: 10.1145/3458336.3465300. URL `https://doi.org/10.1145/3458336.3465300`.

[269] Radu Stoica, Manos Athanassoulis, Ryan Johnson, and Anastasia Ailamaki. Evaluating and repairing write performance on flash devices. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 9–14, 2009.

[270] Hui Sun, Xiao Qin, and Chang-sheng Xie. Exploring optimal combination of a file system and an I/O scheduler for underlying solid state disks. *Journal of Zhejiang University Science C*, 15(8):607–621, 2014.

[271] Anand Suresh, Garth A. Gibson, and Gregory R. Ganger. Shingled Magnetic Recording for Big Data Applications. 2012.

[272] Iain Sutherland, Huw Read, and Konstantinos Xynos. Forensic analysis of smart tv: A current issue and call to arms. *Digital Investigation*, 11(3):175–178, 2014.

[273] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *USENIX Annual Technical Conference*, volume 15, 1996.

[274] Miklos Szeredi. FUSE: Filesystem in userspace. *http://fuse. sourceforge. net*, 2010.

[275] Jianzhe Tai, Deng Liu, Zhengyu Yang, Xiaoyun Zhu, Jack Lo, and Ningfang Mi. Improving flash resource utilization at minimal management cost in virtualized flash-based storage systems. *IEEE Transactions on Cloud Computing*, 5(3):537–549, 2015.

[276] Andrew S Tanenbaum. The microarchitecture level. In *Structured computer organization*, chapter 4.5, page 298. Pearson Education India, 2016.

[277] Andrew S Tanenbaum. Computer systems organization. In *Structured computer organization*, chapter 2.1.5, pages 61–65. Pearson Education India, 2016.

[278] Swamit S. Tannu and Prashant J. Nair. The dirty secret of ssds: Embodied carbon. *CoRR*, abs/2207.10793, 2022. doi: 10.48550/arXiv.2207.10793. URL `https://doi.org/10.48550/arXiv.2207.10793`.

[279] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra Incognita: On the Practicality of User-Space File Systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 15)*, Santa Clara, CA, 2015. USENIX Association. URL `https://www.usenix.org/conference/hotstorage15/workshop-program/presentation/tarasov`.

[280] Nick Tehrany and Animesh Trivedi. Understanding nvme zoned namespace (zns) flash ssd storage devices, 2022. URL `https://arxiv.org/abs/2206.01547`.

[281] Nick Tehrany and Animesh Trivedi. Understanding NVMe Zoned Namespace (ZNS) Flash SSD Storage Devices. *ArXiv preprint*, abs/2206.01547, 2022. URL `https://arxiv.org/abs/2206.01547`.

[282] Nicolas Tsiftes, Adam Dunkels, Zhitao He, and Thiemo Voigt. Enabling large-scale storage in sensor networks with the Coffee file system. In *2009 International Conference on Information Processing in Sensor Networks*, pages 349–360, 2009.

[283] Yaofeng Tu, Yinjun Han, Zhenghua Chen, Zhengguang Chen, and Bing Chen. URFS: A User-space Raw File System based on NVMe SSD. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 494–501, 2020. doi: 10.1109/ICPADS51040.2020.00070.

[284] ubuntu wiki. IOSchedulers. `https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers`, 2019. Accessed: 2022-07-10.

[285] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is big data performance reproducible in modern cloud networks? In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 513–527, 2020.

[286] Benny Van Houdt. A mean field model for a class of garbage collection algorithms in flash-based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):191–202, 2013.

[287] 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)*, pages 59–72, 2017.

[288] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. Performance and Resource Utilization of FUSE User-Space File Systems. *ACM Trans. Storage*, 15(2), 2019. ISSN 1553-3077. doi: 10.1145/3310148. URL `https://doi.org/10.1145/3310148`.

[289] Vijay Vasudevan, Michael Kaminsky, and David G Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–13, 2012.

[290] Robin Verschoren and Benny Van Houdt. On the endurance of the d-choices garbage collection algorithm for flash-based ssds. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 4(3), jul 2019. ISSN 2376-3639. doi: 10.1145/3326121. URL `https://doi.org/10.1145/3326121`.

[291] Laurens Versluis, Roland Mathá, Sacheendra Talluri, Tim Hegeman, Radu Prodan, Ewa Deelman, and Alexandru Iosup. The workflow trace archive: Open-access data from public and private computing infrastructures. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2170–2184, 2020.

[292] vtess. FEMU: Accurate, Scalable and Extensible NVMe SSD Emulator (FAST'18). `https://github.com/vtess/FEMU`, Accessed: 2023-Feb-01.

[293] 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*, pages 1–14, 2014.

[294] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Annual international cryptology conference*, pages 17–36. Springer, 2005.

[295] Jane Webster and Richard Thomas Watson. Analyzing the Past to Prepare for the Future: Writing a Literature Review. *MIS Q.*, 26, 2002.

[296] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 111–118, 2015.

[297] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(06):8–19, 1984.

[298] Mark D Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. The fair guiding principles for scientific data management and stewardship. *Scientific data*, 3(1):1–9, 2016.

[299] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Martin J. Shepperd, Tracy Hall, and Ingunn Myrtveit, editors, *18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014*, pages 38:1–38:10. ACM, 2014. doi: 10.1145/2601248.2601268. URL `https://doi.org/10.1145/2601248.2601268`.

[300] David Woodhouse. JFFS: The journalling flash file system. In *Ottawa linux symposium*, volume 2001, 2001.

[301] NVM Express Workgroup. NVM Express NVM Command Set Specification 2.0. Standard, January 2022. Available from: `https://nvmexpress.org/specifications`.

[302] Bing Wu, Mengye Peng, Dan Feng, and Wei Tong. DualFS: A Coordinative Flash File System with Flash Block Dual-mode Switching. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 65–72, 2020. doi: 10.1109/ICCD50377.2020.00028.

[303] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, 2019. USENIX Association. URL `https://www.usenix.org/conference/hotstorage19/presentation/wu-kan`.

[304] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets. In *2014 Data Compression Conference*, pages 203–212. IEEE, 2014.

[305] Jian Xu and Steven Swanson. $NOVA$: A Log-structured File System for Hybrid $Volatile/Non-volatile$ Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[306] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015.

[307] Hua Yan and Qian Yao. An efficient file-aware garbage collection algorithm for NAND flash-based consumer electronics. *IEEE Transactions on Consumer Electronics*, 60(4):623–627, 2014.

[308] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM Trans. Storage*, 13(3), 2017. ISSN 1553-3077. doi: 10.1145/3121133. URL `https://doi.org/10.1145/3121133`.

[309] Jinfeng Yang, Bingzhe Li, and David J. Lilja. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Trans. Model. Perform. Evaluation Comput. Syst.*, 5(1):4:1–4:28, 2020. doi: 10.1145/3372783. URL `https://doi.org/10.1145/3372783`.

[310] 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, 2014. USENIX Association. URL `https://www.usenix.org/conference/inflow14/workshop-program/presentation/yang`.

[311] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

[312] Lihua Yang, Fang Wang, Zhipeng Tan, Dan Feng, Jiaxing Qian, and Shiyun Tu. ARS: Reducing F2FS Fragmentation for Smartphones using Decision Trees. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1061–1066, 2020. doi: 10.23919/DATE48585.2020.9116318.

[313] Yue Yang and Jianwen Zhu. Algebraic Modeling of Write Amplification in Hotness-Aware SSD. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336079. doi: 10.1145/2757667.2757671. URL `https://doi.org/10.1145/2757667.2757671`.

[314] Zhengyu Yang, Manu Awasthi, Mrinmoy Ghosh, and Ningfang Mi. A fresh perspective on total cost of ownership models for flash storage in datacenters. In *2016 IEEE International conference on cloud computing technology and science (CloudCom)*, pages 245–252. IEEE, 2016.

[315] 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)*, pages 154–161, 2017. doi: 10.1109/CloudCom.2017.14.

[316] Ibrar Yaqoob, Ibrahim Abaker Targio Hashem, Abdullah Gani, Salimah Mokhtar, Ejaz Ahmed, Nor Badrul Anuar, and Athanasios V. Vasilakos. Big data: From beginning to future. *International Journal of Information Management*, 36(6, Part B):1231–1247, 2016. ISSN 0268-4012. doi: https://doi.org/10.1016/j.ijinfomgt.2016.07.009. URL https://www.sciencedirect.com/science/article/pii/S0268401216304753.

[317] Jinsoo Yoo, Joontaek Oh, Seongjin Lee, Youjip Won, Jin-Yong Ha, Jongsung Lee, and Junseok Shim. Orcfs: Orchestrated file system for flash storage. *ACM Trans. Storage*, 14(2):17:1–17:26, 2018. doi: 10.1145/3162614. URL https://doi.org/10.1145/3162614.

[318] Takeshi Yoshimura, Tatsuhiro Chiba, and Hiroshi Horii. EvFS: User-level, Event-Driven File System for Non-Volatile Memory. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, 2019. USENIX Association. URL https://www.usenix.org/conference/hotstorage19/presentation/yoshimura.

[319] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. Optimizing the block I/O subsystem for fast storage devices. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–48, 2014.

[320] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 87–100, Denver, CO, 2016. USENIX Association. ISBN 978-1-931971-30-0. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/zhang.

[321] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. Flashkv: Accelerating kv performance with open-channel ssds. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

[322] Runyu Zhang, Duo Liu, Xianzhang Chen, Xiongxiong She, Chaoshu Yang, Yujuan Tan, Zhaoyan Shen, and Zili Shao. LOFFS: A Low-Overhead File System for Large Flash Memory on Embedded Devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi: 10.1109/DAC18072.2020.9218635.

[323] Runyu Zhang, Duo Liu, Xianzhang Chen, Xiongxiong She, Chaoshu Yang, Yujuan Tan, Zhaoyan Shen, Zili Shao, and Lei Qiao. ELOFS: An Extensible Low-overhead Flash File System for Resource-scarce Embedded Devices. *IEEE Transactions on Computers*, pages 1–1, 2022. doi: 10.1109/TC.2022.3152079.

[324] Yiwen Zhang, Ting Yao, Jiguang Wan, and Changsheng Xie. Building gc-free key-value store on hm-smr drives with zonefs. *ACM Trans. Storage*, 18(3), aug 2022. ISSN 1553-3077. doi: 10.1145/3502846. URL https://doi.org/10.1145/3502846.

[325] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. Remap-SSD: Safely and Efficiently Exploiting SSD Address Remapping to Eliminate Duplicate Writes. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 187–202. USENIX Association, 2021. ISBN 978-1-939133-20-5. URL https://www.usenix.org/conference/fast21/presentation/zhou.

[326] Yue Zhu, Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, Muhib Khan, and Weikuan Yu. Directfuse: Removing the middleman for high-performance fuse file system support. In *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*, pages 1–8, 2018.

[327] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[328] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

[329] Aviad Zuck, Ohad Barzilay, and Sivan Toledo. NANDFS: A Flexible Flash File System for RAM-Constrained Systems. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, page 285–294, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605586274. doi: 10.1145/1629335.1629374. URL `https://doi.org/10.1145/1629335.1629374`.

A
Artifact Reproducibility

In order to provide reproducibility of the results during this thesis work, we detail the setup and install instructions, as well as provide several commands and scripts we use during the evaluation. In order to aid developers starting with kernel development, we detail the development process of msF2FS during this thesis work in Appendix C.

A.1. System Setup

In order to set up a QEMU VM, we must first have an image to install a basic Linux version. We use ubuntu server 22.04 during the evaluation. As setting up for a basic VM image is readily available online, we assume a existing system image to use and proceed with how to enable the specific settings of this study. Firstly, the ZNS device must be passed through to the QEMU VM using vfio-pci. Listing A.1 depicts the commands to passthrough device nvme1 to the QEMU VM. Note, the namepsace used during this process is not important, as long as one namespace of the device is used, as the entire device can only be passed through in this way. To start the QEMU VM with the ZNS device, we provide a startup script in Listing A.2, where the ZNS device is specified with the pci address obtained in the prior command. Further settings specify the address of the installed VM image to use, the system configuration as detailed in Section 6.1, and additional port forwarding to allow to ssh into the VM on port 8889.

```
nty@node1:/home/nty/src$ sudo su
root@node1:/home/nty/src# ls -l /sys/block/nvme1n1/device/device
lrwxrwxrwx 1 root root 0 Jan 27 00:36 /sys/block/nvme1n1/device/device -> ../../../0000:b0:00.0
root@node1:/home/nty/src# echo "0000:b0:00.0" > /sys/bus/pci/drivers/nvme/unbind
root@node1:/home/nty/src# modprobe vfio-pci
root@node1:/home/nty/src# lspci -n -s 0000:b0:00.0
b0:00.0 0108: 1b96:2600
root@node1:/home/nty/src# echo 1b96 2600 > /sys/bus/pci/drivers/vfio-pci/new_id
```

Listing A.1: Attaching the ZNS device to the vfio-pci driver to passthrough to the QEMU VM.

```
#!/bin/bash

set -e

U20_IMG="/home/nty/src/images/ubuntu-22.04-f2fs.qcow2"

sudo ${QEMU_BUILD}/qemu-system-x86_64 -name f2fs-vm -m 25G -cpu host -smp 34 -enable-kvm -nographic \
    -hda ${U20_IMG} \
    -device virtio-scsi-pci,id=scsi0 \
    -device vfio-pci,host=0000:b0:00.0 \
    -net nic,model=virtio \
    -net user,hostfwd=tcp::8889-:22,hostfwd=tcp::3333-:3000 \
    -device virtio-tablet-pci,id=tablet0,serial=virtio-tablet \
```

Listing A.2: VM startup script to start QEMU with the specified ZNS device in passthrough mode.

A.1.1. msF2FS Build and Install

This section depicts the commands for building a Linux kernel with msF2FS support, and install the kernel build. Building and installing the Linux kernel can be done in numerous ways, we only depict one such way, however any method can be applied with the only requirement being that msF2FS support is enabled in the kernel configuration. Listing A.3 depicts the example commands to build and install the msF2FS kernel. The depicted example assumes the kernel is built on a host machine, and build files are copied into a VM to be installed. We build the kernel with debug support (e.g., DebugFS) to retrieve metadata statistics from msF2FS and F2FS. With such configuration enabled, a debug binary package will be generated, which can be removed as it contains additional debug symbols that are not needed, and it consumes significantly more storage space when installed. The kernel configuration file we use for building is available at `https://github.com/nicktehrany/notes/blob/master/kconfigs/.config-f2fs_multi_stream`

```
# Copy the current kernel config file
cp /usr/src/linux-headers-$(uname -r)/.config .config
make menuconfig

# Scroll to File Systems --> F2FS and enable "F2FS multi-streamed data logging (EXPERIMENTAL)"

# build deb packages
make -j10 bindeb-pkg LOCALVERSION=-msf2fs

# Files are in parent dir
cd ..

# Remove the -dbg image (if present), change version number to yours!
rm linux-image-5.19.0-msf2fs-dbg_5.19.0-msf2fs-66_amd64.deb

# copy to host VM machine (Adjust user/system info and target dir)
scp -P 8889 *.deb user@localhost:~/src/f2fs-build-debs/

# Install in VM
sudo dpkg -i *.deb
sudo update-grub2
sudo sync

# Reboot the VM
sudo reboot
```

Listing A.3: Commands for building and installing a Linux kernel with msF2FS support.

A.2. Setting the ZNS Device LBAF

With ZNS devices supporting different LBAF, we provide the instructions to change the LBAF of a device. Firstly, Listing A.4 shows how to identify the supported LBAF on the ZNS device. The showcased output depicts supported LBAF of 512B (equal to LBADS 9, $2^9 = 512$), with and without metadata (ms), and similarly 4KiB LBAF (equal to LBADS 12, $2^{12} = 4,096$). Next, Listing A.5 shows the commands to change the active LBAF to 512B. In order to modify the LBAF of a namespace, we must first delete all existing namespaces, followed by checking the available size (to configure namespaces given the available capacity), followed by creating new namespaces with the new LBAF, and attaching the namespaces to the controller.

```
user@stosys:~/src/msF2FS/$ sudo nvme id-ns /dev/nvme1n2 | grep -i "lbaf"
nlbaf   : 4
lbaf  0 : ms:0   lbads:9  rp:0
lbaf  1 : ms:8   lbads:9  rp:0
lbaf  2 : ms:0   lbads:12 rp:0 (in use)
lbaf  3 : ms:8   lbads:12 rp:0
lbaf  4 : ms:64  lbads:12 rp:0
```

Listing A.4: Checking the support LBAF on a ZNS device.

A.3. FEMU Setup

Modifying the LBAF in FEMU for ZNS emulation requires minor code modifications, depicted in A.6. It only modifies the `lba_index` and utilizes the set value in the device configuration.

```
# Check available space to setup namespace sizes
user@stosys:~$ sudo nvme id-ctrl /dev/nvme1 | grep -i "tnvmcap"
tnvmcap   : 7924214661120
user@stosys:~$ sudo nvme id-ctrl /dev/nvme1 | grep -i "unvmcap"
unvmcap   : 0
user@stosys:~$ sudo nvme delete-ns /dev/nvme1 -n 1
delete-ns: Success, deleted nsid:1
user@stosys:~$ sudo nvme delete-ns /dev/nvme1 -n 2
delete-ns: Success, deleted nsid:2
# Create 4GiB Conventional Space
user@stosys:~$ sudo nvme create-ns /dev/nvme1 -s 8388608 -c 8388608 -b 512 --csi=0
# Remaining Capacity is zoned
user@stosys:~$ sudo nvme create-ns /dev/nvme1 -s 15468593152 -c 15468593152 -b 512 --csi=2
create-ns: Success, created nsid:2
user@stosys:~$ sudo nvme attach-ns /dev/nvme1 -n 1 -c 0
attach-ns: Success, nsid:1
user@stosys:~$ sudo nvme attach-ns /dev/nvme1 -n 2 -c 0
attach-ns: Success, nsid:2
```

Listing A.5: Changing the active LBAF on a ZNS device.

```
diff --git a/hw/femu/femu.c b/hw/femu/femu.c
index dcb5dcaee..ab738b85e 100644
--- a/hw/femu/femu.c
+++ b/hw/femu/femu.c
@@ -631,7 +631,7 @@ static Property femu_props[] = {
     DEFINE_PROP_UINT8("mpsmin", FemuCtrl, mpsmin, 0),
     DEFINE_PROP_UINT8("mpsmax", FemuCtrl, mpsmax, 0),
     DEFINE_PROP_UINT8("nlbaf", FemuCtrl, nlbaf, 5),
-    DEFINE_PROP_UINT8("lba_index", FemuCtrl, lba_index, 0),
+    DEFINE_PROP_UINT8("lba_index", FemuCtrl, lba_index, 3),
     DEFINE_PROP_UINT8("extended", FemuCtrl, extended, 0),
     DEFINE_PROP_UINT8("dpc", FemuCtrl, dpc, 0),
     DEFINE_PROP_UINT8("dps", FemuCtrl, dps, 0),
diff --git a/hw/femu/zns/zns.c b/hw/femu/zns/zns.c
index b870d180a..4dc3bc708 100644
--- a/hw/femu/zns/zns.c
+++ b/hw/femu/zns/zns.c
@@ -134,8 +134,8 @@ static void zns_init_zone_identify(FemuCtrl *n, NvmeNamespace *ns, int lba_index
     id_ns_z->zoc = 0;
     id_ns_z->ozcs = n->cross_zone_read ? 0x01 : 0x00;

-    id_ns_z->lbafe[lba_index].zsze = cpu_to_le64(n->zone_size);
-    id_ns_z->lbafe[lba_index].zdes = n->zd_extension_size >> 6; /* Units of 64B */
+    id_ns_z->lbafe[n->lba_index].zsze = cpu_to_le64(n->zone_size);
+    id_ns_z->lbafe[n->lba_index].zdes = n->zd_extension_size >> 6; /* Units of 64B */

     n->csi = NVME_CSI_ZONED;
     ns->id_ns.nsze = cpu_to_le64(n->num_zones * n->zone_size);
```

Listing A.6: Enforcing a 4KiB LBAF on the emulated ZNS device by FEMU.

A.4. Fio Modifications

We detail the modifications we make to fio to utilize it for the evaluation in Chapter 6. In particular we force fio to use the *F_SET_RW_HINT* flag instead of *F_SET_FILE_RW_HINT*, as it is no longer supported by kernels 5.17+. Listing A.7 shows the diff of the changes made.

```
user@stosys:~/src/fio$ git diff ioengines.c
diff --git a/ioengines.c b/ioengines.c
index e2316ee4..525cbcd1 100644
--- a/ioengines.c
+++ b/ioengines.c
@@ -587,9 +587,6 @@ int td_io_open_file(struct thread_data *td, struct fio_file *f)
                  * the file descriptor. For buffered IO, we need to set
                  * it on the inode.
                  */
-                if (td->o.odirect)
-                        cmd = F_SET_FILE_RW_HINT;
-                else
                 cmd = F_SET_RW_HINT;

                 if (fcntl(f->fd, cmd, &hint) < 0) {
```

Listing A.7: Modifications made to fio in order to correctly utilize file write lifetime hints.

A.5. Running Micro-Benchmarks

This guide details the steps to run all micro-benchmarks depicted in Chapter 6. It assumes to have a full setup with the ZNS device and QEMU VM with the installed kernel. We recommend having two separate QEMU images, one with the default F2FS kernel, and another with the msF2FS kernel installed. Therefore, switching between the two requires to simply switch the QEMU image being booted. The required tools for the micro-benchmarks are; fio and nvme-cli, for which each repository contains respective installation instructions. With the required tools, we clone the msF2FS-bench repository and can run the individual scripts. The various micro-benchmarks are divided into different directories, which each contain a `bench-f2fs` and `bench-msf2fs` script to benchmark F2FS and msF2FS respectively. The scripts assume the ZNS device supports a maximum of 14 active zones to configure the number of msF2FS data streams. The `zns-baseline` experiment has only a single script, as it benchmarks only the ZNS device without file systems. Furthermore, the benchmarking scripts for F2FS and msF2FS always set up a 19GiB `nullblk` device, as this is the minimum space required to format the file system with the particular ZNS device. For differently sized ZNS devices, we recommend scaling the size to the minimum required by F2FS. The scripts can therefore be easily modified to pass the size of the `nullblk` device in its creation function call. To identify the minimum size of the `nullblk` device, simply create a small `nullblk` (e.g., 1GiB) device by using the `nullblk_create` script in the root directory of the benchmarking repository (size argument is given in MiB), and format the ZNS device with F2FS. Upon formatting F2FS reports that the conventional block device has insufficient capacity, as well as stating how much additional space is needed. Therefore, adding the additional space required, as reported by `mkfs.f2fs`, to the initial `nullblk` device provides the minimum required size.

A.6. Running Macro-Benchmarks

As the macro-benchmarks utilize several tools, we depict the instructions to install and setup all the tools. The required tools are libzbd, RocksDB, ZenFS, f2fs-tools, and zbdbench. Listing A.8 depicts the installation instructions for all the tools. It assumes to do global installation for all tools. Note, when cloning Rocksdb, we use our forked repository that contains the msF2FS hint integrations for the different policies. When running the macro-benchmark, simply checkout the respective branch of the integration to be benchmarked, and copy the built db_bench again into `/usr/local/bin`. For conventional F2FS the `main` branch of the repository is at the commit used during this evaluation. Applying the modifications we make to `zbdbench`, we show the diff of our build in Listing A.9. The modifications include, updating the number of keys to 6B, adding an open file limit of 1000 files, fixing the mountpoint where F2FS is to be mounted, and setting the number of streams for as msF2FS. For evaluating the AMFS hint integration we manually modify the policy passed in the F2FS mount to be `-o stream_policy=amfs`. Adapt this to the specified integration to benchmark. For default F2FS remove all added arguments by msF2FS, for stream configuration. It is important to utilize the correct build of RocksDB and copy it again into `/usr/local/bin/` for the benchmark. Lastly, Listing A.10 shows the commands to increase the system file limits and run the `zbdbench` benchmark. We increase the file system limit to ensure that it is not exceeded, in addition to `zbdbench` enforcing an open file limit in the RocksDB configuration. If any of the required tools is not installed globally, simply pass the respective paths (e.g., if installed in the home directory) using the `env`, such as for instance `env "PATH=/home/user/local/bin/"`.

```
# Install libzbd
user@stosys:~/src/$ git clone https://github.com/westerndigitalcorporation/libzbd
user@stosys:~/src/$ cd libzbd
user@stosys:~/src/libzbd/$ sh ./autogen.sh
user@stosys:~/src/libzbd/$ ./configure
user@stosys:~/src/libzbd/$ sudo make install

# Install rocksdb with zenfs
user@stosys:~/src/$ git clone https://github.com/nicktehrany/rocksdb.git
user@stosys:~/src/$ cd rocksdb
# Checkout the respective branch for RocksDB with hint integration
user@stosys:~/src/rocksdb/$ git checkout amfs-2
user@stosys:~/src/rocksdb/$ git clone https://github.com/westerndigitalcorporation/zenfs plugin/zenfs
user@stosys:~/src/rocksdb/$ DEBUG_LEVEL=0 ROCKSDB_PLUGINS=zenfs make -j30 db_bench install
user@stosys:~/src/rocksdb/$ cp db_bench /usr/local/bin/
user@stosys:~/src/rocksdb/$ cd plugin/zenfs/util
user@stosys:~/src/rocksdb/plugin/zenfs/util$ make
user@stosys:~/src/rocksdb/plugin/zenfs/util$ cp zenfs /usr/local/bin/

# Install f2fs-tools
user@stosys:~/src/$ git clone https://git.kernel.org/pub/scm/linux/kernel/git/jaegeuk/f2fs-tools.git
user@stosys:~/src/$ cd f2fs-tools
user@stosys:~/src/f2fs-tools/$ ./configure
user@stosys:~/src/f2fs-tools/$ sudo make install

# Setup zbdbench in the msF2FS-bench repo
user@stosys:~/src/$ cd ~/src/msF2FS-bench/
user@stosys:~/src/msF2FS-bench/$ git submodule update --init --recursive
```

Listing A.8: Instructions to install all required tools for the macro-benchmarks.

```
user@stosys:~/src/msF2FS-bench/submodules/zbdbench$ git diff
diff --git a/benchs/usenix_atc_2021_zns_eval.py b/benchs/usenix_atc_2021_zns_eval.py
index 3fce92d..0adf641 100644
--- a/benchs/usenix_atc_2021_zns_eval.py
+++ b/benchs/usenix_atc_2021_zns_eval.py
@@ -24,7 +24,8 @@ class Run(Bench):
     # Original run on a 2TB ZNS SSD: (3.8B)
     # scale_num = 3800000000
     # The current state of ZenFS creates a bit more space amplification
-    scale_num = 3300000000
+    scale_num = 6000000000

     # All benchmarks
     wb_size = str(2 * 1024 * 1024 * 1024)
@@ -63,7 +64,7 @@ class Run(Bench):
         params = " ", self.db_env_param, \
                       " --key_size=", self.key_size, \
                       " --value_size=", self.value_size, \
-                      " --target_file_size_base=", self.target_fz_base, \
+                      " --open_files=1000 --target_file_size_base=", self.target_fz_base, \
                       " --write_buffer_size=", self.wb_size, \
                       " --max_bytes_for_level_base=", self.max_bytes_for_level_base, \
                       " --max_bytes_for_level_multiplier=4", \
@@ -186,8 +187,9 @@ class Run(Bench):

     def create_mountpoint(self, dev, filesystem):
         relative_mountpoint = "%s_%s" % (dev.strip('/dev/'), filesystem)
-        mountpoint = os.path.join(self.output, relative_mountpoint)
-        os.mkdir(mountpoint)
+        # mountpoint = os.path.join(self.output, relative_mountpoint)
+        # os.mkdir(mountpoint)
+        mountpoint = "/mnt/f2fs"
         return mountpoint, relative_mountpoint

     def create_new_nullblk_dev_config_path(self):
@@ -209,6 +211,8 @@ class Run(Bench):

     def create_f2fs_nullblk_dev(self, dev, container):
         dev_config_path = self.create_new_nullblk_dev_config_path()
+        with open(os.path.join(dev_config_path, 'size') , "w") as f:
+            f.write("19456")
         with open(os.path.join(dev_config_path, 'blocksize') , "w") as f:
             f.write(str(self.get_sector_size(dev)))
         with open(os.path.join(dev_config_path, 'memory_backed') , "w") as f:
@@ -231,8 +235,8 @@ class Run(Bench):
             self.conv_nullblk_dev = self.create_f2fs_nullblk_dev(dev, container)
             self.run_cmd(dev, container, 'mkfs.f2fs', f'-f -o 5 -m -c {dev} {self.conv_nullblk_dev}', f'-v "{self.
     conv_nullblk_dev}:{self.conv_nullblk_dev}"')
            subprocess.check_call('sudo modprobe f2fs', shell=True)
-            subprocess.check_call(f'mount -t f2fs -o active_logs=6,whint_mode=user-based {self.conv_nullblk_dev} {
     mountpoint}', shell=True)
-            self.db_env_param = f'--db=/output/{relative_mountpoint}/eval'
+            subprocess.check_call(f'mount -t f2fs -o hot_data_streams=2 -o warm_data_streams=3 -o cold_data_streams=4 -
     o stream_policy=spf {self.conv_nullblk_dev} /mnt/f2fs', shell=True)
+            self.db_env_param = f'--db=/mnt/f2fs/eval'
             return mountpoint
         else:
             print("Filesystem %s is not currently not supported for ZNS drives in this benchmark" % filesystem)
```

Listing A.9: Diff showing the modifications made to zbdbench in the evaluation.

```
user@stosys:~/src/msF2FS-bench/submodules/zbdbench/$ sudo su
user@stosys:~/src/msF2FS-bench/submodules/zbdbench/$ vim /etc/security/limits.conf
# Insert the following and save the file
* hard nofile 1048576
* soft nofile 1048576

# Manually setting the soft limit to ensure it is valid for the running process
user@stosys:~/src/msF2FS-bench/submodules/zbdbench/$ ulimit -Sn 1048576

# Run the benchmark
user@stosys:~/src/msF2FS-bench/submodules/zbdbench/$ ./run.py -d /dev/nvme0n2 -c no -b usenix_atc_2021_zns_eval
```

Listing A.10: Running zbdbench with a modified file limit configuration.

B
Using msF2FS fcntl() Flags

As we extend *fcntl()* flags to pass information from the application to msF2FS, we provide simplified example code that utilizes these flags for the different scheduling policies. This is not fully functional code, however is meant to provide a foundation on how to use the custom hints in full applications. Appendix B.1 shows the example code for msF2FS with the SPF policy, and Appendix B.2 depicts code of msF2FS configured with the AMFS policy. Both programs can simply be compiled with gcc, as illustrated in Listing B.1.

```
user@stosys:~/src/example/$ gcc -O2 example-code.c -o example-code
```

Listing B.1: Compiling of example programs.

B.1. SPF Policy

This section shows the example code for msF2FS configured with the SPF policy, and passing the *fcntl()* flags to set and unset exclusive streams. The example code illustrated in Listing B.2 illustrated the opening of a file descriptor, setting the exclusive stream flag, and unsetting the exclusive stream map. No write I/O are issued to file, however this code base can simply be expanded to read/write the open file descriptor.

B.2. AMFS Policy

This section shows the example code for msF2FS configured with the AMFS policy, and passing of a stream bitmap. Listing B.3 depicts an example code that opens a file and passes a bitmap of the stream mapping, where this particular file is mapped to stream 2 (note, streams start indexing at 0).

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

#define F_LINUX_SPECIFIC_BASE 1024
#define F_SET_EXCLUSIVE_DATA_STREAM (F_LINUX_SPECIFIC_BASE + 15)
#define F_UNSET_EXCLUSIVE_DATA_STREAM (F_LINUX_SPECIFIC_BASE + 16)

#define ERR_MSG(fmt, ...)                                                      \
    do {                                                                       \
        printf("\033[0;31mError\033[0m: [%s:%d] " fmt, __func__, __LINE__,     \
               ##__VA_ARGS__);                                                 \
        exit(1);                                                               \
    } while (0)

int main(int argc, char *argv[]) {
    int out, flags;

    flags |= O_WRONLY | O_CREAT;

    out = open("/mnt/msf2fs/test-file.txt", flags, 0664);

    if (!out) {
        ERR_MSG("Failed opening file for writing\n");
    }

    /* Set exclusive stream */
    if (fcntl(out, F_SET_EXCLUSIVE_DATA_STREAM) < 0) {
        if (errno == EINVAL) {
            ERR_MSG("F_SET_EXCLUSIVE_DATA_STREAM Invalid Argument\n");
        }

        ERR_MSG("F_SET_EXCLUSIVE_DATA_STREAM Not Supported\n");
    }

    /* Un-set exclusive stream */
    if (fcntl(out, F_UNSET_EXCLUSIVE_DATA_STREAM) < 0) {
        if (errno == EINVAL) {
            ERR_MSG("F_UNSET_EXCLUSIVE_DATA_STREAM Invalid Argument\n");
        }

        ERR_MSG("F_UNSET_EXCLUSIVE_DATA_STREAM Not Supported\n");
    }

    close(out);

    return 0;
}
```

Listing B.2: Example code to set an exclusive stream with msF2FS and SPF, followed by unsetting the stream.

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

#define F_LINUX_SPECIFIC_BASE 1024
#define F_SET_DATA_STREAM_MAP (F_LINUX_SPECIFIC_BASE + 17)

#define ERR_MSG(fmt, ...)                                                       \
    do {                                                                        \
        printf("\033[0;31mError\033[0m: [%s:%d] " fmt, __func__, __LINE__,     \
               ##__VA_ARGS__);                                                  \
        exit(1);                                                                \
    } while (0)

int main(int argc, char *argv[]) {
    int out, flags;
    unsigned long *streammap = 0;

    flags |= O_WRONLY | O_CREAT;

    out = open("/mnt/msf2fs/test-file.txt", flags, 0664);

    if (!out) {
        ERR_MSG("Failed opening file for writing\n");
    }

    streammap = calloc(sizeof(unsigned long), sizeof(char *));

    /* set stream 2 on the stream bitmap */
    *streammap |= (1 << 2);

    /* Pass the stream bitmap to msF2FS */
    if (fcntl(out, F_SET_DATA_STREAM_MAP, streammap) < 0) {
        if (errno == EINVAL) {
            ERR_MSG("F_SET_DATA_STREAM_MAP Invalid Argument\n");
        }

        ERR_MSG("F_SET_DATA_STREAM_MAP Not Supported\n");
    }

    close(out);
    free(streammap);

    return 0;
}
```

Listing B.3: Example code to set the stream mapping with msF2FS and AMFS by passing a bitmap of which streams is file is to be mapped to.

C
Development Cycle

As Linux kernel development is a complicated task, particularly in the early stages of working with the Linux kernel, we detail several helpful setup commands and tools for debugging source code for the Linux kernel.

C.1. Debugging Linux kernel with QEMU and gdb

For the development cycle we need to use a VM, for which QEMU has well established kernel support and debugging tools. Assuming we have a setup a basic VM image, by downloading and installing a distribution `iso` (e.g., Ubuntu Server iso), and having a compiled QEMU, we can start a VM as illustrated in the script depicted in Listing C.1. Particularly important are numerous arguments with the startup command for QEMU. Firstly, we pass it the `bzImage`, of the compiled kernel we are debugging. The `bzImage` is a minimal kernel, without any modules or header files, that can be booted. As it has no modules or header files, functionality is severely limited, requiring any necessary features to be included in the kernel build, instead of being a module. However, including features increases compile time and required space, hence include only what is needed. This typically implies only features to have a running kernel that can be used for debugging. For msF2FS we require a passed through ZNS device, using *vfio-pci*, hence NVMe support and additional parts are required to be built in. Secondly, we append options to the booting with the `bzImage`, which ensure the kernel can be booted, and we disable kernel randomization (`nokaslr`) in order to allow setting of breakpoints with gdb. Lastly, we pass `-S` and `-s`, which have QEMU wait with kernel booting until gdb is attached, and enables to attach gdb to QEMU on `localhost:1234`.

```bash
#!/bin/bash

set -e

U20_IMG="/home/nty/src/images/ubuntu-22.04-f2fs.qcow2"

sudo ${QEMU_HOME}/qemu-system-x86_64 -name ZNS -m 25G -smp 34 -enable-kvm -nographic \
    -hda ${HOME}/src/storage/images/ubuntu-20.04.qcow2 \
    -kernel ${HOME}/src/storage/ZNS/linux/arch/x86/boot/bzImage \
    -append "root=/dev/sdb2 rootfstype=ext4 console=ttyS0 nokaslr" \
    -device vfio-pci,host=0000:60:00.0 \
    -net nic,model=virtio \
    -net user,hostfwd=tcp::8888-:22,hostfwd=tcp::3333-:3000 \
    -device virtio-tablet-pci,id=tablet0,serial=virtio-tablet \
    -S -s
```

Listing C.1: Example script to start a QEMU VM with gdb debugging enabled on port 1234.

With the established QEMU setup, QEMU waits for attaching of a gdb session to the VM, to tell it to continue boot process. Listing C.2 depicts the attaching of gdb to the QEMU VM on the specified port. For illustration, we then set a breakpoint at the allocation function in msF2FS, and continue to start the boot process. With gdb we can set breakpoints at desired functions, example call stacks, variables, memory, and further system information, allowing to debug the source code.

```
# Create symbol file from the compiled kernel
user@stosys:~/src/msF2FS/$ objcopy --only-keep-debug vmlinux kernel.sym

# Run gdb with QEMU remote on port 1234
user@stosys:~/src/msF2FS/$ gdb
(gdb) file ./kernel.sym
(gdb) target remote :1234
(gdb) hbreak f2fs_allocate_data_block
(gdb) c
```

Listing C.2: Example of running QEMU with built kernel and gdb attached to QEMU VM for debugging, with a hardware-assisted breakpoint in F2FS.