ABSTRACT

The explosive growth of data has led to increased attention on NAND flash-based solid-state drives (SSDs), which offer high performance, low power consumption, and significant capacity per unit volume when compared to traditional hard-disk drives (HDDs). However, as NAND flash memory density continues to scale, modern SSDs suffer from what is known as *fail-slow* symptoms, and their performance degrades over time as they wear out. In this dissertation, we focus on understanding and addressing the performance, reliability, and sustainability challenges of modern flash-based storage systems by modeling key metrics, analyzing the design tradeoffs between these metrics, and optimizing existing storage I/O stack across various layers.

The first part of the dissertation presents our comprehensive study on SSD aging and fail-slow symptoms. Flash memory has a limited lifetime and suffers reduced reliability as the number of program/erase (P/E) cycles increases. To understand the aged behavior of SSDs, we first propose fast-forwardable SSD (FF-SSD), a machine learning-based SSD aging framework that generates representative future wear-out states without enduring prohibitive simulation times. FF-SSD incrementally builds lightweight regression models for flash blocks to capture the changes in SSD-internal states and predicts their trajectory using the information from past executions. With that, we further conduct an in-depth study on existing SSD lifetime extension algorithms and uncover that conventional wear leveling methods are far from perfect. Our finding challenges the established norms of SSD design and calls for a reevaluation of wear leveling's role.

The second part of the dissertation introduces the design and implementation of a capacity-variant storage system (CVSS) for modern SSDs. In the current capacity-invariant interface where the storage device exports a fixed capacity throughout its lifetime, an SSD has

no other choice but to trade performance for reliability as it wears out. Instead, CVSS redesigns the existing storage interface from a holistic perspective by exploiting trade-offs among capacity, performance, and reliability (CPR) and proposes a flexible capacity-variant interface that allows the SSD to maintain performance by gracefully reducing the number of exported blocks.

The third part of the dissertation proposes to improve the performance and sustainability of all-flash array (AFA) storage under heterogeneous device environments. The overall architecture of existing AFA systems is built around the assumption that the underlying storage components are homogeneous. This architecture results in significant disk under-utilization when considering heterogeneity among storage devices, particularly for modern SSDs. In response to these challenges, we present paRAID (placement asymmetric RAID), a heterogeneity-aware AFA system that transitions from traditional symmetric architectures to an asymmetric architecture, enabling the full utilization of all storage devices for optimized performance and sustainability.

TOWARDS THE NEXT GENERATION OF STORAGE STACK FOR NAND FLASH MEMORY-BASED SYSTEMS

by

Ziyang Jiao

B.E., Jilin Business and Technology College, 2019

Dissertation
Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer & Information Science & Engineering.

Syracuse University
May 2025

ACKNOWLEDGEMENT

This dissertation summarizes my multi-year research effort as a graduate student at Syracuse University (SU). This long march would not have been possible without the support and guidance from my advisor, friends, colleagues, and family.

First and foremost, I am deeply grateful to my advisor, Prof. Bryan Kim, for his invaluable mentorship throughout my Ph.D. journey. His dedication and guidance have significantly shaped my approach to research and academic growth. Under his supervision, I learned how to conduct impactful research, identify meaningful challenges, develop high-level perspectives, formulate research ideas, and refine academic presentations. Our brainstorming sessions and discussions were truly inspiring. His passion for research has been a driving force in my pursuit of a Ph.D., and his encouragement has strengthened my resilience in overcoming challenges. I would not have reached this milestone without his unwavering support.

I would also like to extend my sincere appreciation to my collaborators. Working with esteemed researchers from other institutions, including Prof. Jongmoo Choi, Prof. Jaeho Kim, and Prof. Janki Bhimani, has been an enriching experience, broadening my understanding of Computer Science. Additionally, I am fortunate to have worked alongside talented colleagues, including Xiangqun Zhang, Shao-peng Yang, Omkar Desai, Hojin Shin, and many others. The insightful discussions and exchanges with them have made my Ph.D. experience both stimulating and enjoyable.

Furthermore, I am thankful for the support of my proposal and dissertation defense committee members, Prof. Qinru Qiu, Prof. Wenliang Du, Prof. Reza Zafarani, and Prof. Joon S. Park. Their dedication and rigorous academic standards have provided an inspir-

ing example for my future career in academia. I also appreciate the faculty, staff, and students at Syracuse University for their continuous help.

Outside of SU, I am deeply grateful to my parents and sister, whose support has been a constant source of motivation. My parents have always trusted me and given me the freedom to make independent decisions, while my sister has stood by me, sharing her thoughts and emotions. Their belief in me has been invaluable.

Last but not least, I would like to express my heartfelt appreciation to my girlfriend for her love, patience, and companionship during this journey.

Thank you all for your support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction

The I/O stack is a fundamental component of storage system architecture, bridging the gap between applications and physical storage media. It consists of multiple layers, including the application layer, file system, block layer, device drivers, and firmware controller, each responsible for managing and optimizing data flow to and from storage devices. The stack must address challenges such as workload diversity, scalability, and hardware heterogeneity while maintaining efficient utilization of underlying storage media.

NAND flash-based solid-state drives (SSD) are an integral part of today's computing systems, used in mobile and embedded devices to large-scale data centers. In the context of flash-based storage systems, the I/O stack plays a crucial role in managing the unique properties of NAND flash, such as its erase-before-write requirement, limited endurance, and asymmetric read/write performance. Effective optimization of the I/O stack is critical to leveraging the full potential of modern storage systems.

1.1. Key Performance Metrics for Modern Storage Systems

The performance of a modern storage system is usually measured with four metrics: (1) throughput and latency, (2) reliability, (3) lifetime, and (4) sustainability.

- **Throughput and Latency.** (1) Throughput measures the average bandwidth of data processed per unit of time. High throughput is a primary objective for data-intensive workloads like video streaming and machine learning training. (2) Latency refers to the time required to complete a single I/O operation. Minimizing latency is essential for achieving responsive systems, especially in latency-sensitive applications such as online transaction processing (OLTP) and real-time analytics.

- **Reliability.** Reliability ensures the consistent and error-free operation of storage systems, even under adverse conditions. It is measured as the probability that data retrieved from the system is not the same as the originally stored data [162]. Key aspects include: (1) data integrity: protecting against bit errors and ensuring accurate data retrieval; and (2) fault tolerance: mitigating the impact of hardware failures through redundancy mechanisms such as redundant array of independent disks (RAID) and erasure coding.

- **Lifetime.** The lifetime of a storage system is influenced by the physical characteristics of storage media. Lifetime is measured as the amount of time from the point that a storage device is used to the point that the storage device is no longer usable. Techniques such as wear leveling, garbage collection, and write amplification reduction are employed to extend the lifespan of SSDs and maintain performance over time.

- **Sustainability.** With the growing emphasis on environmental impact, sustainability has become a critical metric for modern data-intensive storage systems. The carbon footprint of computing systems typically consists of embodied emissions from hardware manufacturing and infrastructure activities [111] and operational emissions from device usage.

1.2. Open Research Questions

Solid-State Drives (SSDs) have emerged as the dominant storage medium in both consumer and enterprise environments, driven by their exceptional performance and reliability. Unlike traditional Hard Disk Drives (HDDs), SSDs rely on NAND flash memory, which introduces unique research questions that directly influence the aforementioned performance metrics:

- **Performance consistency**: Although SSDs offer significantly lower latency and higher throughput than HDDs due to the absence of mechanical components, they suffer from severe performance variability. Compared to their HDD counterparts they are much faster, but their tail latency is more unstable due to background operations such as garbage collection, wear leveling, and error handling. Optimizing the storage stack for flash-specific challenges is crucial to achieving peak performance.

- **Aging resilience**: The drive for high storage density has caused the underlying hardware for the flash memory-based solid state drive to become increasingly unreliable [79], and the current state-of-the-art approaches implement performance-sacrificing techniques to prevent data loss, leading to fail-slow symptoms where the components continue to function but experience reduced performance [80]. With the decreasing endurance and reliability of modern SSDs [68], this problem becomes even more critical.

- **Lifetime management**: The finite program/erase (P/E) cycles of NAND flash necessitate sophisticated wear leveling and garbage collection strategies. Modern flash memory cells can withstand only a few hundred program-erase cycles, and wear leveling (WL) techniques have traditionally been used to distribute wear evenly across flash memory blocks. Recent study [65, 108] uncovers that conventional WL methods are far from perfect: they are not only limited in effectiveness but can also produce counter-productive results in modern SSDs.

- **Sustainability under heterogeneity**: Modern SSDs exhibit significantly greater heterogeneity compared to traditional hard disk drives. However, the overall architecture of existing all-flash array systems is built around the assumption that the underlying storage components are homogeneous [66]. This mismatch leads to inefficiencies and under-utilization, exacerbating the carbon footprint of storage systems.

1.3. Research Goal and Philosophy

The evolution of storage technologies, particularly in NAND flash-based solid-state drives (SSDs) and emerging storage devices, has introduced unprecedented opportunities for improving performance, reliability, and efficiency in modern computing systems. However, the existing system software stack—-ranging from the block I/O subsystem and device drivers to the file system and RAID layers—-has not kept pace with these hardware advancements. This misalignment creates a significant barrier to fully exploiting the capabilities of modern storage devices, resulting in significant under-utilization and performance loss in storage systems.

This dissertation is grounded in the research philosophy that **a small amount of the right information, strategically infused across system layers, can unlock disproportionate performance benefits**. Rather than exposing the full device complexity to the software layers, my approach advocates for a lightweight, information-driven methodology. Specifically, my research goal is to selectively **imbuing minimal yet meaningful knowledge about the underlying storage devices into various layers of the I/O stack**, thereby enabling more efficient, adaptive, and high-performance storage system behavior.

This research philosophy is reflected throughout the dissertation in the design of various layers within the storage stack (shown in Figure 1):

- **At the device level**, the *Fast-Forwardable SSD (FF-SSD)* aging framework leverages machine learning to predict future wear-out states, providing insights that guide upper-layer decisions. Furthermore, an in-depth analysis of wear leveling technique reveals inefficiencies in current implementations and challenges the traditional fixed-capacity abstraction of storage devices.

- **At the file system level**, the *Capacity-Variant Storage System (CVSS)* incorporates awareness of device aging, allowing it to dynamically adjust the logical capacity exposed to the system. This adaptation helps mitigate performance degradation and addresses fail-slow symptoms, which are commonly observed in traditional flash-based storage systems.

- **At the RAID layer**, the *Placement-Asymmetric RAID (paRAID)* introduces a heterogeneity-aware architecture that considers both device- and data-level characteristics. This design enables more efficient utilization of underlying SSDs by shifting away from conventional, symmetric RAID architectures.

By aligning storage design more closely with hardware characteristics without introducing excessive complexity, my research aims to enable a more efficient, synergistic, and adaptive storage ecosystem.

1.4. Contributions

This thesis makes the following contributions.

- **New SSD aging framework:** We build *Fast-Forwardable SSD (FF-SSD)*, to the best of our knowledge, the first ML-based SSD aging framework that generates representative future wear-out states. We evaluate our design using real-world workloads across multiple platforms to demonstrate the usefulness of FF-SSD for SSD aging. FF-SSD is accurate (up to 99% similarity), efficient (accelerates simulation time by 2×), and modular (can be integrated with existing simulators and emulators).

- **New understandings of wear leveling:** We uncover that wear leveling in SSDs does more harm than good under modern settings where the endurance limit is in the hundreds. To support this claim, we systematically evaluate existing wear leveling techniques and show that they exhibit anomalous behaviors and produce a high

write amplification. These findings are consistent with a recent large-scale field study on millions of SSDs. We discuss the option of forgoing wear leveling and instead adopting capacity variance in SSDs, and show that the capacity variance extends the lifetime by up to 2.94×.

- **New interface for flash storage:** We rethink the existing storage interface from a holistic perspective by exploiting tradeoffs among capacity, performance, and reliability (CPR) and designing a flexible capacity-variant interface that allows the SSD to maintain performance while it gracefully reduces the number of exported blocks. CVSS provides a practical solution to the growing challenges of flash storage reliability and performance stability.

- **New RAID architecture for sustainability:** We study and identify the inefficiency of conventional RAID solutions when considering disk heterogeneity for modern SSDs through real system deployment. We present paRAID (placement asymmetric RAID), to our knowledge, the first RAID architecture designed to leverage a mix of heterogeneous SSDs to improve overall system performance and sustainability by distributing data asymmetrically.

1.5. Organization of Dissertation

As shown in Figure 1, the remainder of the dissertation is organized as follows. Chapter 2 provides the necessary background of SSDs and flash-based storage systems. Chapter 3 proposes a learning-based SSD aging framework that generates representative future wear-out states. Chapter 4 analyzes popular SSD lifetime management algorithms and uncovers the inefficiencies and unintended drawbacks of wear leveling algorithms in today's environments. Chapter 5 presents the design and implementation of a capacity-variant storage system. CVSS addresses the fail-slow symptoms in solid-state drives by developing a soft-

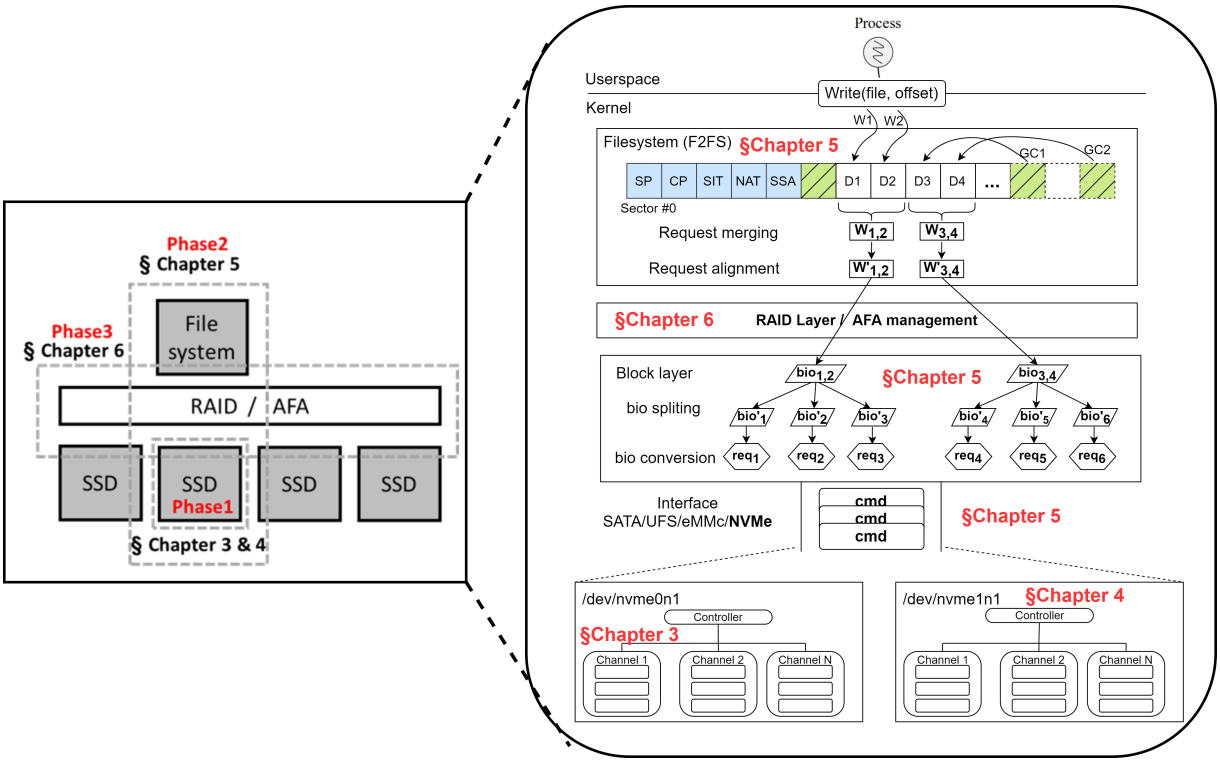


Figure 1: Overview of research contribution and activities.

ware/hardware co-design solution. Chapter 6 introduces paRAID, a heterogeneity-aware RAID architecture for all-flash-array systems that optimizes storage utilization and sustainability. Chapter 7 summarizes the thesis, and Chapter 8 presents future research directions.

1.6. Bibliographic Notes

Most of the research work appearing in this dissertation has been published at various venues and has appeared in the publications listed below.

Works related to the thesis:

• **Ziyang Jiao**, Omkar Desai, Jaeho Kim, Jongmoo Choi, and Bryan S. Kim. "paRAID: A Sustainable Storage Architecture with Heterogeneous SSDs." *Submitted to Anonymized Conference on File and Storage Systems, 2026 (Under review).*

• **Ziyang Jiao** and Bryan S. Kim. "Asymmetric RAID: Rethinking RAID for SSD Heterogeneity." *In ACM Workshop on Hot Topics in Storage and File Systems, 2024.*

• **Ziyang Jiao**, Xiangqun Zhang, Hojin Shin, Jongmoo Choi, and Bryan S. Kim. "The Design and Implementation of a Capacity-Variant Storage System." *In USENIX Conference on File and Storage Technologies, 2024.*

• **Ziyang Jiao**, Janki Bhimani, and Bryan S. Kim. "Wear Leveling in SSDs Considered Harmful." *In ACM Workshop on Hot Topics in Storage and File Systems, 2022 (Best Paper Award).*

• **Ziyang Jiao** and Bryan S. Kim. "Generating Realistic Wear Distributions for SSDs." *In ACM Workshop on Hot Topics in Storage and File Systems, 2022.*

• **Ziyang Jiao** and Bryan S. Kim. "The Fast-Forwardable SSD Aging Framework." *In USENIX Conference on File and Storage Technologies (WiP), 2022.*

Other related contributions:

• Omkar Desai, **Ziyang Jiao**, Shuyi Pei, Janki Bhimani, and Bryan S. Kim. "Preparation Meets Opportunity: Enhancing Data Preprocessing for ML Training With Seneca." *Submitted to Anonymized Conference on File and Storage Systems, 2026 (Under review).*

• Xiangqun Zhang, **Ziyang Jiao**, and Bryan S. Kim. "ByteZ: When ZNS Meets Byte Interface." *Submitted to Anonymized Workshop on Hot Topics in Storage and File Systems, 2025 (Under review).*

• Xiangqun Zhang, **Ziyang Jiao**, Farzana Rahman, and Bryan S. Kim. "Filling in the Missing Piece: Integrating Storage into CompOrg Courses." *In American Society for Engineering Education (ASEE) Annual Conference and Exposition, 2025.*

CHAPTER 2

Solid State Drive Basics

In this chapter, we highlight important concepts to facilitate a better understanding of the rest of the dissertation. We first discuss the basics of a NAND flash cell, the building block of almost every solid state drive, and then provide a summary of how SSDs work.

2.1. Basic Flash Operations

Flash memory operations involve three fundamental low-level commands: **Read**, **Erase**, and **Program**. These operations dictate how data is accessed and modified on the flash chip.

2.1.1. Read (a Page)

The host can read any page (typically 4KiB) by issuing a read command with the appropriate page number. This operation is generally fast, taking tens of microseconds, regardless of the page's location. Unlike traditional disk storage, flash memory provides uniform random access.

2.1.2. Erase (a Block)

Before writing to a page, the entire block containing that page must first be erased. The erase operation resets all bits in the block to 1, effectively destroying any existing data. Therefore, any important data within the block must be backed up before issuing an erase command. This operation is relatively slow, typically taking a few milliseconds. Each flash memory block can only endure a fixed number of erase operations.

2.1.3. Program (a Page)

Once a block has been erased, a `program` command can be issued to write data to a specific page. This process modifies certain bits from 1 to 0 as required. Programming a page is faster than erasing a block but slower than reading a page, typically taking hundreds of microseconds.

These three operations define the core behavior of flash memory and influence storage performance and management strategies.

2.2. SSD Architecture

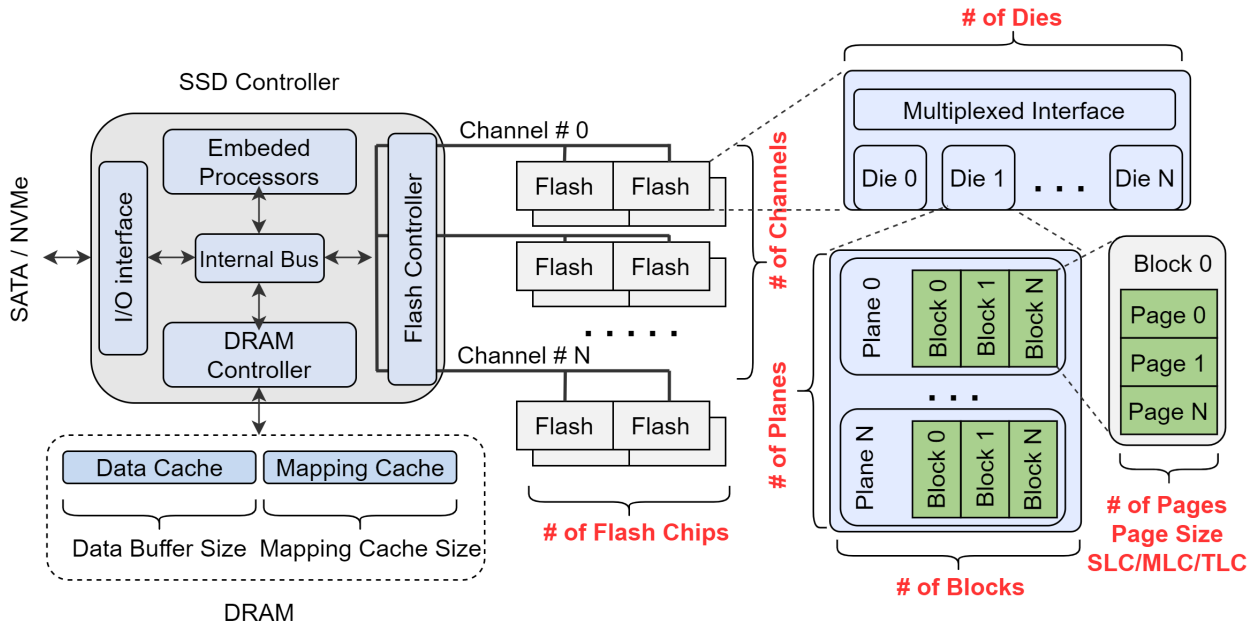


Figure 2: An overview of SSD architecture.

Figure 2 shows an overview of SSD. An SSD NAND package is structured into dies, planes, blocks, and pages. Due to the **erase-before-write** characteristic of NAND flash, data cannot be directly overwritten. Erase operations occur at the granularity of erase blocks (EBs), which range from tens to hundreds of megabytes, whereas write operations are performed at the page level (e.g., 16KiB, 48KiB, 64KiB).

To manage overwrites, the Flash Translation Layer (FTL) follows a three-step process: (1) writing the updated data to a free page, (2) marking the previous data as invalid, and (3) updating metadata to reference the new location. This metadata maps logical addresses to physical addresses in NAND storage. Over time, writing to logical addresses generates invalid pages, which must be reclaimed through a process known as Garbage Collection (GC).

2.3. Flash Translation Layer

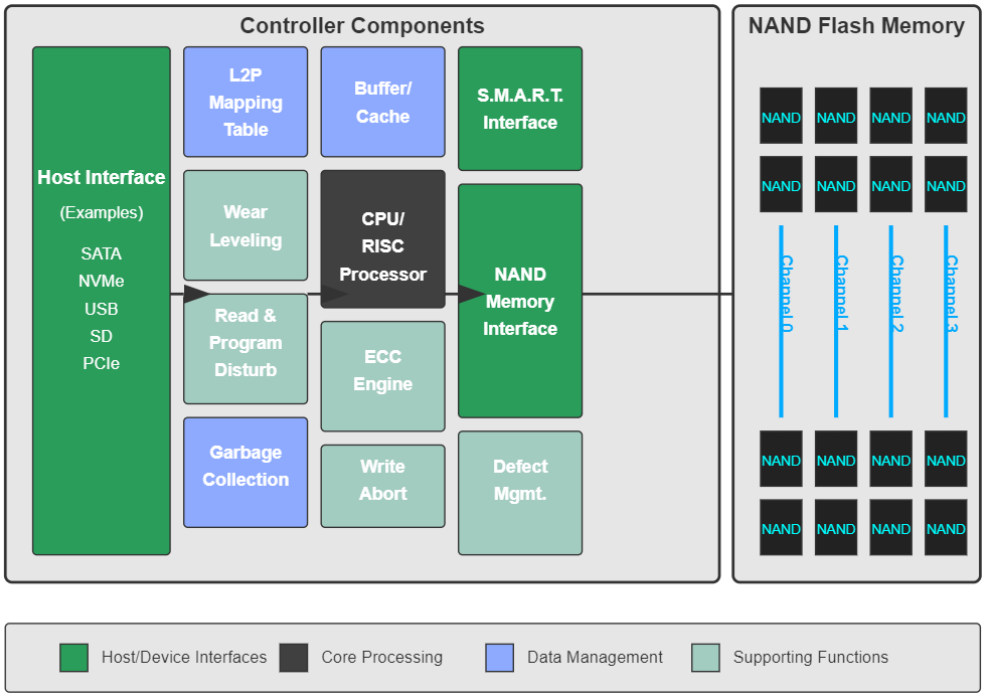


Figure 3: Generic solid-state drive (SSD) controller architecture.

This section focuses on the key components of a typical SSD controller and how it interfaces with the NAND flash memory [13]. Figure 3 illustrates the fundamental components of an SSD. A brief description for each key block is provided below.

2.3.1. L2P Mapping Table

The host accesses the SSD through Logical Block Address (LBA). Each LBA corresponds to a sector, typically 512 bytes in size. Within the SSD, the flash page serves as the basic unit for accessing flash memory between the controller and the flash memory. Every time a write request is issued, the SSD controller allocates a physical flash memory page to store the host data, and this mapping is recorded within the SSD. This mapping ensures that when the host needs to read data, the SSD knows where to retrieve the corresponding data from the flash memory.

2.3.2. Host Interface

The SSD controller's host interface is typically built to comply with established industry standards. Different interfaces are designed to meet varying system architectures and performance needs. Some of the most widely used interfaces include SATA, SD, USB, PATA/IDE, and PCIe.

2.3.3. SMART (Self-Monitoring, Analysis, and Reporting Technology)

Modern SSD controllers typically incorporate the S.M.A.R.T. [124] function, which tracks and logs various performance and health metrics of the SSD and its memory. One key feature is the ability to monitor the remaining percentage of endurance cycles, a crucial factor in estimating the SSD's remaining lifespan.

2.3.4. Wear Leveling

As shown in Figure 4, wear leveling is a technique used to distribute write cycles evenly across the available NAND blocks. Since each NAND block has a finite number of program/erase (P/E) cycles, repeatedly writing to a single block would rapidly exhaust its endurance. To prevent premature wear, the SSD controller employs a wear leveling algorithm that monitors and balances write operations across different physical NAND blocks.

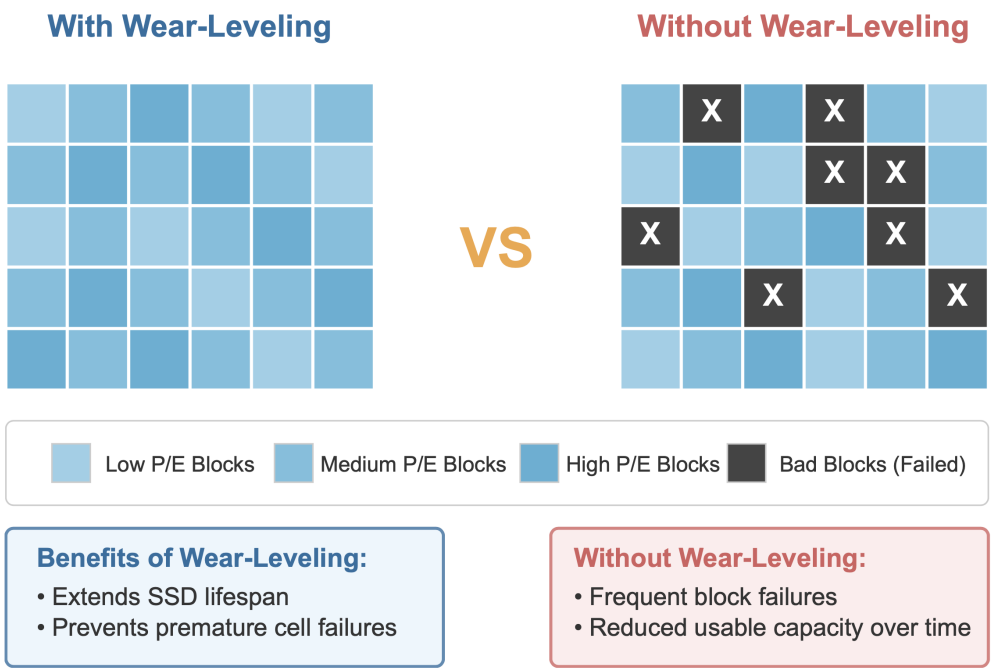


Figure 4: NAND Flash Wear-Leveling Comparison.

2.3.5. Read and Program Disturb

As NAND flash technology advances with increasingly smaller trace widths, maintaining data integrity becomes more challenging. Read and Program Disturb occur when read or write operations induce unintended electrical interference in adjacent cells, potentially altering their stored values. To mitigate this effect, SSD controllers employ specialized algorithms and, in some cases, additional circuitry to ensure data reliability.

2.3.6. Buffer/Cache

SSD controllers typically incorporate a high-speed SRAM/DRAM cache buffer to temporarily store read and write data, improving performance and efficiency. However, since this cache relies on volatile memory, data can be lost if power is unexpectedly interrupted. It is common to have both internal caches within the controller chip and external RAM cache chips to improve performance.

2.3.7. CPU/RISC Processor

At the core of every SSD controller is its primary processing unit, which can be either a CPU or an RISC processor. The processor plays a crucial role in determining the overall efficiency and functionality of the SSD.

2.3.8. ECC Engine

Error Correction Code (ECC) is a key component in modern SSDs to detect and correct errors in stored data. ECC algorithms can recover a specific number of corrupted bits within each data block, ensuring data integrity. Without ECC, the use of low-cost consumer flash storage, which relies on less reliable memory, would not be feasible.

2.3.9. Write Abort

Write Abort occurs when power is unexpectedly lost during a write operation to NAND flash. Without a backup power source, such as a battery or SuperCap-supported cache, data in transit is at risk of being lost. More critically, it is essential to protect the SSD's internal metadata and firmware from corruption. To address this, Write Abort circuitry is commonly implemented in enterprise-grade SSDs, ensuring data integrity and system reliability.

2.3.10. Defect Management

Each SSD requires a strategy to handle faulty memory blocks and emerging defects. When NAND blocks are no longer functional, the SSD controller must take corrective action. In certain instances, a spare sector may be used to substitute the defective block. In cases of inadequate controller design, the SSD may fail. Different controllers may employ different approaches to manage bad flash memory blocks.

2.4. Garbage Collection and Write Amplification

Garbage Collection (GC) is initiated when the SSD has a limited number of free blocks available (i.e., lower than its predefined threshold). During this process, valid pages from an erase block are relocated to a new location, allowing the now completely invalid block to be returned to the free pool. Once in the free pool, an erase block can be erased and repurposed for incoming data writes.

Since garbage collection involves reading, migrating, and erasing data, it is a resource-intensive operation. The energy consumption of an SSD is directly influenced by the frequency and duration of garbage collection activities, making it a critical factor in SSD performance and longevity.

2.4.1. Device-Level Write Amplification (DLWA)

Device-Level Write Amplification (DLWA) measures the ratio of data written internally within the SSD to the data actually written by the host. It quantifies the additional write overhead caused by internal SSD operations such as Garbage Collection. DLWA is calculated using the formula [3]:

$$DLWA = \frac{\text{Total NAND Writes}}{\text{Total SSD Writes}} \tag{2.1}$$

2.4.2. Application-Level Write Amplification (ALWA)

Application-Level Write Amplification (ALWA) represents the ratio of data written to the SSD to the actual data written by the application. It accounts for file system and system-level write overhead before reaching the SSD. ALWA is given by [3]:

$$ALWA = \frac{\text{Total SSD Writes}}{\text{Total Application Writes}} \tag{2.2}$$

The additional read and write operations caused by garbage collection can interfere with the execution of other SSD commands. In addition, the increased NAND operations from data relocation accelerate wear on the NAND media, reducing its lifetime.

For instance, a DLWA of 3 means that for every 4KiB of user data written, the Flash Translation Layer (FTL) performs an additional 8KiB write due to garbage collection. Since NAND flash has a limited number of Program and Erase (P/E) cycles, a DLWA of 3 significantly shortens the SSD's lifespan to one-third.

Device-Level Write Amplification significantly affects various SSD performance metrics, including Quality of Service (QoS), bandwidth, endurance, reliability, and sustainability. Due to its broad impact, DLWA is commonly used as a key indicator for monitoring SSD efficiency.

CHAPTER 3

Generating Realistic Wear Distributions for SSDs

3.1. Introduction

Understanding the aged behavior of SSDs (solid-state drives) is important because the errors in SSDs increase over time as flash memory wears out [14, 115]. Errors not only corrupt the data the SSD stores (silent data corruption) [8, 41], but also induce fail-slow symptoms where performance degrades as the SSD attempts to correct and prevent these errors [46, 79].

However, no existing SSD development frameworks (such as Amber [44], FEMU [95], and MQSim [156]) consider aging in their design. Aging through pre-conditioning is prohibitively expensive as it takes years' worth of simulation time to reach that aged state. Alternatively, the initial erase count can be pre-set to a higher non-zero value, but this will be an unrealistic wear state because modern SSDs cannot effectively even the wear with its wear leveler [108].

In this work, we propose *Fast-Forwardable SSD* (`FF-SSD`), a machine learning-based SSD aging framework that generates representative future wear-out states. Within a typical SSD model, many components perform repetitive work. For example, the garbage collector may keep selecting several hot blocks as the victim over a period of time. Similarly, the trigger condition of the wear leveler is computed regularly during the lifetime of the SSD. Thus, the behavior within an SSD can be learned from past executions to predict the future SSD-internal state.

However, the challenges of using a machine learning approach for making online, fine-grained inferences on SSD internal states are two-fold. First, the inference must be ac-

curate. Modern SSDs are complex embedded systems, managing all of their internal resources with background operations such as garbage collection, wear leveling, error handling, and data scrubbing. These internal complexities need to be learned to make the inference highly accurate. Second, the inference must be fast and efficient relative to the simulation time; otherwise, it either brings negligible benefits or even prolongs the overall process. Deep learning models like convolutional neural network or recurrent neural network would introduce more complexities and may result in a slow training and inference performance. To address these, `FF-SSD` incrementally builds a lightweight regression model for each block to capture the changes in SSD-internal states and predicts their trajectory using the information from past executions. This model would approximate the future wear state of an SSD device if the same workload were to be repeated, resulting in a faster simulation time.

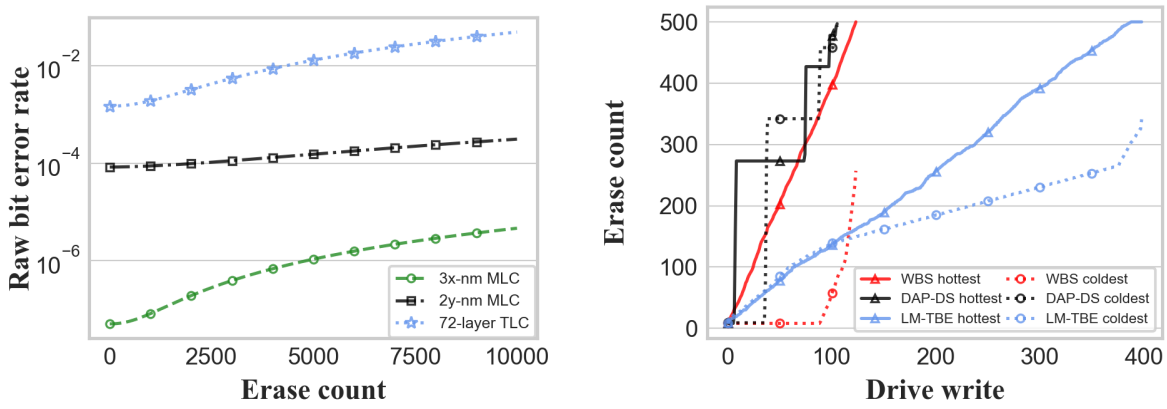
We present the design, implementation, and usage scenarios of the `FF-SSD` framework[1]. We build `FF-SSD` with the following quality attributes: (1) *accuracy* by generating realistic distributions that match up to 99% of the full simulation results, (2) *efficiency* by accelerating the simulation by 2× to reach a desired aged state, and (3) *modularity* by building the prediction module that can be integrated across multiple platforms. We evaluate our design using real-world workloads [78, 91, 172] across multiple platforms [34, 44, 95] to demonstrate the usefulness of `FF-SSD` for SSD aging.

3.2. Motivation and Related Works

We begin by asking a basic question: does the wear state really matter in SSDs? We then describe the irregularity of the SSD's wear, and briefly discuss related works.

Fail-slow symptoms. Previous works [46, 79] have shown the fail-slow symptoms mani-

[1] `FF-SSD` is available at https://github.com/ZiyangJiao/FF-SSD.



(a) The relationship between RBER and erase. (b) The irregularity of block erase.

Figure 5: Figure 5a shows the error rate as the erase count increases for three flash memory chips. Figure 5b shows the changes in erase count for the hottest block and the coldest block under three real-work workloads [78], WBS, DAP-DS, and LM-TBE, with a 256GB SSD. The end of the line indicates the failure of that device.

fest in SSDs as they correct and prevent errors whose rate naturally increases due to wear. Figure 5a shows the relationship between raw bit error rate (RBER) and erase count for three flash memory chips derived from RBER models [79]. As an SSD ages and wears out, the error rate increases, which in turn, degrades the performance [79]. The bit error rate caused by wear-out degrades the performance of the drive. However, it is not only I/O writes that cause programs and erases. Any SSD-internal housekeeping such as garbage collection and wear leveling perform additional erases that wear out the device [79]. Flash memory manufacturers specify how many times a flash memory block can be erased before turning bad [130], also known as endurance limit or P/E cycles. This is because writing to an SSD induces programs (and eventually erases) that weaken the non-volatile memory, leading to a wear-out. An SSD typically maps out these blocks so that they will no longer be used [115, 130]. When too many blocks become bad, the SSD cannot maintain the logical capacity, and the entire storage device becomes unusable.

Irregularity of block erase count. An SSD consists of hundreds of thousands of flash blocks and they have different erase count trajectories during the lifetime of the device.

Figure 5b shows the changes in erase count for the hottest and coldest block under three real-work workloads [78] (WBS, DAP-DS, and LM-TBE), for a 256GiB SSD. The end of the line indicates the failure of that device. As shown in the figure, blocks behave distinctly across three workloads, and the erase counts change at different rates through SSD's lifetime, demonstrating the irregularity of erasure behavior within the SSD. Even though a wear leveler is used [25], the SSD cannot implement perfect wear leveling, similar to the observation from a field study on millions of SSDs [108].

Inferring the overall wear distribution within an SSD is hard, given the internal intricacy of SSDs. Any SSD-internal activities explicitly or implicitly increase the wear. For example, garbage collection will be triggered when the amount of free space decreases to a predefined threshold. A batch of victim blocks will be erased depending on the GC policy. The greedy approach [168] targets the blocks containing minimum amount of valid data, while the cost-benefit approach [140] selects the ones with the highest cost-benefit value. Wear leveling, on the other hand, introduces additional block erasure by performing data relocation to equalize the erase count. SSD reliability enhancement techniques such as internal redundancy, data scrubbing can also accelerate the wear.

File system aging. Although there are existing research and tools for file system aging [1, 73, 148], these cannot be directly applied to SSD aging. File system aging tools generate a fragmented state of logical block layouts, but SSD aging needs to model the physical aging of blocks as the reliability properties of a young and an old block are different. Preconditioning an SSD is more akin to file system aging by populating and invalidating the address space [149], and cannot sufficiently age the device to an end-of-life state.

Machine learning for simulation. We find two prior works that use machine learning to accelerate simulations: DEVS (Discrete EVent Specification) [142] and CML (Continu-

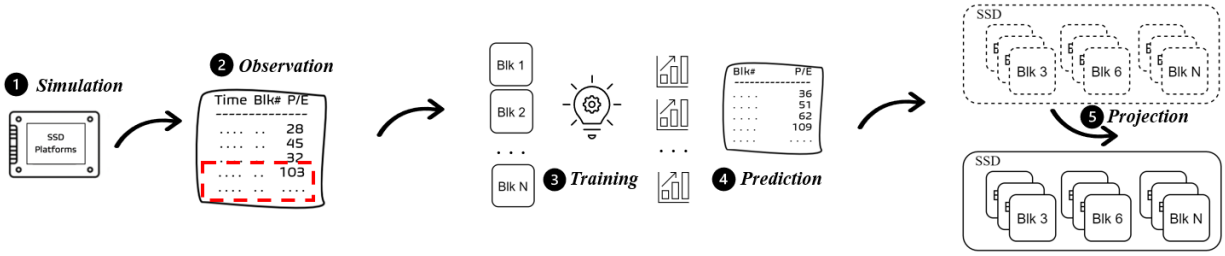


Figure 6: FF-SSD overview. FF-SSD starts from actual (1) simulation to (2) observe the SSD's internal activities, then (3) train lightweight regression models to (4) predict the trajectory for the block's wear out, and (5) project SSD states based on the prediction results.

ous Machine Learning) [135]. At a high-level, DEVS considers multiple model candidates and selects the best one for prediction. This model, however, is not always updated before each prediction stage. On the other hand, CML continuously incorporates the latest data to update its model. However, both are not designed for SSD aging and ignore the complexity of modern SSDs. CML focuses on performance estimation rather than generating internal states, and DEVS is for discrete-event modeling and simulation that assumes that in between events (i.e., external I/O requests), the state of the system does not change [157].

3.3. System Design

We first describe the overall process of learning the wear-out behavior of the SSD to predict the future state, and discuss further optimizations to improve the efficiency of our tool.

3.3.1. Overall Architecture

Our generative process consists of five phases: (1-2) simulation and observation, (3) training, (4) prediction, and (5) projection, as shown in Figure 6.

Simulation and observation. FF-SSD starts from actual simulation during a workload sample to observe the SSD's internal activities, and collect information that will later be

used for the prediction. To infer the future erase count of each block, the following features will be recorded at the end of every observation period: (1) the block identifiers, (2) the current erase count, for each block associated with its identifier (3) the observation time, and (4) the write amplification factor during this observation period. We filter out features that may seemingly look important but in reality, are not for the prediction. These excluded features relate to the workload characteristics observed from the host side such as workload hotness, access footprint, and I/O size, as their effect on erase count heavily depends on the implementation of the flash translation layer (FTL). The selected features are logged throughout the simulation.

Training. Once enough workload has been sampled, we use the gathered data to build a lightweight regression model for each block and predict the future state. Specifically, the system will extract and normalize the time-series data from the observation and train the model for each block. The main challenge here is to decide how much history should be used for the training process. Since changes in wear for each block are dynamic during the SSD's lifetime, most recent activities are more relevant to the future states. We thus filter out stale information and use only the latest history. These models are updated before each prediction, so they can learn the most recent trend within the SSD.

Prediction. The training phase generates the weights in the models that will be temporarily kept in `FF-SSD`. For each block, the model is then activated to infer its future erase count based on the *acceleration factor* (AF), dictated by the user. In this work, we define AF as the ratio of full workloads to the simulated workloads, which can be computed using formula:

$$AF = \frac{simulated + predicted}{simulated} \tag{3.1}$$

The AF ranges from 1 to ∞, where $AF = 1$ means all workloads are handled by the underlying platform and $A = \infty$ means there is no actual simulation involved. Thus, there is an inherent tradeoff between prediction accuracy and acceleration efficiency. An aggressive acceleration saves more time to reach an aged state, but at the cost of decreased accuracy compared to moderate acceleration.

Projection. Finally, these generated future states are fed back into the simulation, where it will continue to sample and run workloads. The relevant components are also updated within the platform based on the new state. These stages will repeat until the SSD reaches its desired age.

3.3.2. Enhancing Efficiency

Since we approach by incrementally building multiple lightweight regression models for all blocks, the prediction overhead is proportional to the number of blocks. Thus, reducing the number of blocks to model and infer would improve the efficiency. We next present an analytic approach to further improve inference efficiency based on distribution modeling.

We assume that the wear distribution of blocks adheres to an underlying measurable distribution $\rho(\cdot)$, such as normal distribution. We then divide all blocks into several groups based on wear conditions (e.g., quantiles) and build one model for each group. Then we estimate the future wear for each block according to the prediction result of these groups and the density function that models the overall underlying distribution.

In this work, we observed that the discrete wear distribution can be almost perfectly matched by a skew-normal distribution in most cases, with skewness α, location μ, and scale parameter σ. Figure 7 shows the histogram of erase counts under WBS workloads, overlaid with the approximating skew-normal distribution with $\alpha = 0.75, \mu = 310, \sigma = 15.1$. Since no statistical method can affirm whether two distributions are the same, we run Kol-

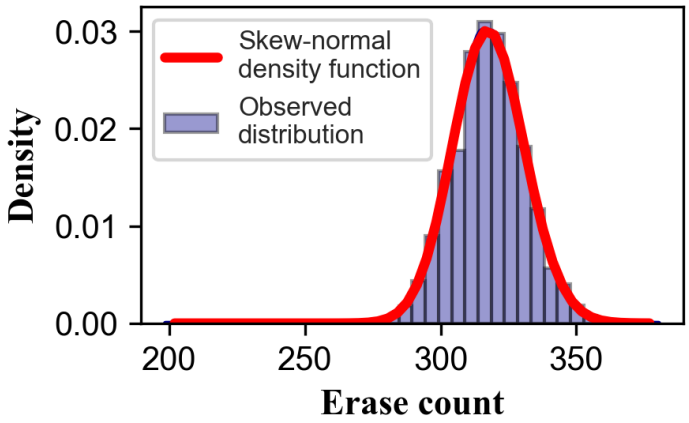


Figure 7: Skew norm fit to the measured distribution. The red line indicates the approximating skew-normal distribution, matching with the observed distribution.

mogorov–Smirnov goodness-of-fit test to check if the observed distribution fits our statistical model. We fail to reject the null hypothesis that the wear distribution matches the skew-normal distribution on 10^5 samples with $p > 0.1$.

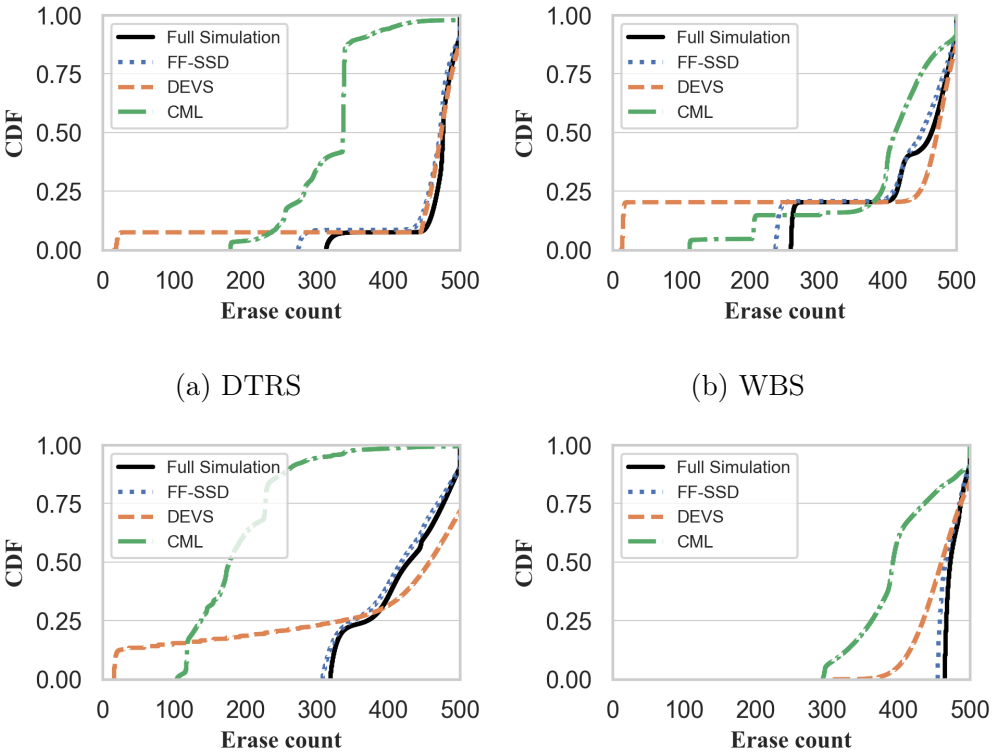
3.4. Evaluation

In this section, we first evaluate the effectiveness of `FF-SSD` for SSD aging, then explore the optimal point along the tradeoff between accuracy and efficiency. Table 5 outlines the system configurations for our evaluation (FTLSim [34], Amber [44], and FEMU [95]).

For the workload, YCSB-A is from running YCSB [172], VDI is from a virtual desktop infrastructure [91], and the remaining workloads are from Microsoft production servers and Microsoft enterprise servers [78]. The traces are modified to fit the logical capacity of the SSD, and all the requests are aligned to 4KiB boundaries. We use a fully simulated result ($AF = 1$) as the baseline and compare our work against two prior works, DEVS [142] and CML [135].

Table 1: Platform specific configurations.

PC platform			
CPU name	i7-9700	Frequency	3.00GHz
Core number	8	Memory	32GiB
OS	Ubuntu	ISA	X86_64
FTLSim			
Page per block	256	Physical capacity	284GiB
Page size	4KiB	Logical capacity	256GiB
Endurance limit	500	Over-provisioning	0.11
Wear leveling	PWL [25]	Garbage collection	Greedy
Amber			
Channels	8	Page size	4KiB
Packages per channel	4	Physical capacity	284GiB
Die per package	2	Logical capacity	256GiB
Plane per die	2	Over-provisioning	0.11
Block per plane	1136	Garbage collection	Greedy
Pages per block	512	Wear leveling	Var-based
FEMU			
Channels	8	Page size	4KiB
Luns per channel	8	Physical capacity	16GiB
Planes per lun	1	Logical capacity	15GiB
Blocks per plane	256	Over-provisioning	0.07
Pages per block	256	Garbage collection	Greedy



(a) DTRS

(b) WBS

(c) YCSB-A

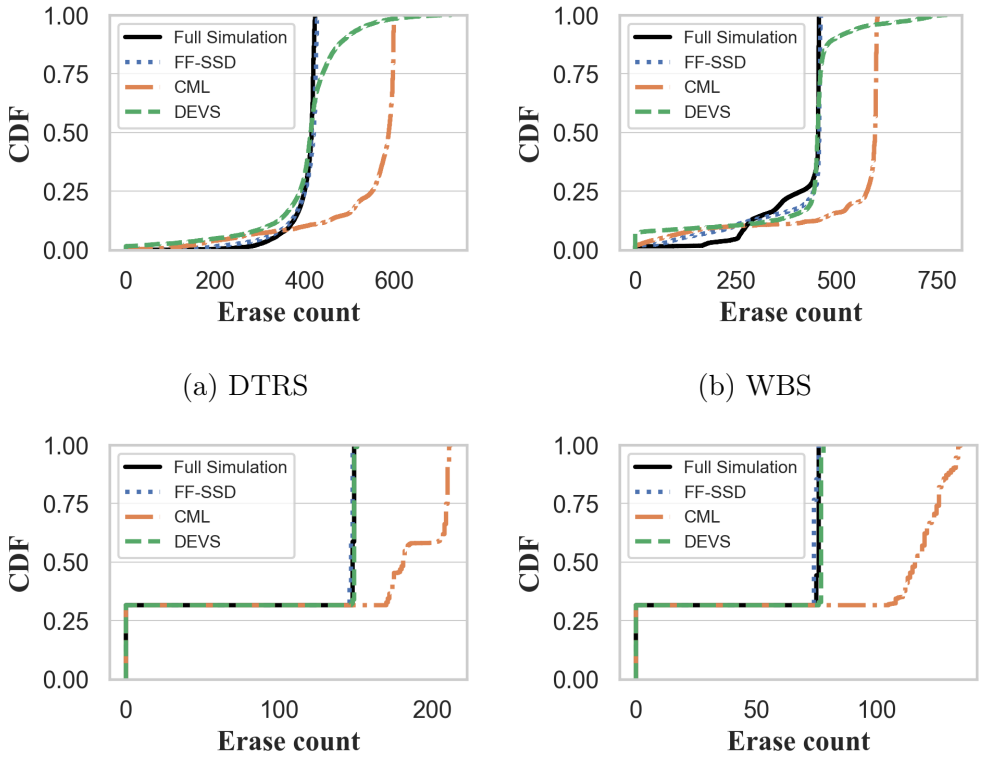
(d) VDI

Figure 8: SSD aging until failure on FTLSim. `FF-SSD` achieves the highest accuracy (91% – 97%) compared to DEVS (60% – 93%) and CML (48% – 84%). The accuracy is computed using the mean difference in erase counts across all blocks relative to their real values from the full simulation.

3.4.1. Effectiveness of FF-SSD

FTLSim

We first examine the effectiveness of `FF-SSD` on FTLSim. We configure a 256GiB SSD and set the acceleration factor (AF) to be 1.5. We set the endurance limit to 500 for each block and run until the number of bad blocks exceeds its over-provisioning. In this work, the accuracy is computed using the mean difference in erase counts across all blocks relative to their real values from the full simulation. Figure 8 shows our experiment results: with $AF = 1.5$, `FF-SSD` continuously learns the behavior within the SSD using the infor-



(a) DTRS (b) WBS

(c) DAP-DS (d) RAD-AS

Figure 9: SSD aging with 600 iterations of the workloads on Amber. The accuracy achieved by `FF-SSD` (88% – 99%) outperforms DEVS (83% – 99%) by a small margin but by a large margin for CML (40% – 60%).

mation from the past two iterations of the workload, and then predicts the wear state after one additional iteration. `FF-SSD` generates the final states of SSD, and achieves the highest accuracy compared to DEVS and CML.

For `FF-SSD`, we observe that the average accuracy is 94% across all workloads, and as high as 97% for VDI. On the other hand, the overall accuracy for DEVS and CML ranges from 60%–93% and 48%—83%, respectively. The lowest accuracy for `FF-SSD` is 91% under YCSB workloads, while DEVS and CML only achieve 60% and 48%, underperforming our design by a large margin. Essentially, we accelerate the simulation by 50% with this workload sampling configuration and the predicted distribution closely follows the fully

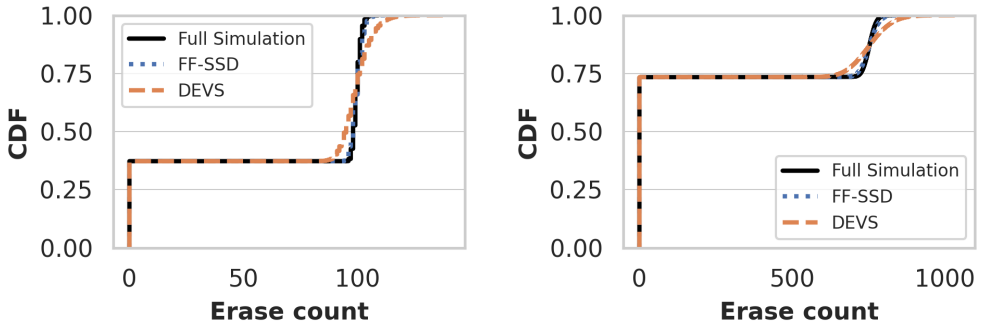
simulated result.

Amber

We next study the performance of `FF-SSD` on Amber and apply a more aggressive acceleration factor. Specifically, we generate the wear distributions by applying 600 iterations of the workloads with $AF = 2$ and compare them with the baseline. However, the endurance limit is set to be sufficiently large at 100,000, the default value of Amber.

Figure 9 shows the experiment results on Amber. We observe that `FF-SSD` outperforms DEVS and CML for all the workloads we evaluated. The accuracy ranges from 88% (for WBS) to 99% (for DAP-PS), with a mean of 93%. On the other hand, the accuracy achieved by CML only ranges from 40% (for RAD-AS) to 60% (for WBS), with a mean accuracy of 49%. DEVS presents different behavior on Amber compared to its results on FTLSim. DEVS delivers a similar accuracy with `FF-SSD` on Amber, ranging from 83% (for DTRS) to 99% (for DAP-PS), with a mean accuracy of 91%.

To further investigate why DEVS performs well on Amber while not on FTLSim, we find that apart from the implementation details of FTLSim and Amber, these two platforms deployed different wear leveling policies. PWL [25] is used in FTLSim, which adopts an adaptive threshold-driven approach for selecting victim blocks for erases; on the other hand, Amber applies a predefined threshold for the *uneven factor* to determine the trigger condition. We run another experiment to study how the wear leveling policy affects the inference accuracy. Figure 10 shows the result on FTLSim after applying 600 iterations of DTRS and WBS workloads without wear leveling. Without wear leveling in FTLSim, DEVS performs similarly to `FF-SSD`, indicating that the performance of DEVS is sensitive to the FTL algorithm.



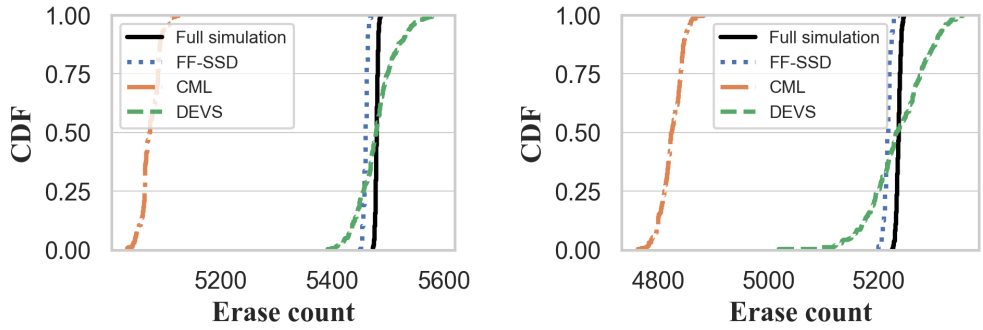
(a) DTRS (b) WBS

Figure 10: Performance comparison of `FF-SSD` and DEVS on FTLSim without WL. DEVS presents a similar performance to `FF-SSD`.

FEMU

The previous experiments show the performance of `FF-SSD` on SSD simulators. To further improve usability, we turn to study the effectiveness of `FF-SSD` on FEMU, a state-of-the-art SSD emulator. We use FEMU to emulate a 16GiB SSD and generate the wear distribution after 50 iterations of two Microsoft enterprise server workloads, ME-TPCC and ME-TPCE [78]. The acceleration factor is set to be 2, and all I/O requests are issued by btreplay to the underlying SSD.

Figure 11 shows the experiment results on FEMU. `FF-SSD` achieves the highest accuracy (93% for TPCC and 91% for TPCE) compared to DEVS (79% for TPCC and 60% for TPCE) and CML (18% for TPCC and 11% for TPCE). Moreover, the wear states generated by `FF-SSD` align with the overall shape of our baselines, while DEVS and CML present apparent discrepancies. Comparing this to the previous results, we find that CML underperforms other methods when applied to SSD aging and DEVS only works well under a particular configuration. On the other hand, `FF-SSD` generates a more realistic distribution on all the platforms while accelerating the emulation by 2×.



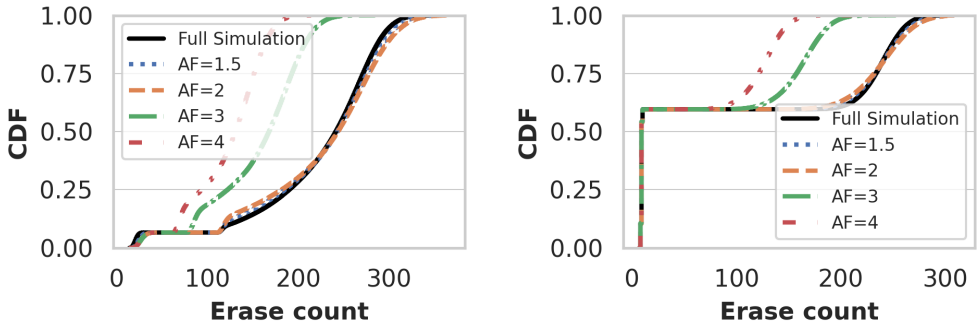
(a) TPCC (b) TPCE

Figure 11: SSD aging with 50 iterations of the workloads on FEMU. `FF-SSD` delivers a higher accuracy (93% for TPCC and 91% for TPCE) than DEVS (79% for TPCC and 60% for TPCE) and CML (18% for TPCC and 11% for TPCE).

3.4.2. Accuracy and Efficiency Tradeoff

In this section, we quantitatively analyze how different acceleration factors affect the overall accuracy and then provide a conservative AF to balance this tradeoff.

We apply four different acceleration factors (1.5, 2, 3, and 4) to `FF-SSD` and generate the wear states of running 100 iterations of LM-TBE and MSN-BEFS workloads on FTLSim. The performance comparison is shown in Figure 12. Overall, the accuracy is similar for $AF = 1.5$ and $AF = 2$ and drops distinctively when AF is greater than 2. Specially, for LM-TBE, the accuracy ranges from 56% ($AF = 4$) - 91% ($AF = 1.5$). The accuracy decreases by 4% from $AF = 1.5$ to $AF = 2$, while 19% from $AF = 2$ to $AF = 3$. Similarly, for RAD-BEFS, the accuracy is 98% for $AF = 1.5$ and 97% for $AF = 2$, and decreases by 13% for $AF = 3$ and 20% for $AF = 4$. Given the experiment results above, we conclude that `FF-SSD` generates an accurate distribution when at least half of the workloads are observed ($AF \leq 2$) and may occur more errors beyond that point.



(a) LM-TBE (b) RAD-BEFS

Figure 12: The tradeoff between aging accuracy and efficiency. `FF-SSD` generates an accurate estimation when at least half of the workloads are observed ($AF \leq 2$) and occur more errors beyond that point.

3.5. Conclusion and Limitations

We present *Fast-Forwardable SSD*, to the best of our knowledge, the first ML-based SSD aging framework that generates representative future wear-out states. We examine the effectiveness and usefulness of `FF-SSD` across state-of-the-art SSD development platforms. Our evaluations show that `FF-SSD` generates the desired age states of SSDs with high accuracy under real-world workloads. This work suggests many promising directions, and our immediate plan includes improving the accuracy through adaptive acceleration and predicting the wear states on real SSDs.

3.5.1. Improving Accuracy through Adaptive Acceleration.

We plan to implement *outlier detection* [126] on WAF to further improve accuracy and achieve adaptive acceleration control. WAF is defined by the increased writes caused by the background processes, measured as the ratio of total internal to external writes. A stable WAF over a long observation is an indicator that SSD is in a steady-state (i.e., the activeness of SSD background operations keeps at the same level). In this case, `FF-SSD` submits the estimation with high confidence and a more aggressive AF should be applied

to maximize aging efficiency. On the other hand, if the current WAF is an outlier from the past observations, indicative of the changes in host-side workloads or configurations, `FF-SSD` should adopt a moderate AF or even revoke the estimation to avoid high inference error.

Moreover, introducing outlier detection would also make `FF-SSD` adapt to dynamic workloads. While the past might be a good indicator of the future in some workloads, it may be not the case in many realistic settings. The changes in external workloads will be reflected by SSD internal activities, and finally detected by performing outlier detection algorithms. Once an outliner is determined, `FF-SSD` should skip the next prediction or roll back to a previous state.

3.5.2. Predicting on Real Devices.

The current design have only been validated on simulated environments. Our work would have greater applicability if we extend our validation to real SSDs where their internal details and states such as FTL logic and erase counts are not accessible. For these cases, we plan to use the information available through the SSDs' SMART (Self-Monitoring, Analysis, and Reporting Technology) interface, such as percentage used, available spare, and error detection count to predict SSDs' future health. Users may not be able to get the full distribution as in the simulation environment, but more general information can be inferred. For example, the changes in *percentage lifetime used* can be used to estimate how much host data the SSD can sustain under the current workload before its failure.

CHAPTER 4

Wear Leveling in SSDs Considered Harmful

4.1. Introduction

NAND flash-based solid-state drives (SSD) are an integral part of today's computing systems, used in mobile and embedded devices to large-scale data centers. However, flash memories wear out as they are programmed and erased, and progressively exhibit more errors [14, 115]. Furthermore, once used beyond an endurance limit, memory cells may not behave correctly and are considered bad [130]. Thus, minimizing the total amount of writes that causes programs and erases in the SSD is paramount in managing the overall lifetime [60, 93, 105]. However, this is becoming increasingly challenging as the endurance limit of flash memory has been steadily decreasing, as shown in Figure 47. This figure plots the endurance limit for a wide variety of SSDs over the past several years, estimated by dividing the amount of writes the SSD can sustain (in the form of TBW, terabytes written) by the logical capacity of the device. This downward trend is a result of trading reliability for higher density [14, 45, 79], and a modern flash memory cell can only withstand up to a few hundred programs and erases.

Wear leveling (WL) techniques seek to equalize the amount of wear within the SSD so that most, if not all, cells reach the endurance limit prior to the end of the SSD's lifetime [20, 25, 27, 42]. While there are different approaches to implementing WL (from static [20, 25, 42] to dynamic [27]), the underlying goal is to use younger blocks with fewer erases more than the older blocks. Static wear leveling techniques [20, 25, 42], in particular, proactively relocate data within an SSD, thereby incurring additional write amplification for the sake of equalizing the number of erases. In other words, WL techniques incur

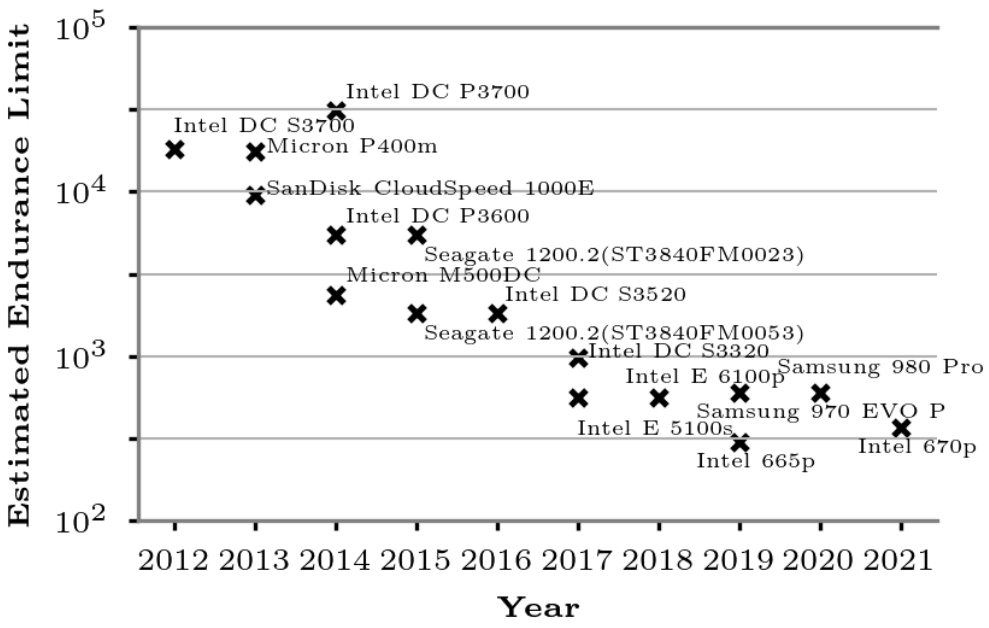


Figure 13: The estimated endurance limit of various SSDs in the past years. We estimate the endurance limit by dividing the SSD's TBW (terabytes written: the total amount of writes the SSD manufacturer guarantees) by the logical capacity. The y-axis is shown in log-scale.

additional wear-out to maximize the total amount of writes that the SSD can sustain.

We argue that the conventional approach of wear leveling is ill-suited for today's SSDs for the following three reasons. First, as Figure 47 shows, the endurance of flash memory cells has been decreasing to a level where only a few hundred program-erase cycles would wear them out. Because each WL decision also contributes to wear-out due to data relocation, suboptimal decisions for WL cause more harm than good. Secondly, we find that WL algorithms produce a counter-productive result where the erase counts diverge, increasing the spread rather than reducing it. We observe this when the workload's access footprint is large and its access pattern is skewed, as the WL attempts to proactively move data that it perceives to be cold into old blocks. Lastly, WL algorithms are a double-edged sword in which achieving a tight distribution of erase count comes the cost of high write amplification. Although adaptive WL approaches [25, 120] exhibit a low write amplification while

they are inactive during the early stages of the SSD's life, they overcompensate toward the end of life and significantly accelerate the wear-out.

Instead of designing a new wear leveling algorithm that patches these performance issues, we quantify the benefits of our work on capacity variance in SSDs. Unlike a traditional SSD that exports a fixed capacity, a capacity-variant SSD reduces its capacity gracefully as flash memories become bad. The fixed capacity interface, the current paradigm for all storage devices, is built around traditional hard disk drives that exhibit a *fail-stop* behavior[2] when their mechanical parts crash [136]. However, SSDs fail *partially* as flash memory blocks are the basic unit of failure, and mapping out failed bad blocks is an inherent responsibility of the SSD firmware as manufacture time bad blocks exist [130]. Wear leveling exists to maintain the fixed capacity interface and emulate a fail-stop behavior when the underlying technology is actually fail-partial. Thus, it is necessary to revisit the efficacy of wear leveling and the validity of a fixed capacity interface in a modern setting with a low endurance limit.

In this work, we evaluate three representative wear leveling techniques [20, 25, 42] in the traditional fixed capacity interface and show that their write amplification factors can reach up to 5.4. We also uncover that WL algorithms can produce opposite results where the variance of the erase count increases as WL becomes more active. We then evaluate the presence and absence of WL in both fixed-capacity and capacity-variant SSDs. Our experimental results show that capacity variance allows up to 84% more writes to the SSD with wear leveling. Interestingly, capacity variance *without* WL allows up to 2.94× more writes, demonstrating that WL techniques can accelerate the overall wear state significantly.

[2]In this context, a fail-stop model means that the storage device as a whole is either working or not, and its failure state can immediately be made aware by other components of the system.

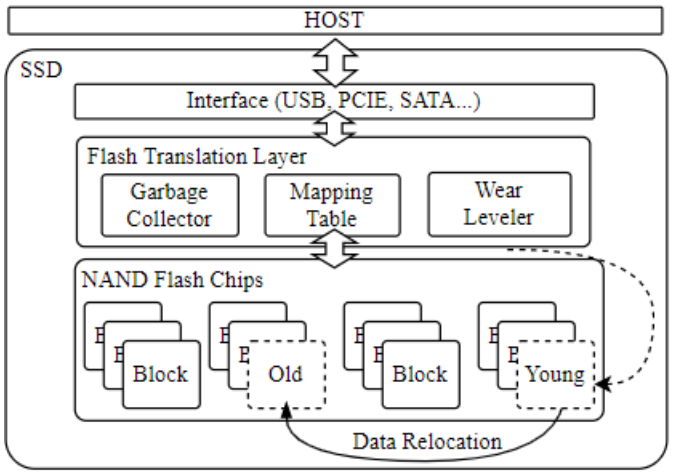


Figure 14: SSD controllers employ wear leveling algorithms to extend the device lifetime, which equalize the number of erases by migrating hot data (from older blocks) to younger blocks

The contributions of this work are as follows:

- We evaluate representative wear leveling algorithms and show that they exhibit high write amplification and undesirable results in modern SSD configurations. (§ 4.3)
- We qualitatively describe the benefits of capacity variance for SSDs and discuss the necessary modifications in the system and storage management. (§ 4.4)
- We quantitatively demonstrate that a capacity-variant interface can significantly extend the SSD's lifetime by evaluating the presence and absence of wear leveling algorithms in both fixed and variable capacity SSDs using a set of real I/O workloads. (§ 4.5)

4.2. Motivation

In this section, we provide the challenges in managing lifetime in SSDs, describe some of the typical wear leveling algorithms, and then qualitatively describe their shortcomings and side effects.

4.2.1. Managing the SSD lifetime

The overall architecture of a typical SSD is shown in Figure 14. The flash translation layer (FTL) hides the peculiarities of the underlying flash memory and provides an illusion of a block storage device [39]. Some of these peculiarities are as follows: (1) Flash memory prohibits in-place updates. Because all updates must be out-of-place, data are written to a new location and the *mapping table* associates the logical address with the physical location. (2) The program operation ($\texttt{1} \rightarrow \texttt{0}$) and the erase operation ($\texttt{0} \rightarrow \texttt{1}$) have different granularity: a page and a block respectively. Thus, to erase a block and reclaim space for new updates, all valid pages within the block must first be copied to another location through *garbage collection*. (3) Flash memory cells wear out and eventually become unusable. The *wear leveler* aims to equalize the wear for all blocks so that blocks do not prematurely wear out before the SSD's lifetime.

An SSD's lifetime is typically defined with a conditional warranty restriction under *DWPD (drive writes per day)*, *TBW (terabytes written)*, or *GB/day (gigabytes written per day)* [159]. This is because writing to an SSD induces programs (and eventually erases) that weaken the non-volatile memory, leading to a wear-out. Flash memory manufacturers specify how many times a flash memory block can be erased before turning bad [130]. Symptoms of these bad blocks include flash memory operation errors and high bit error rates, and an SSD typically maps out these blocks so that they will no longer be used [115, 130]. When too many blocks become bad, the SSD cannot maintain the logical capacity, and the entire storage device becomes unusable. As such, managing the number of writes to the SSD is critical in guaranteeing the lifetime.

However, it is not only I/O writes that cause programs and erases. Any SSD-internal housekeeping such as garbage collection and wear leveling perform erases that wear out the device [79]. This additional amount of writes in relation to the amount external I/O write

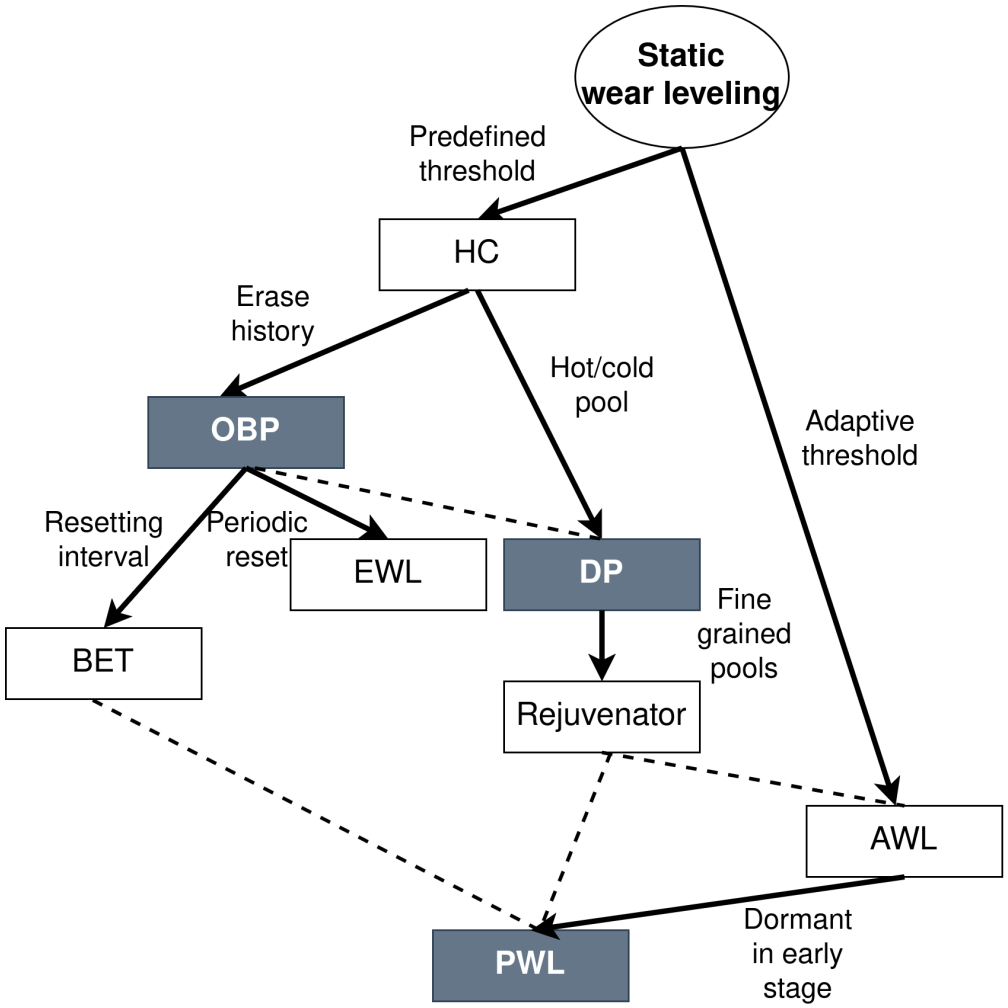
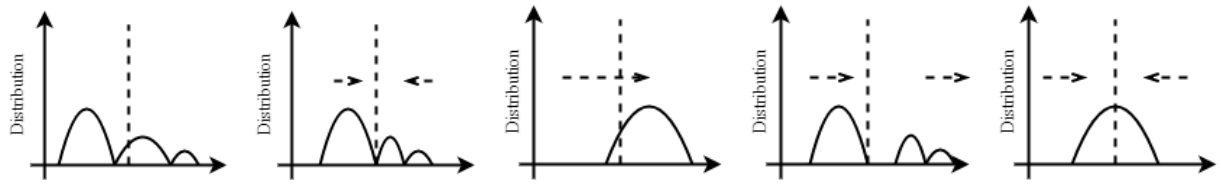


Figure 15: The relationship between various static wear leveling algorithms [20, 22, 23, 25, 42, 82, 98, 120]. Solid arrows indicate an enhancement from the previous algorithm (top-down implies the chronology), and the dotted lines indicate that they were evaluated in their respective paper.

is referred to as *write amplification (WA)*, and minimizing WA benefits not only the lifetime, but also the performance [79]. However, achieving this is not as straightforward. For garbage collection, greedily selecting the minimum amount of data to copy leads to unexpected inefficiencies [140], and wear leveling directly opposes WA as it causes additional data relocation to equalize the erase count.



(a) Without WL (b) Moderate WL (c) Aggressive WL (d) Incorrect WL (e) Ideal WL

Figure 16: Pictorial description of wear leveling behavior. Given a distribution of erase count without wear leveling (WL) in Figure 16a, Figure 16b shows the spread given a typical, moderately active WL. However, an overly aggressive WL accelerates the erase count as shown in Figure 16c, and an incorrectly behaving WL even causes an increased spread, shown in Figure 16d. Ideally, WL should achieve a distribution similar to Figure 16e.

4.2.2. Wear Leveling Algorithms

Figure 15 describes the relationship between different wear leveling algorithms [20, 22, 23, 25, 42, 82, 98, 120].

One naïve implementation called *hot-cold swap* simply swaps the data stored in the oldest block and the youngest block once the difference in erase count exceeds some predefined threshold [82]. For example, as shown in Figure 14, the data in the young block is relocated to the old block. This approach is based on the intuition that the youngest block likely holds infrequently updated *cold* data while the oldest block likely has frequently updated *hot* data. After the hot-cold swap, the youngest block will likely be erased because of the hot data, while the oldest block with cold data will not be erased. However, the oldest block may be selected for WL shortly after (even though it holds cold data) if the algorithm incorrectly judges it to hold hot data, causing unnecessary data swaps [20].

Old Block Protection (OBP) [42] addresses this problem by maintaining a recent history of WL activities. Once a block is erased, its erase count is compared with that of the youngest block in use: if the difference exceeds a predefined threshold, and if this just-erased block hasn't been involved in WL recently, the content of the youngest block will

be copied into the just-erased block. OBP prevents the old block now containing a likely-cold data (that was copied from the youngest block) from being involved in wear leveling again.

Dual-Pool (DP) [20] implements a more sophisticated algorithm to prevent the same block from partaking in WL repeatedly. It maintains two pools of blocks, hot pool and cold pool, each intended for storing hot data and cold data, respectively. It also performs a hot-cold swap, but between the youngest block (lowest erase count) in the cold pool and the oldest block (highest erase count) in the hot pool. Furthermore, the pool associations for the two blocks are also changed so that the oldest block in the hot pool becomes part of the cold pool (since it now holds cold data). This prevents the same oldest block from getting selected by the wear leveler. DP also implements an adaptive pool resize where a hot pool block that contains data that became cold is moved to the cold pool and vice versa.

Unlike OBP and DP, **Progressive Wear Leveling (PWL)** [25] adaptively changes the trigger condition for wear leveling based on the overall wear state of the SSD. In the early stage of life for the SSD (when the average erase count is low), PWL lies dormant, avoiding unnecessary data copies. However, as the overall erase count increases, the activeness of PWL proportionally increases as well. This algorithm requires a predefined initial threshold (THR_{init}) that dictates the behavior: lower THR_{init} causes WL to become active early in the SSD's life while a higher threshold delays the point in which WL becomes active.

OBP [42], DP [20], and PWL [25] represent different types of wear leveling algorithms for SSDs. They are often described as static WL, distinct from dynamic WL [26, 27] that is integrated with garbage collection and block allocation. We do not consider dynamic WL in our evaluation as it conflates the efficiency of space reclamation and effectiveness of

wear leveling.

4.2.3. Wear Leveling Behaviors

Wear leveling algorithms, in practice, often exhibit unintended odd behaviors that stem from misjudging the lifetime of data in a block. Figure 16 depicts the different possible erase count distributions, given that without WL in Figure 16a. A moderately active WL (Figure 16b) would reduce the spread of erase count compared to that without, but an overly aggressive WL (Figure 16c) increases the overall erase count through significant write amplification. Hence, WL can be a double-edged sword where it is either ineffective reducing the variance in erase count, or overly active and causes the younger blocks to become as old as the others. A highly undesirable scenario is where the erase count diverges as shown in Figure 16d, and we describe the conditions in which this occurs in § 4.3. Ideally, WL should reduce the spread as shown in Figure 16e.

In the next section, we demonstrate that the existing WL algorithms show behaviors in Figure 16, and argue that because of their shortcomings they may not be suitable for today's flash memory with a low endurance limit where each WL decision consumes an erase count.

4.3. Performance of Wear Leveling

We evaluate the performance of three representative wear leveling (WL) algorithms: Old Block Protection (OBP) [42], Dual-Pool (DP) [20], and Progressive Wear Leveling (PWL) [25]. $OBP(TH, PT)$ configuration copies the data of the youngest block to the just-erased block if the difference between the erase counts exceeds TH, unless the block has been involved in WL in the recent PT erases. $DP(TH)$ swaps the content of the oldest block in the hot pool and the youngest block in the cold pool if the erase count different exceeds TH. $PWL(TH)$ performs wear leveling on blocks that exceed a certain erase count thresh-

Table 2: SSD configuration and policies. Only the parameters relevant to understanding the wear leveling behavior are shown.

Parameter	Value	Parameter	Value
Page size	4 KiB	Physical capacity	284 GiB
Pages per block	256	Logical capacity	256 GiB
Block size	1 MiB	Over-provisioning	11%
Block allocation	FIFO	Garbage collection	Greedy

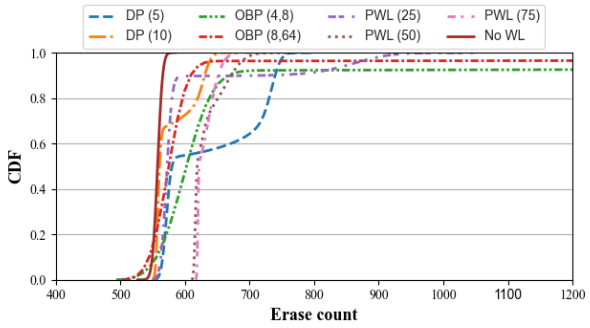
old that scales proportionally with the average erase count, offset by the $TH\%$ of the endurance limit. We first describe our experimental setup, present the evaluation results, and summarize our findings.

4.3.1. Experimental Setup

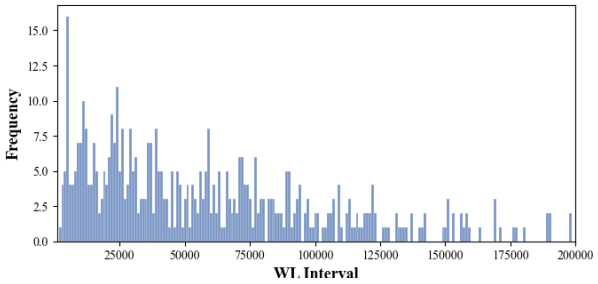
We extend FTLSim [34][3] to include a WL model for our experiments. The original FTLSim validates the analytical model for garbage collection, and thus focuses on accurately modeling SSD-internal statistics such as write amplification rather than SSD-external performance such as latency and throughput. This simplicity makes FTLSim fast, allowing us to simulate the entire lifetime of an SSD (a few hundreds of TiB written). Table 2 summarizes the SSD configuration and policies for our experiments.

To understand the behavior of WL techniques, we synthetically generate workload to control the I/O pattern better. All I/Os are small random writes, but the distribution is controlled by two parameters r and h ($0 < r < 1$ and $0 < h < 1$): r fraction of writes go to the h fraction of the footprint (hot addresses) [140]. The r fraction of accesses are uniformly and randomly distributed within the hot addresses, and conversely, the $1 - r$ fraction of accesses are uniformly and randomly distributed within the $1 - h$ cold addresses. We use r/h to indicate that the r fraction of writes occurs on the h fraction of the logical space. Unless otherwise noted, we generate the I/O workload for the entire logical address

[3]We plan to make our FTLSim extension publicly available.



(a) The distribution of erase count after writing 25TiB.



(b) The interval between consecutive OBP WLs for the oldest block.

Figure 17: The performance anomaly of wear leveling under $r/h = 0.9/0.1$ workload. In Figure 17a, we observe that wear leveling makes the erase count more *uneven*, as evident by the flat plateaus in the CDF curve. This is because a small set of blocks are heavily involved in wear leveling, as shown in Figure 17b: most of the erases happen soon after the protection period ends.

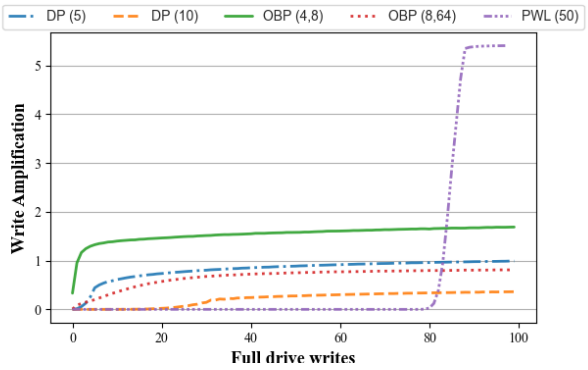
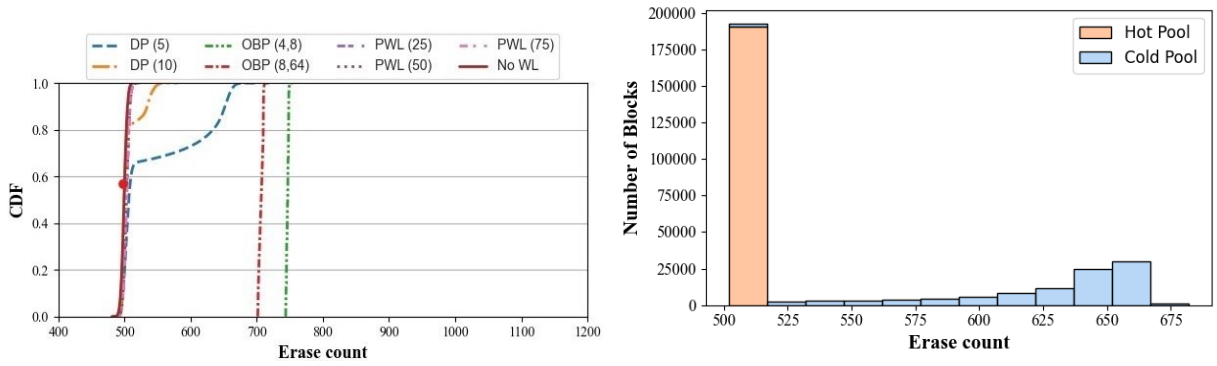


Figure 18: The write amplification caused by wear leveling under a $r/h = 0.9/0.1$ synthetic workload. $PWL(50)$ that aggressively performs wear leveling at the late stage causes its write amplification to be as high as 5.4×.

space. Prior to each experiment, we pre-condition the SSD with one sequential full-drive write, followed by three random full-drive writes (256 GiB sequential + 768 GiB random write) to put drive into a steady state [151].

4.3.2. Evaluation of Wear Leveling under Synthetic Workloads

We investigate the performance of wear leveling in the following three aspects: (1) write amplification, (2) effectiveness in equalizing the erase count, and (3) behavior under differ-



(a) A skewed workload with $r/h = 0.8/0.2$. (b) The pool association histogram for $DP(5)$.

Figure 19: The distribution of erase count under $r/h = 0.8/0.2$ (skewed). The red dot indicates the average erase count for NoWL. For the skewed workload in Figure 19a, wear leveling comes at a high cost of write amplification, as evident by the fact that the lines are on the right side of the red dot. DP causes the erase count to diverge, and examining the pool association reveals that the blocks associated with cold data are much older than the blocks associated with hot data, as shown in Figure 19b. This inversion occurs because the algorithm assumes that blocks with hot data are old and those with cold data are young, and swaps the oldest block in the hot pool with the youngest block in the cold pool. If the youngest block in the cold pool becomes older than the oldest block in the hot pool, an unnecessary WL takes place, moving the hot data into the older block.

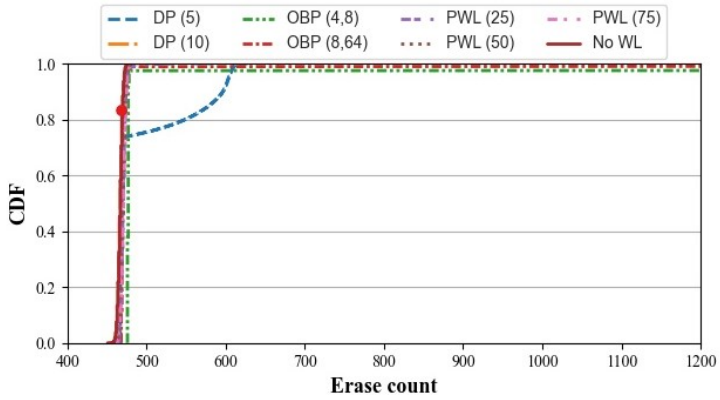


Figure 20: The distribution of erase count under $r/h = 0.5/0.5$ (uniform). we observe that the benefit from wear leveling is negligible compared to not running at all.

ent access footprint.

Write Amplification

We measure the WL-induced write amplification (WA) by using a synthetic workload of $r/h = 0.9/0.1$ for up to 100 full-drive writes (25 TiB). We examine PWL and two different configurations for OBP and DP wear leveling each. The WL parameter values we experiment with are similar to those used in the prior work [20] [25].

Figure 18 shows the write amplification caused by wear leveling over time, and we make the following four observations. First, the wear leveling-induced write amplification can be as high as 5.4×. This means that for each 256 GiB user data written, wear leveling alone will create an additional 1.35 TiB of data writes internally. Secondly, the WL threshold parameter TH dictates the WA for both OBP and DP. Changing the TH from 8 to 4 for the OBP algorithm will amplify the amount of data written by 1.2×, while changing from 10 to 5 for the DP, by 1.4×. Thirdly, as expected, PWL is inactive during the early stages. However, beyond about 80 full-drive writes, it exhibits an aggressive behavior, causing significant WA, as high as 5.4×. Lastly, WA steadily increases over time as the SSD ages, indicating that SSD aging will accelerate as more data are written.

Wear Leveling Effectiveness

We measure the distribution of erase count of different configurations under a synthetic workload as shown in Figure 17. We exercise a workload with $r/h = 0.9/0.1$ for 100 full-drive writes (25 TiB).

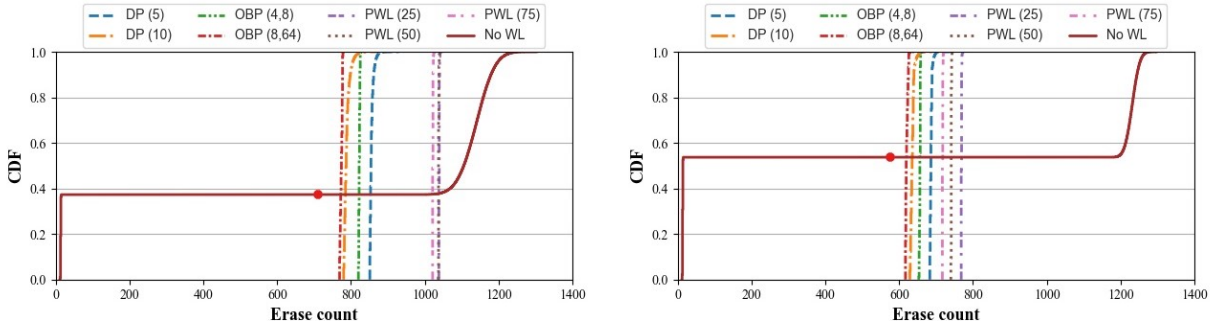
Figure 17a shows that under $r/h = 0.9/0.1$, all three WL show uneven distribution of erase counts, in fact, worse than not running WL ($NoWL$). For $OBP(4, 8)$ and $OBP(8, 64)$, we observe that the CDF plateaus before reaching 1.0, indicating that there are a small group (of 7% for $OBP(4, 8)$ and 3% for $OBP(8, 64)$) blocks that exceed the 1200 erase count. Similarly, DP and PWL also show a concave dip in the CDF curve, indicating a

bimodal distribution of erase counts. $NoWL$, on the other hand, shows a nearly vertical line, meaning that the erase counts are more tightly distributed. We consider this to be a *performance anomaly* of wear leveling because it behaves the opposite of what is expected.

Through analysis, we find that the block with the highest erase count in the OBP algorithm is overly involved in wear leveling, despite the fact that the PT parameter is designed to protect the block from being erased again. We run another experiment using the same $r/h = 0.9/0.1$ workload, but for $OBP(5, 1000)$. Figure 17b shows the interval between two consecutive WL for the most worn-out block. Even with a long $PT = 1000$, the block is involved in WL again soon after its protection period expires.

We also evaluate OBP, DP and PWL with $r/h = 0.8/0.2$ and $r/h = 0.5/0.5$, as shown in Figure 19 and Figure 20. Even though $OBP(4, 8)$ and $OBP(8, 64)$ show nearly vertical CDF curves, the average erase count is much higher, indicating that it amplifies the amount of written data significantly. $PWL(25)$, $PWL(50)$, and $PWL(75)$, however, show a similar result with $NoWL$. On the other hand, both $DP(5)$ and $DP(10)$ show performance anomaly with a skewed workload (Figure 19a), as shown in the wider CDF curves compared to $NoWL$. We examine the bimodal distribution of $DP(5)$ and find that the erase count distribution is bimodal with blocks associated with the cold pool older than those in the hot pool. The DP algorithm's underlying assumption is that blocks containing hot data are older than blocks with cold data, and it compares the erase count of the oldest block in the hot pool and the youngest block in the cold pool. If the youngest block in the cold pool happens to be older than the oldest block in the hot pool, however, it will still trigger the swap between the two blocks, causing this inversion.

On the other hand, with a uniformly random workload (Figure 20), there is a negligible difference between running WL and not running WL at all. This is because with a uniform workload, all blocks are used equally, and there is little room for wear leveling. We



(a) A skewed distributed workload with $r/h = 0.9/0.1$.

(b) A uniformly random workload with $r/h = 0.5/0.5$.

Figure 21: The distribution of erase count when only 5% of the logical address space is used. The red dot indicates the average erase count for NoWL. With a small footprint, wear leveling achieves good evenness in erase count while not running it causes a bimodal distribution. However, with a skewed workload in Figure 21a, we observe that the erase counts for WL are further to the right of the red dot compared to those in Figure 21b, indicating that WL amplified the amount of data writes.

do observe, however, that $DP(5)$ still exhibits a performance anomaly though at a smaller degree than under $r/h = 0.9/0.1$ (*cf.* Figure 17a).

These experiments show that WL algorithms are a double-edged sword. As shown in Figure 17a, it can make the distribution of wear worse than not running WL at all. On the other hand, it can achieve good wear leveling but at a high cost of accelerated overall wear state.

Small Access Footprint

Previously in § 4.3.2 and § 4.3.2, the experiments were conducted by writing to the entire logical address space. Here we explore the performance of wear leveling when the accesses are restricted to a small address space using two synthetic workloads, $r/h = 0.9/0.1$ and $r/h = 0.5/0.5$, as shown in Figure 21.

Overall, we observe that most WL techniques are effective in equalizing the erase count, as shown by the near-vertical CDF curve in both Figure 21a and Figure 21b. $NoWL$, on

the other hand, shows a bimodal distribution between used blocks and unused blocks in both workloads. We also observe that when the workload is skewed (Figure 21a), the WL techniques achieve this evenness by amplifying the amount of data writes, as shown by the rightward shift in the CDF curves. For a uniform workload, on the other hand (Figure 21b), the overall write amplification from wear leveling is much lower as data in used blocks are equally likely to be invalidated.

Unlike the results from Figure 17 and Figure 19 where the entire logical address space is written, WL is effective only when a small fraction of the address space is used, restricting its overall usefulness.

4.3.3. Summary of Findings

Based on the experimental results using synthetic workloads, we qualitatively summarize the effectiveness of wear leveling in Table 3.

- **Uniform access with a small access footprint:** Overall, WL evens out the erase count with low write amplification. (Figure 21b)
- **Skewed access with a small access footprint:** WL achieves good performance but at the cost of high write amplification. (Figure 21a)
- **Uniform access with a large access footprint:** The performance of wear leveling has a negligible difference with not running it at all. There may be cases where performance anomaly occurs. (Figure 20)
- **Skewed access with a large access footprint:** Wear leveling not only amplifies writes, but also exhibits an anomaly by significantly accelerating the erase count for a group of blocks. (Figure 17a)

Instead of proposing a new wear leveling algorithm that solves both the write amplifica-

Table 3: Qualitative effectiveness of wear leveling.

	Uniform access	Skewed access
Small footprint	Effective	Write amplified
Large footprint	Negligible	Anomaly

tion overhead and performance anomaly, we question the circumstances that require wear leveling and examine its necessity in the next section.

4.4. A Capacity-Variant SSD

Our experiments on the effectiveness of wear leveling show that it is difficult to achieve a good WL performance. There is only a limited number of scenarios where WL achieves its intended goal; in other cases, it either results in high write amplification or further unevenness.

We argue that WL is an artifact only designed to maintain an illusion of a fixed capacity device wherein its underlying storage components (i.e., flash memory blocks) either all work or fail. This, however, is far from the truth where blocks wear out individually (often at different rates) and are mapped out by the SSD once they become unusable. In other words, it is natural for the SSD's physical capacity to reduce over time, and it is only the fixed capacity abstraction that mandates the SSD's lifetime to be defined as when the physical capacity becomes less than the logical capacity.

Instead, a capacity-variant SSD [80] would gracefully reduce its exported capacity. This has three benefits over the traditional fixed capacity interface. First, wear leveling becomes unnecessary as it does not need to ensure that all blocks wear out evenly. This would free the SSD from needing the computational resources for the WL algorithm and incur no write amplification overhead. Secondly, the lifetime of an SSD would extend significantly, and it would be defined by the amount of data stored in the SSD, not by the initial logical

Table 4: Trace workload characteristics. `YCSB-A` is from running YCSB [172], `VDI` is from a virtual desktop infrastructure [91], and the remaining 9 (from `WBS` to `RAD-BE`) are from Microsoft production servers [78]. `Footprint` is the size of the logical address space that has write accesses, and `Hotness` is the skewness where r% of data are written to the top h% of the frequently accessed address. `Sequentiality` is the fraction of write I/Os that are sequential.

Workload	Description	Footprint (GiB)	Avg. write size (KiB)	Hotness (r/h)	Sequentiality
YCSB-A	User session recording	89.99	50.48	64.69/35.31	0.49
VDI	Virtual desktop infrastructure	255.99	17.99	64.45/35.55	0.14
WBS	Windows build server	56.05	27.82	60.34/39.66	0.02
DTRS	Developer tools release	150.63	31.85	54.20/45.80	0.12
DAP-DS	Advertisement caching tier	0.17	7.21	78.62/21.38	0.03
DAP-PS	Advertisement payload	36.06	97.20	55.02/44.98	0.16
LM-TBE	Map service backend	239.49	61.90	60.29/39.71	0.94
MSN-CFS	Storage metadata	5.58	12.92	69.28/30.72	0.25
MSN-BEFS	Storage backend file	31.42	11.62	70.18/29.82	0.03
RAD-AS	Remote access authentication	4.53	9.87	70.90/29.10	0.45
RAD-BE	Remote access backend	14.73	13.02	65.51/34.49	0.33

capacity. Lastly, it would be easier to determine when to replace an SSD. An SSD whose capacity is reduced, close to the amount of data stored and in use, would be a clear indication of an ailing SSD.

4.5. Evaluation

We evaluate both the presence and absence of wear leveling (OBP [42], DP [20], and PWL [25]) on both a fixed capacity SSD and a capacity-variant SSD using real-world I/O traces. We design the experiments with the following questions in mind:

- Does capacity variance extend the lifetime of the SSD? How does wear leveling (WL) interact with capacity variance? (§ 4.5.1)
- How sensitive is the effectiveness of capacity variance to different garbage collection (GC) policies? Do GC policies alter WL results? (§ 4.5.2)
- What is the file system-level overhead with capacity-variant SSDs? Is there a tunable

We use the same extended FTLSim [34] from § 4.3 and the SSD configuration in Table 2 for our evaluation. However, we set the endurance limit to 500 erases, a typical level for QLC [97, 102], and once a block reaches this, it will be mapped out and no longer used in the SSD. This mapping out will effectively reduce the SSD's internal physical capacity. For the fixed capacity SSD, the SSD is considered to reach its end of life once the physical capacity becomes smaller than its logical capacity. On the other hand, the capacity-variant SSD can reduce its capacity below the initial logical space, to as low as the access footprint for the workload. We assume that the file system will `TRIM` unnecessary data as the exported capacity shrinks, and that the overhead for maintaining a contiguous block address space is negligible.

For the workload, we use eleven real-world I/O traces that were collected from running YCSB [172], a virtual desktop infrastructure (VDI) [91], and Microsoft production servers [78]. The traces are modified into a 256GiB range (the logical capacity of the SSD), and all the requests are aligned to 4KiB boundaries. Similar to the synthetic workload evaluation, the SSD is pre-conditioned with one sequential full-drive write and three random full-drive writes on the entire logical space. The traces run in a loop indefinitely, continuously generating I/O until the SSD becomes unusable at its end of life. Table 4 summarizes the trace workload characteristics, only showing the information relevant to understanding the behavior of wear leveling (WL).

We evaluate the following eight designs:

- **Fix_NoWL** does not run any WL on a fixed capacity SSD.
- **Fix_OBP** runs $OBP(4, 8)$ on a fixed capacity SSD.

- **Fix_DP** runs $DP(5)$ on a fixed capacity SSD.
- **Fix_PWL** runs $PWL(50)$ on a fixed capacity SSD.
- **Var_NoWL** does not run any WL on a capacity-variant SSD.
- **Var_DP** runs $DP(5)$ on a capacity-variant SSD.
- **Var_PWL** runs $PWL(50)$ on a capacity-variant SSD.

4.5.1. Effectiveness of Capacity Variance

Figure 22 shows the amount of data written to the SSD before failure for the 11 I/O traces. The y-axis is in terms of the number of drive writes. For example, for 100 drive writes, 25TiB of data have been written. Overall, we observe that with fixed capacity SSDs, running WL is better than not running WL, but only by a small margin: Fix_DP extends the lifetime by only 18% on average compared to Fix_NoWL, and with workloads such as `VDI` and `DTRS`, Fix_OBP and Fix_DP perform worse than Fix_NoWL. However, with capacity variance, not running WL is better than running WL by a large margin: Var_NoWL extends the lifetime by 1.33× on average, and as much as 2.94× for `RAD-BE` compared to Fix_PWL. We explain this result by the measurement of write amplification caused by wear leveling, shown in Figure 23.

Workloads with a Relatively Small Footprint

We observe that capacity variance is most effective on workloads such as `DAP-DS`, `DAP-PS`, `MSN-CFS`, `RAD-AS`, and `RAD-BE`. These workloads are characterized by a small access footprint where gracefully reducing the capacity allows for the SSD to be continuously used. The lifetime extension gained by capacity variance with wear leveling is minimal. At most, 9.2% is gained from Fix_PWL to Var_PWL, and only 7.7% from Fix_DP to Var_DP, 4.9% from Fix_OBP to Var_OBP. The underlying reason for this is WL causes write

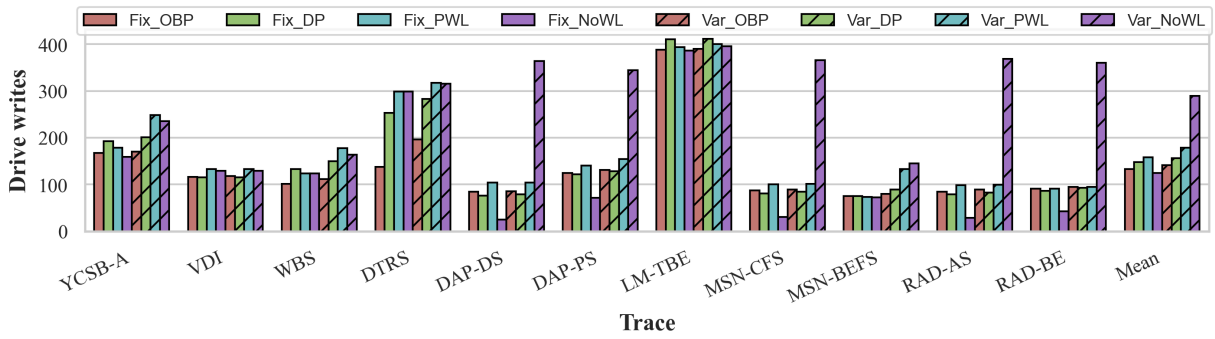


Figure 22: Evaluation of the presence and absence of wear leveling in both a fixed capacity and a capacity-variant SSD. Capacity variance extends the lifetime by 1.33× on average, and as high as 2.94× in the case of `RAD-BE`.

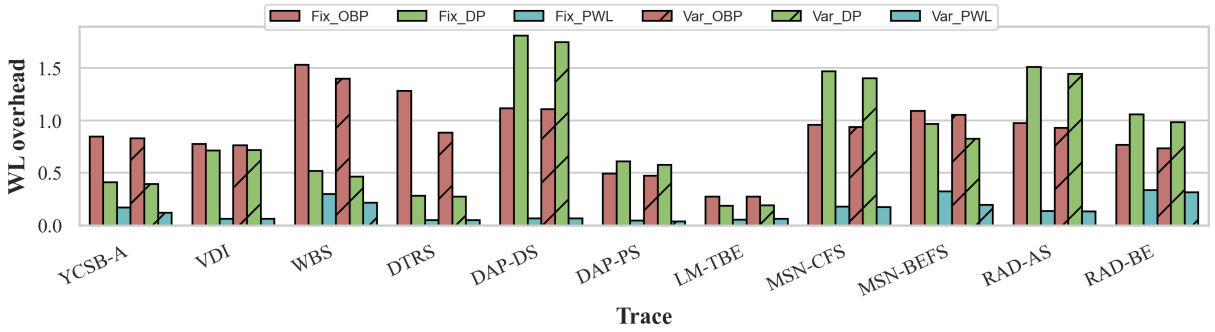


Figure 23: Write amplification caused by WL. While a large sequential workload such as `LM-TBE` only has a low write amplification overhead of 0.18, most other workloads exhibit high wear leveling overhead for OBP an DP, reaching as high as 1.8. PWL is aggressive at the late stage but dormant initially, causing the overall overhead is relatively low.

amplification (especially on skewed workload), aging all blocks in the SSD. In particular, under workload such as `DAP-DS`, the write amplification for Fix_DP is as high as 1.80. A capacity-variant SSD running WL thus has to reduce its capacity abruptly as all blocks are similarly worn out. On the other hand, a capacity variance without wear leveling allows 2.94× more data to be written to the SSD for `RAD-BE`, and 13.9× for `DAP-DS` with most gain compared to Fix_NoWL.

`MSN-BEFS` also has a small footprint, but we observe a comparatively lower lifetime extension of 0.91×. In fact, the lifetime of Fix_NoWL isn't too far off from that of Fix_DP, only 5% less. The reason for this is due to garbage collection: This workload contains a lot

of small random writes, causing garbage collection to be active, dwarfing the write amplification from WL. Because of this, MSN-BEFS only allows 145 full-drive writes (36.27 TiB) even for the capacity-variant SSD.

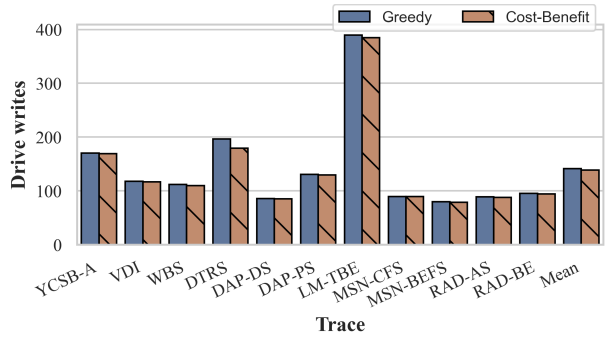
Workloads with a Relatively Large Footprint

LM-TBE and VDI are two workloads with the largest footprint, and the benefit of capacity variance is diminished in such workloads. For VDI, Var_NoWL reduces the lifetime by 3.1% compared to Fix_PWL, and for LM-TBE, Var_NoWL reduces it by 3.6% compared to Fix_DP. A large footprint means that there is little to gain from reducing the capacity as data are still in use. For LM-TBE, the large sequential write with relatively high uniformity causes the write amplification for WL to be small, as low as 0.18. This allows wear leveling to squeeze more lifetime out of the SSD. This is the only workload where Var_DP has the longest lifetime.

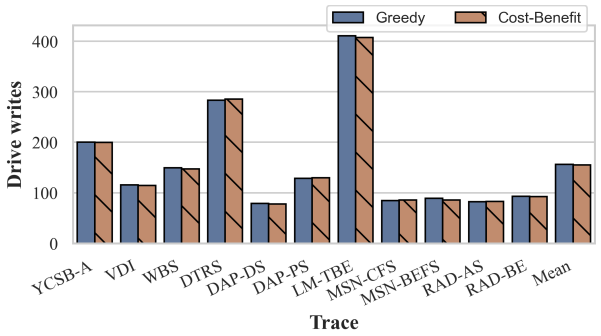
DTRS is one of the rare occasions where not running WL is better in a fixed capacity SSD. Fix_NoWL allows 117% more writes compared to Fix_OBP, and 18% more compared to Fix_DP. This is due to the high write amplification of wear leveling. Although the write access pattern of DTRS is fairly uniform, we suspect that a wear leveling anomaly occurred, causing a subset of blocks to age rapidly. Introducing capacity variance extends the lifetime for all three cases, however, with Var_NoWL extending the lifetime by 24% compared to Fix_DP. Var_PWL outperforms Var_NoWL, but the difference is only 3%.

4.5.2. Sensitivity to Garbage Collection

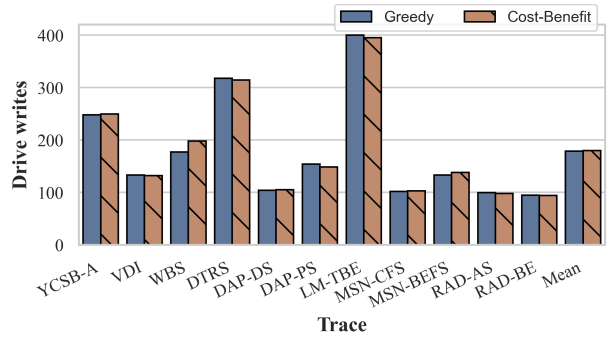
In this experiment, we investigate the effect that garbage collection (GC) policies have on the lifetime of the SSD. GC has a profound impact not only on the overall performance of the drive, but also on the lifetime because of its write amplification [32, 54, 77, 81, 173]. We compare two GC policies: greedy [176] and cost-benefit [140]. Greedy policy selects



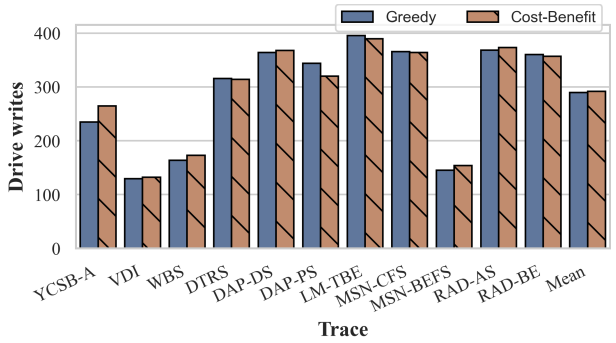
(a) `Greedy` vs. `cost-benefit` with Var_OBP.



(b) `Greedy` vs. `cost-benefit` with Var_DP.



(c) `Greedy` vs. `cost-benefit` with Var_PWL.



(d) `Greedy` vs. `cost-benefit` with Var_NoWL.

Figure 24: The sensitivity testing of capacity variance to garbage collection (GC) policies. GC has negligible effect on the overall lifetime of the capacity-variant SSD, with at most 6.6% difference between `greedy` and `cost-benefit` in the case of `YCSB-A` without wear leveling. The average difference between the two GC policies across all workload are 2.03%, 0.43%, 0.6%, and 0.67% for Var_OBP, Var_DP, Var_PWL, and Var_NoWL, respectively.

the block with the least number of valid pages as the victim to clean with the rationale that it would incur the least overhead. However, LFS [140] shows that greedy can lead to a suboptimal performance especially on skewed workloads because hot, soon-to-be-updated data gets cleaned when it would have been invalidated on its own.

Figure 24 shows the lifetime of a capacity-variant SSD when changing the GC policy with each subfigure representing a different WL policy. Overall, we observe a negligible difference in the SSD's lifetime when changing GC: on average, the difference is 2.03% for OBP

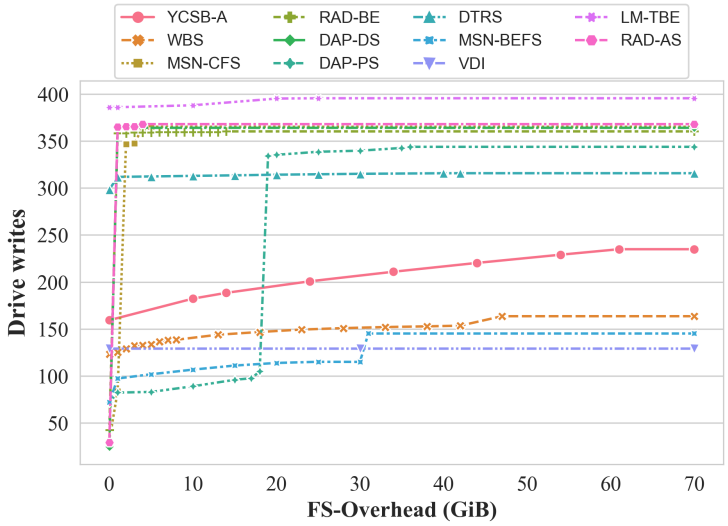


Figure 25: Lifetime of a capacity-variant SSD as a function of maximum allowed file system-level data relocation when no wear leveling is active. With the overhead of relocation set to zero, the capacity-variant SSD behaves identical to that of a fixed capacity SSD. We observe that with most workloads gain significant lifetime even with a small allowance of data relocation.

in Figure 24a, 0.43% for DP in Figure 24b, 0.6% for PWL (Figure 24c), and 0.67% without wear leveling (Figure 24d). In fact, the largest difference is for `YCSB-A` under Var_NoWL, and this is only a 6.6% difference. The experiment results here are also consistent with the data presented in Figure 22, where Var_NoWL extends the lifetime more than other designs.

We attribute the lack of sensitivity to GC to the following two reasons. First, we experiment on the entire lifetime of the SSD and the WA of WL steadily increases with age. This reduces the overall effect of the GC's WA on the lifetime. Secondly, capacity variance mitigates the WA from GC. In a fixed capacity SSD, the overhead of GC increases as the effective over-provisioning decreases when physical blocks map out. However, a capacity-variant SSD also reduces the logical space, and thereby does not exacerbate the WA from GC.

4.5.3. Limiting File System Overhead

Figure 25 shows the relationship between different file system (FS)-overhead bounds and the maximum amount of data that can be written to the system. The x-axis is the FS-level overhead bound we set for the capacity variance and the y-axis is the amount of external data written to the SSD under that bound in terms of the number of drive writes. For example, the bound equal to 0 means that the system doesn't allow any FS-level overhead caused by capacity variance. On the other hand, setting the bounds to be unlimited means the exported logical space will be reduced as the number of bad blocks increase until the logical space is the same as the workload footprint, meaning that the file system partition is full. No wear leveling is active in this experiment.

We observe that for the workload with a small footprint (`DAP-DS`, `DAP-PS`, `MSN-CFS`, `RAD-AS`, and `RAD-BE`), a small amount of capacity reduction allows a significant gain for the SSD lifetime. For example, with a 1GiB allowed overhead for data relocation at the file system, Var_NoWL under `RAD-AS` gains 11.64× lifetime compared to Fix_NoWL. On the other hand, for workloads with a large footprint (`VDI`, `DTRS`, `LM-TBE`) we do not see a noticeable increase in the lifetime as we increase the amount of allowed data relocation. However, these workloads typically achieve a long lifetime even without capacity variance. Interestingly, `DAP-PS` shows a sudden increase in its lifetime when allowing the 18th GiB to relocate. This is due to the large capacity of allocated space in the high address range for `DAP-PS`.

Overall, Figure 25 demonstrates that capacity variance can be effective even when limiting the file system-level overhead. Var_NoWL with only a 1GiB allowance achieves a 3.21× increase in lifetime across all workloads compared to Fix_NoWL, 4.07× at 5GiB, and 4.10× at 10GiB.

4.6. Discussion and Related Works

Wear leveling is a mature and well-studied topic in both academia and industry [39]. However, to the best of our knowledge, our work is the first that presents a comprehensive evaluation of existing wear leveling algorithms in the context of today's low endurance limit, and quantitatively examines the benefits of capacity variance. This work builds upon a number of prior works, and we discuss how it relates to them below.

Wear leveling and write amplification. There exists a large body of work on garbage collection and its associated write amplification (WA) for SSDs, from analytical approaches [34, 57, 133, 176] to designs supported by experimental results [21, 77, 181]. However, there is surprisingly limited work that measures the WA caused by wear leveling (WL), and they often rely on a *back-of-the-envelope calculation* for estimating the overhead and lifetime [182]. Even those that perform a more rigorous study evaluate the efficacy of WL by measuring the amount of writes the SSD can endure [25, 69, 93, 166] or the distribution of erase count [2, 20, 58]; only the Dual-Pool algorithm [20] present the overhead of WL. Our faithful implementation of the Dual-Pool algorithm yields different WA from the original work, however, due to the differences in system configuration and workload.

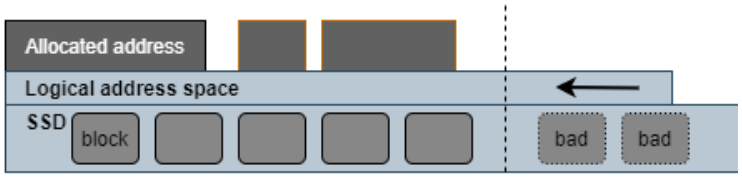
Lifetime extension without wear leveling. Similar to our approach, there are a number of prior works that extend the lifetime of an SSD without wear leveling [69, 93, 166]. READY [93] dynamically throttles writes to exploit the self-recovery effect of flash memory, and ZombieNAND [166] re-uses expired flash memory blocks after switching them into SLC mode. Wear unleveling [69] identifies strong and weak pages, and skips writing to weaker ones to prolong the lifetime of the block. We believe that capacity variance is compatible with these approaches as we focus on the interface exported by the SSD.

Zoned namespace. Zoned namespace (ZNS) [165] is a new abstraction for storage de-

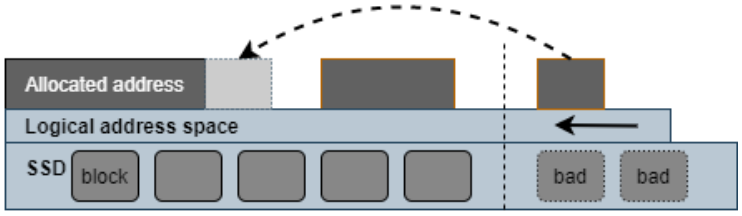
vices that has gained significant interest in the research community [10, 52, 152]. ZNS SSDs export large fixed-sized zones (typically in the GiB range) that can be written sequentially or reset (deleted) entirely. The approach shifts the responsibility of managing space via garbage collection from the SSD to the host-side file system. Unlike its predecessor (Open Channel SSD [11]), ZNS does not manage the physical flash memory block and leaves the lifetime management to the underlying SSD, allowing for the integration of capacity variance with ZNS.

File systems and workload shaping. It is widely understood that SSDs perform poorly under random writes [24, 56], and file systems optimized for SSDs take a log-structured approach so that writes would be sequential [42, 90, 116, 129, 180]. Because of the log-structured nature, these file systems create data of similar lifetime, leading to a reduction in wear leveling overhead [56]. Similarly, aligning data writes to flash memory's page size reduces the wear on the device [75].

Flash arrays and clusters. Most existing works at the intersection of RAID and SSDs address its poor performance due to the compounded susceptibility to small random writes [28, 62, 83, 87, 119]. Of these, log-structured RAID [28, 83] naturally achieves good inter- and intra-SSD wear leveling, but at the cost of another level of indirection. On the other hand, Diff-RAID [9] takes a counter-intuitive approach and intentionally skews the wear across the SSDs by distributing the parity blocks unevenly to avoid correlated failures in the SSD array. While we do not advocate the intentional wear out of flash memory blocks, willful neglect of wear leveling can be beneficial in the context of capacity variance.



(a) Capacity reduction without relocation overhead.



(b) Capacity reduction with relocation overhead.

Figure 26: Illustration of the data relocation overhead for reducing capacity. In Figure 26a, the capacity can be reduced without the file system relocating data at the high address. In Figure 26b, however, data in the allocated space at the high address must be moved before reducing capacity.

4.7. Limitations

4.7.1. File System Overhead

A capacity-variant interface would need support from both the file system and device driver. Thankfully, the current system design can make this transition seamless for the following three reasons. First, the `TRIM` command, widely supported by interface standards such as `NVMe`[4] [124] allows the file system to explicitly declare that the data (at the specified addresses) are no longer in use. This allows the SSD to discard the data safely and would help determine if the exported capacity can be gracefully reduced. Second, modern file systems can safely compact their content so that the data in use are contiguous in the logical address space. Log-structure file systems such as `F2FS` [90] support this more readily, but file system defragmentation can also achieve the same effect in in-place update file systems such as `ext4` [110]. Lastly, the file abstraction to the applications thankfully hides

[4] `TRIM` is called `Deallocate` in `NVMe`.

the remaining space left on storage. A file is simply a sequence of bytes, and managing storage capacity is the responsibility of the system administrator. We expected the system support for a capacity-variant SSD to be no more complex than the *zoned namespace* [165] command set that shifts the responsibility of garbage collection to the host side.

This file system support potentially incurs overhead for the file system to relocate data from one logical space to another, as illustrated in Figure 26. In the case for Figure 26a, the high logical address space is unused by the file system, and the capacity-variant SSD gracefully reduce this space with no data relocation overhead for the file system. However, in the case of Figure for Figure 26b, the SSD and the file system must handshake prior to reducing the capacity. Naïvely, the file system would relocate not only the data at the high address space, but also update any metadata for the block allocation and inode. A more advanced commands such as SHARE [127] can be used to reduce the relocation overhead.

4.8. Conclusion

In this work, we examine the performance of wear leveling (WL) in the context of modern flash memory with reduced endurance. Unfortunately, our findings show that existing WL exhibits limited effectiveness and may generate counter-productive results. To explore an alternative strategy, we quantify the benefits of capacity variance in SSDs and find that the lifetime can be extended significantly. We believe our findings confirm the potential for the capacity-variant interface, and that WL may become an artifact of the past.

CHAPTER 5

The Design and Implementation of a Capacity-Variant Storage System

5.1. Introduction

Fail-slow symptoms where components continue to function but experience degraded performance [46, 132] have recently gained significant attention for flash memory-based solid-state drives (SSDs) [103, 104, 183]. In SSDs, such degradation is often caused by the SSD-internal logic's attempts to correct errors [14, 46, 115, 130]. Recent studies have demonstrated that fail-slow drives can cause latency spikes of up to 3.65× [103], and since flash memory's reliability continues to deteriorate over time [65, 103, 183], we expect the impact of fail-slow symptoms on overall system performance to increase.

Figure 27 demonstrates a steady performance degradation for a real enterprise-grade SSD. We age the SSD through random writes by writing about 100 terabytes of data each day, and during morning hours when no other jobs are running, we measure the throughput of the read-only I/O, both sequential and random reads. As shown in Figure 27, the performance of the SSD degrades as the SSD wears out, at a rate of 4.2% and 4.3% of the initial performance for each petabyte written, for random reads and sequential reads, respectively. It is unlikely that the throughput drop is due to garbage collection as (1) this was measured daily over months, and (2) only reads are issued during measurement. By the end, writing a total of 9 petabytes of data to the SSD decreased the throughput by 37% for random reads and 38% for sequential reads.

To address this problem, we start with two key observations. First, flash memory, when it eventually fails, does so in a fail-partial manner. More specifically, an SSD's failure unit

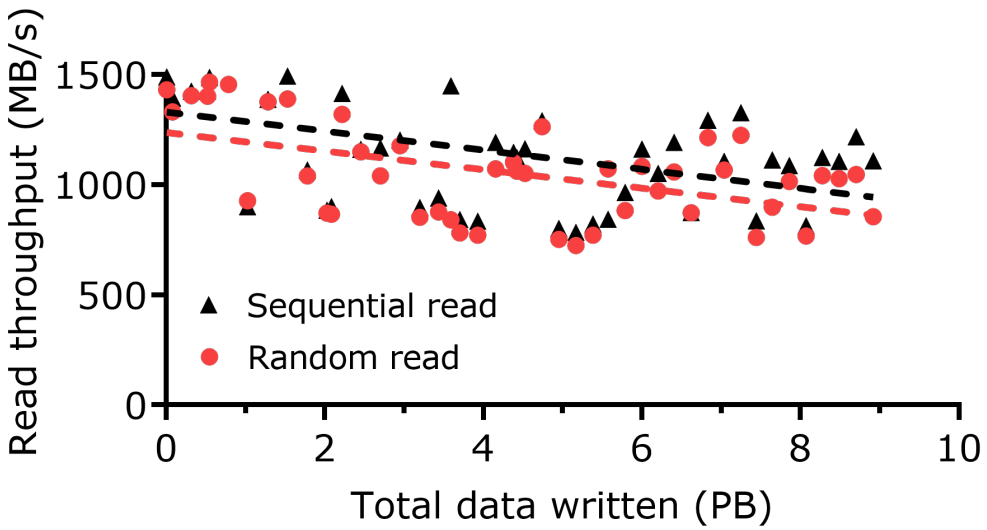


Figure 27: SSD performance degradation due to wear-out. The dashed line represents the linear regression of the daily data points. The throughput decreases by 37% for random reads and 38% for sequential reads after 9 petabytes of data writes.

is an individual flash memory block [14, 115, 130], and the SSD-internal wear leveling algorithms are artifacts to emulate a hard disk drive-like fail-stop behavior [65, 80]. Second, an SSD has no other choice but to trade performance as flash memory's reliability deteriorates, because a storage device's capacity remains fixed and unchanged from its newly installed state until its retirement. SSD's internal data re-reads [15, 16, 106, 134] or preventive re-writes [17, 50] are such choices that lead to fail-slow symptoms [79, 80].

Based on the two key observations, we propose a capacity-variant storage system (CVSS) that maintains high performance even as SSD reliability deteriorates. In CVSS, the logical capacity of an SSD is not fixed; instead, it gracefully reduces the number of exported blocks below the original capacity by mapping out error-prone blocks that would exhibit fail-slow behavior and hiding them from the host. This approach is enabled by the SSD's ability to update data out-of-place. Surprisingly, we find that maintaining a fixed-capacity interface comes at a heavy cost, and reducing capacity counterintuitively extends the lifetime of the device. Our experiments show that, compared to traditional storage systems,

the capacity-variant approach of CVSS outperforms by 49–316% and outlasts by 268–327% under real-world workloads.

We enable capacity variance by designing kernel-level, device-level, and user-level components. The first component is a file system (CV-FS) that dynamically tunes the logical partition size based on the aged state of the storage device. CV-FS is designed to reduce capacity in an online, fine-grained manner and carefully manage user data to avoid data loss. The device-level component, CV-SSD, maintains its performance and reliability by mapping out aged and poor-performing blocks. Without needing to maintain fixed capacity, CV-SSD simplifies flash management firmware, avoids fail-slow symptoms, and extends its lifetime. Lastly, the user-level component, CV-manager, provides necessary interfaces to the host for capacity variance. Users can set performance and reliability requirements for the device through commands, and the CV-manager then adaptively orchestrates CV-FS and the underlying CV-SSD.

The contributions of this paper are as follows.

- We present the design of a capacity-variant storage system that relaxes the fixed-capacity abstraction of the storage device. Our design consists of user-level, kernel-level, and device-level components that collectively allow the system to maintain performance and extend its lifetime. (§ 5.3)
- We develop a framework that allows for a full-stack study on fail-slow symptoms in SSDs over a long time, from start to failure. This framework provides a comprehensive model of SSD internals and aging behavior over the entire lifetime of SSDs[5]. (§ 5.4)

[5]Our framework and extensions are available at https://github.com/ZiyangJiao/FAST24_CVSS_FEMU

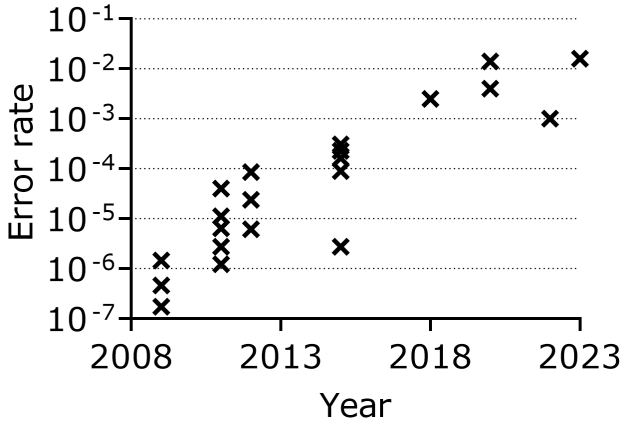


Figure 28: Flash memory error rates have increased significantly over the past years.

- We evaluate and quantitatively demonstrate the benefits of capacity variance using a set of synthetic and real-world I/O workloads throughout the SSD's *entire* lifetime. Capacity variance avoids the fail-slow symptoms and can significantly extend the SSD's lifetime. (§ 5.5)

5.2. Background and Motivation

We first show the increasing trend of flash memory errors in SSDs and describe how flash cells wear out. We then explain how the current storage system abstraction exacerbates reliability-related performance degradation, and summarize prior work for addressing these fail-slow symptoms.

Flash memory errors and wear-out. The rapid increase in NAND flash memory density has come at the cost of reduced reliability and exacerbated fail-slow symptoms. Figure 28 shows the reported flash raw bit error rates (RBERs) in recent publications [14, 15, 36, 79, 106, 145, 150, 178], and this trend indicates that flash memory errors are already a common case.

One of the significant flash memory error mechanisms is wear-out, where flash cells are gradually damaged with repeated programs and erases [115, 130]. Because wear-outs are

irreversible, once a flash block reaches its endurance limit or returns an operation failure, it is marked as bad by the SSD-internal flash translation layer (FTL) and taken out of circulation. To replace these unusable blocks, SSDs are often over-provisioned with more physical capacity than the logically exported capacity.

SSD's wear-outs are caused not only by write I/Os, but also by SSD-internal management such as garbage collection, reliability management, and wear leveling (WL). Although much of the literature has emphasized the role of garbage collection in the SSD's internal writes, studies have revealed that SSD's reliability management and WL also significantly impact the lifetime [65, 70, 79, 108]. WL, in particular, is revealed to be far from perfect, wearing out some of the blocks 6× faster [108] and often leads to counterintuitive acceleration of wear-outs, increasing the write amplification factor as high as 11.49 [65].

Fixed capacity abstraction. Unfortunately, the current storage system abstraction of fixed capacity requires SSDs to implement wear leveling (WL), even if it is imperfect and harmful [65, 108]. Specifically, with the fixed capacity abstraction, the device is not allowed to have part of its capacity fail (i.e., wear out) prematurely, and therefore the SSD has to perform wear leveling to ensure that most, if not all, of its capacity is wearing out at roughly the same rate. If the SSD cannot maintain its original exported capacity when too many blocks become bad, then the entire storage device becomes unusable [130]. This is despite the fact that the SSD internally has a level of indirection and abstracts the physical capacity.

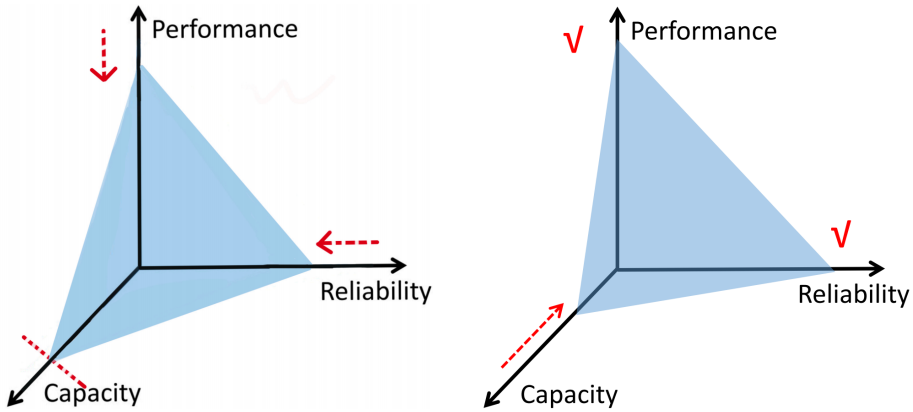
However, the file system provides a file abstraction to the user-level applications, and this abstraction hides the notion of capacity. While utility programs such as `df` and `du` report the storage capacity utilization, file operations such as `open()`, `close()`, `read()`, and `write()` do not expose capacity directly. Instead, the file system manages the storage ca-

pacity using persistent data structures such as superblock and allocation maps to track the utilization of the SSD.

The fixed capacity abstraction used between the file system and storage devices necessitates the implementation of WL on physical flash memory blocks. However, WL leads to an overall increase in wear on the SSD, resulting in a significantly higher error rate as all the blocks age. This, in turn, manifests into fail-slow symptoms in SSDs.

Fail-slow symptoms. Fail-slow symptoms are caused by the SSD's effort to correct errors [15, 16, 106, 134] and prevent the accumulation of errors [17, 50]. Because SSDs are commonly used as the performance tier in storage systems where the identification and removal of ill-performing drives are critical, fail-slow symptoms in SSDs have gained significant attention recently. Prior research in this area can be categorized into three types. The first group focuses on developing machine learning (ML) models to quickly identify SSDs experiencing fail-slow symptoms [19, 55, 171, 183, 184]. Various models, including neural networks [55], autoencoders[19], LSTM [184], feature ranking [171], and random forest [183], have been explored with varying accuracy and efficacy. The second group aims to isolate and remove ailing drives using mechanisms deployed in large-scale systems, identified through ML [103] or system monitoring [54, 132]. The third group proposes modifications to the interface to reject slow I/O and send hedging requests to a different node [53] or drive [96].

Unfortunately, ML-based learning of SSD failures requires an immense number of data points, is often expensive to train, and is only available in large-scale systems [103, 183]. Furthermore, as SSDs evolve and new error mechanisms emerge (e.g., lateral charge spreading [92] and vertical and horizontal variability [146]), older ML models become obsolete, making it difficult to reap the benefits of fail-slow prediction. Most critically, these prior



(a) TrSS (b) CVSS

Figure 29: Comparison between the traditional fixed-capacity storage system (TrSS) and capacity-variant storage system (CVSS). For TrSS (Figure 29a), the performance and reliability degrade as the device ages to maintain a fixed capacity; for CVSS (Figure 29b), the performance and reliability are maintained by trading capacity.

approaches only treat the symptoms and fail to consider the underlying cause: the flash error mechanism.

5.3. Design for Capacity Variance

The high-level design principle behind the capacity-variant system is illustrated in Figure 29. This system relaxes the fixed-capacity abstraction of the storage device and enables a better tradeoff between capacity, performance, and reliability. The traditional fixed-capacity interface, which was designed for HDDs, assumes a fail-stop behavior where all storage components either work or fail at the same time. However, this assumption is not accurate for SSDs since flash memory blocks are the basic unit of failure, and it is the responsibility of the FTL to map out failed, bad, and aged blocks [80, 130].

By allowing a flexible capacity-variant interface, an SSD can gracefully reduce its exported capacity, and the storage system as a whole would reap the following three benefits.

- **Performant SSD even when aged.** An SSD can avoid fail-slow symptoms by

gracefully reducing its number of exported blocks. Error management techniques such as data re-reads [15, 16, 106, 134] and data re-writes [17, 50] would be performed less frequently as blocks with high error rates can be mapped out earlier. This, in turn, reduces the tail latency and lowers the write amplification, making it easier to achieve consistent storage performance.

- **Extended lifetime for SSD-based storage.** An SSD's lifetime is typically defined with a conditional warranty restriction under *DWPD (drive writes per day)*, *TBW (terabytes written)*, or *GB/day (gigabytes written per day)* [159]. With the fixed capacity abstraction, the SSD reaches the end of its lifetime when the physical capacity becomes less than the original logical capacity. Instead, with capacity variance, the lifetime of an SSD would be extended significantly, as it would be defined by the amount of data stored in the SSD, not by the initial logical capacity.

- **Streamlined SSD design.** By adopting the approach of allowing the logical capacity to drop below the initial value, SSD vendors can design smaller and more efficient error correction hardware and their SSD-internal firmware: There is no need to overprovision the SSD's error handling logic or to ensure that all blocks wear out evenly.

Figure 30 illustrates the main components of a capacity-variant system. Enabling the capacity variance feature is achieved by designing the following three components: (1) CV-FS, a log-structured file system for supporting elastic logical partition; (2) CV-SSD, a capacity-variant SSD that maintains device performance and reliability by effectively mapping out aged blocks; and (3) CV-manager, a capacity management scheme that provides the interface for adaptively managing the capacity-variant system.

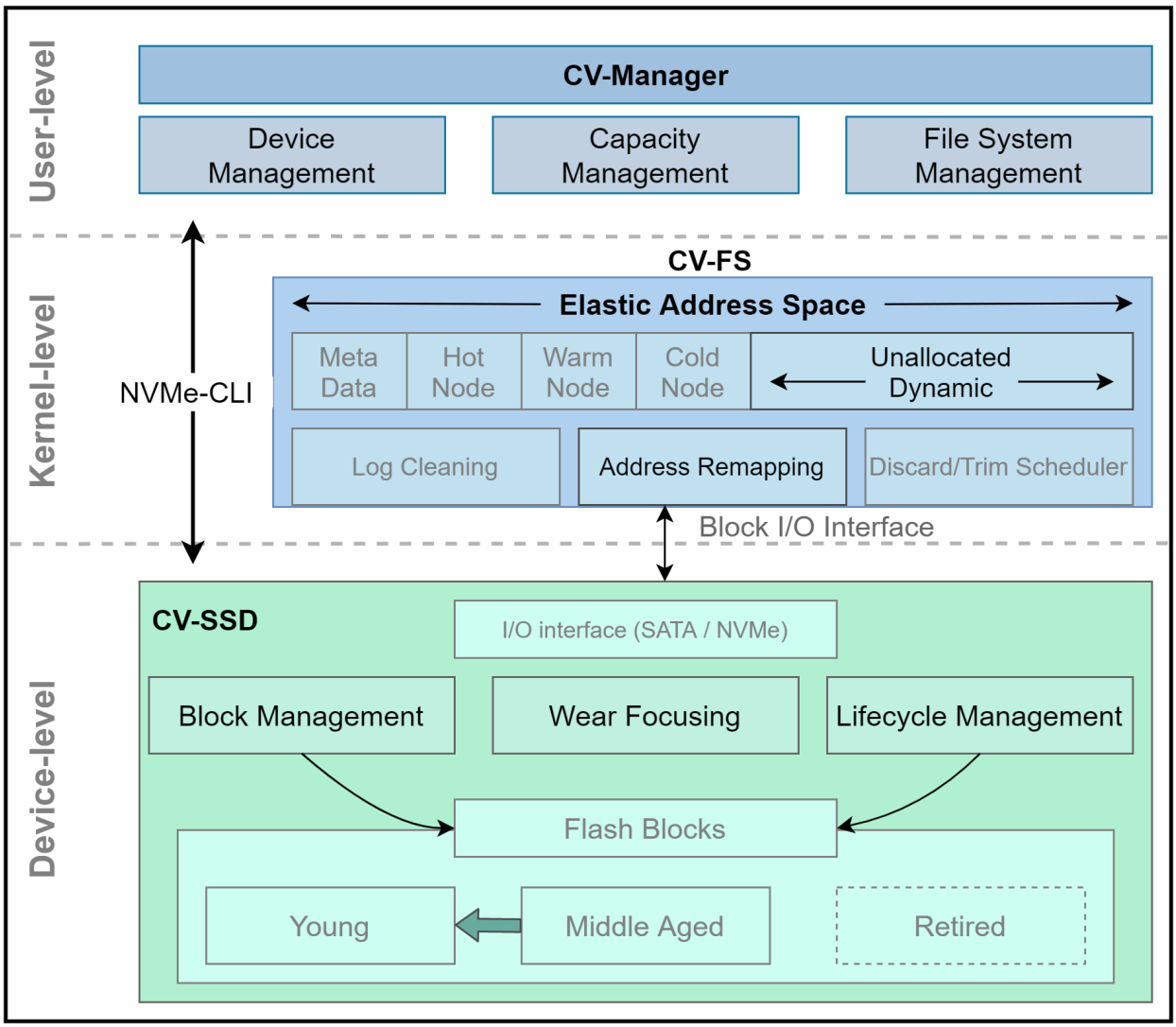


Figure 30: An overview of the capacity-variant system: (1) CV-FS exports an elastic logical space based on CV-SSD's aged state; (2) CV-SSD retires error-prone blocks to maintain device performance and reliability; and (3) CV-manager provides user-level interfaces and orchestrates CV-SSD and CV-FS. The highlighted components are discussed in detail.

5.3.1. Capacity-Variant FS

The higher-level storage interfaces, such as the POSIX file system interface, allow multiple applications to access storage using common file semantics. However, to support capacity variance, the file system needs to be modified. In this section, we discuss the feasibility of an elastic logical capacity based on existing storage abstractions and then investigate

different approaches for supporting capacity variance, Lastly, we describe our new interface for capacity-variant storage systems based on the selected approach.

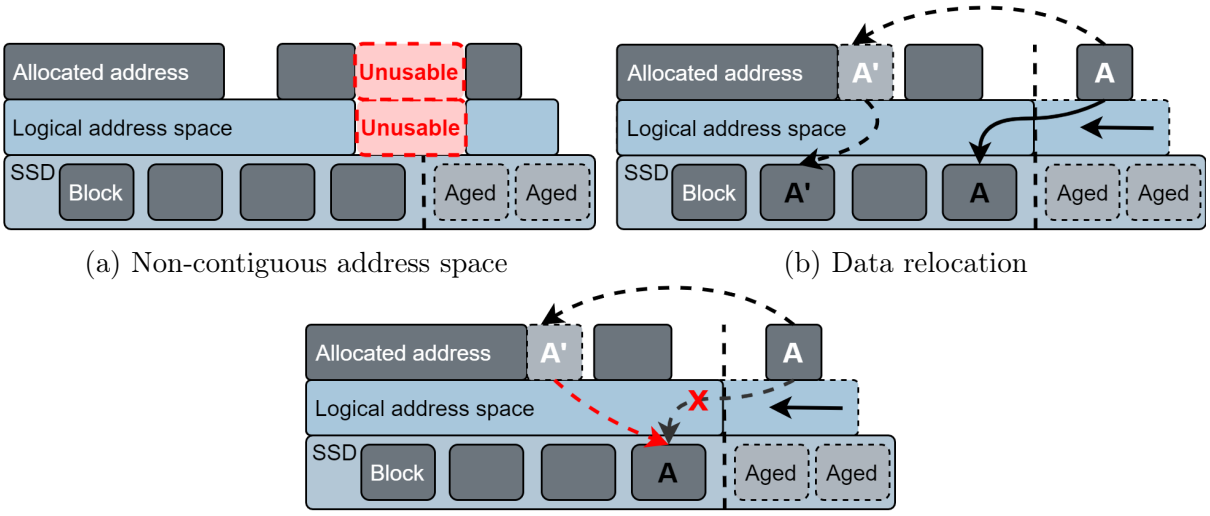
Feasibility of Elastic Capacity

Current file systems assume that the capacity of the storage device does not change and tightly couple the size of the logical partition to the size of the associated storage device. To overcome this limitation, the CV-FS file system declares the entire address space for use at first and then dynamically adjusts the declared space as the storage device ages in an online manner. This is achieved by defining a variable logical partition that is independent of the physical storage capacity.

Thankfully, this transition is feasible for three reasons. First, the TRIM [124] command, which is widely supported by interface standards such as NVMe, enables the file system to explicitly declare the data that is no longer in use. This allows the SSD to discard the data safely, making it possible to reduce the exported capacity gracefully. Second, modern file systems can safely compact their content so that the data in use are contiguous in the logical address space. Log-structured file systems [140] support this more readily, but file system defragmentation [160] techniques can be used to achieve the same effect in in-place update file systems. Lastly, the file abstraction to the applications hides the remaining space left on storage. A file is simply a sequence of bytes, and file system metadata such as utilization and remaining space is readily available to the system administrator.

File System Designs for Capacity Variance

Shrinking the logical capacity of a file system can be a complex procedure that may result in data loss if not done carefully [80]. Most importantly, any valid data within the to-be-shrunk space must be relocated and the process must be coordinated with underlying storage accordingly. Moreover, to ensure users do not need to unmount and remount the device, the logical capacity should be reduced in an online manner, and the time it takes



(a) Non-contiguous address space

(b) Data relocation

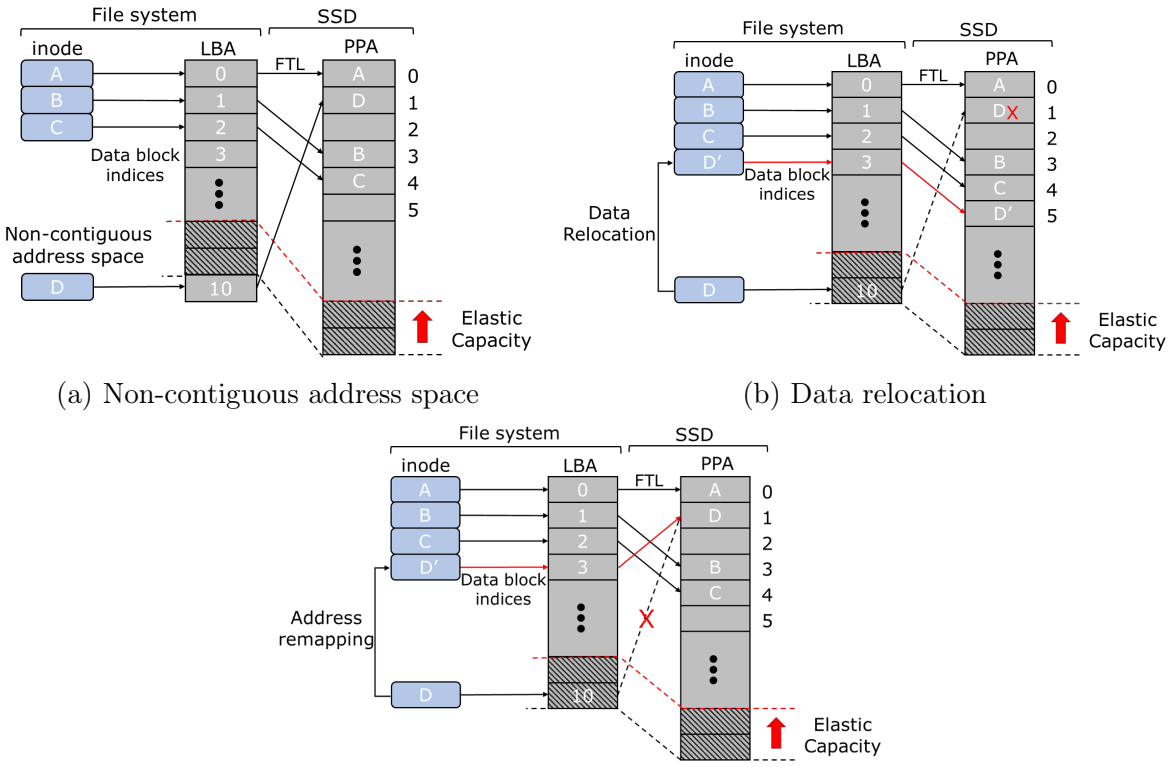
(c) Address remapping

Figure 31: High-level ideas of three different approaches to support capacity variance.

to reduce capacity should be minimal with low overhead.

Figure 31 illustrates three high-level ideas and Figure 32 depicts their implementation approaches to performing online address space reduction: (1) through a non-contiguous address space; (2) through data relocation; and (3) through address remapping. We describe each approach and our rationale for choosing the address remapping (Figure 32c).

- **Non-contiguous address space** (Figure 32a). The file system internally decouples the space exported to users from the LBA. When logical capacity should be reduced, the file system identifies an available range of free space from the end of the logical partition and then restricts the user from using it, for example, by marking that as allocated. With this approach, the adjustment of logical capacity can be efficiently achieved with minimal upfront costs, as the primary task involved is allocating the readily available free space. However, this approach increases the file system cleaning overhead and fragments the file system address space. Due to the negative effect of address fragmentation [29, 30, 51, 63], we avoid this approach despite the lowest upfront cost.



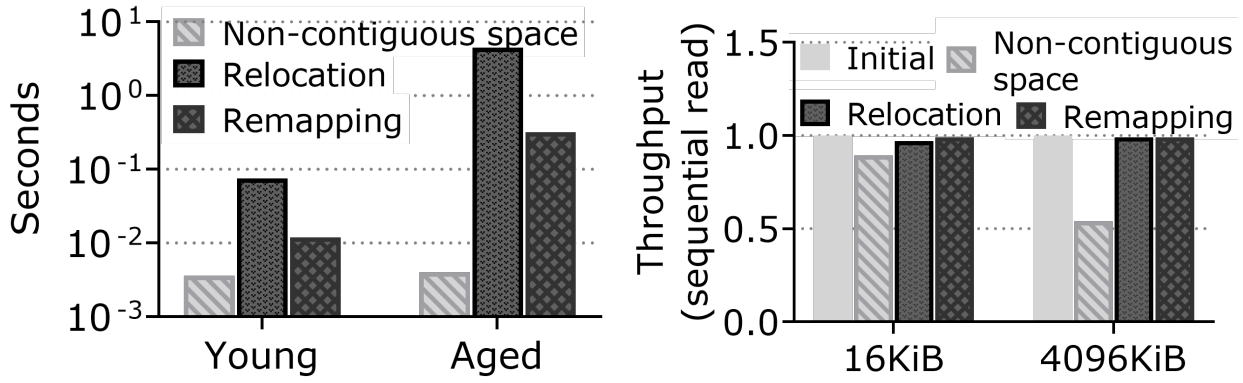
(a) Non-contiguous address space

(b) Data relocation

(c) Address remapping

Figure 32: Design options for capacity variance. In Figure 32a, the FS internally maps out a range of free LBA from the user, causing address space fragmentation. In Figure 32b, the data block is physically relocated to lower LBA. This approach maintains the contiguity of the entire address space but exerts additional write pressure on the SSD. Lastly, in Figure 32c, the data block can be logically remapped to lower LBA. This approach incurs negligible system overhead by introducing a special SSD command to associate data with a new LBA.

- **Data relocation** (Figure 32b). Similar to segment cleaning or defragmentation, the file system relocates valid data within the to-be-shrunk space to a lower LBA region before reducing the capacity from the higher end of the logical partition. This approach maintains the contiguity of the entire address space. Nevertheless, it is essential to note that data relocation exerts additional write pressure on the SSD and the overhead is proportional to the amount of valid data copied. Moreover, user requests are potentially stalled during the relocation process.



(a) Time elapsed for shrinking 1 GiB logical capacity.

(b) Performance comparison before and after shrinking 60 GiB logical capacity.

Figure 33: Performance results for three capacity variance approaches. The address remapping approach introduces lower overhead (Figure 33a) and does not incur fragmentation after shrinking the address space (Figure 33b).

- **Address remapping** (Figure 32c). Data is relocated logically at the file system level without data relocation at the SSD level by taking advantage of the already existing SSD-internal mapping table [127, 185]. While this approach necessitates the introduction of a new SSD command to associate data with a new LBA, it effectively mitigates address space fragmentation and incurs negligible system overhead, as no physical data is actually written.

We implement the three approaches above on F2FS and measure the elapsed time for reducing capacity by 1 GiB. The reported results represent an average of 60 measurements. On average, each measurement resulted in the relocation or remapping of 0.5 GiB of data for the aged file system case and 0.05 GiB of data for the young case. We further compare the performance under the sequential read workload with two I/O sizes (i.e., 16 KiB and 4096 KiB) before and after capacity is reduced. As depicted in Figure 33, the elapsed time required to shrink 1 GiB of logical space on an aged file system is 0.317 seconds when employing the address remapping approach. In contrast, the data relocation approach takes approximately 4.5 seconds. Notably, while the non-contiguous address space approach only

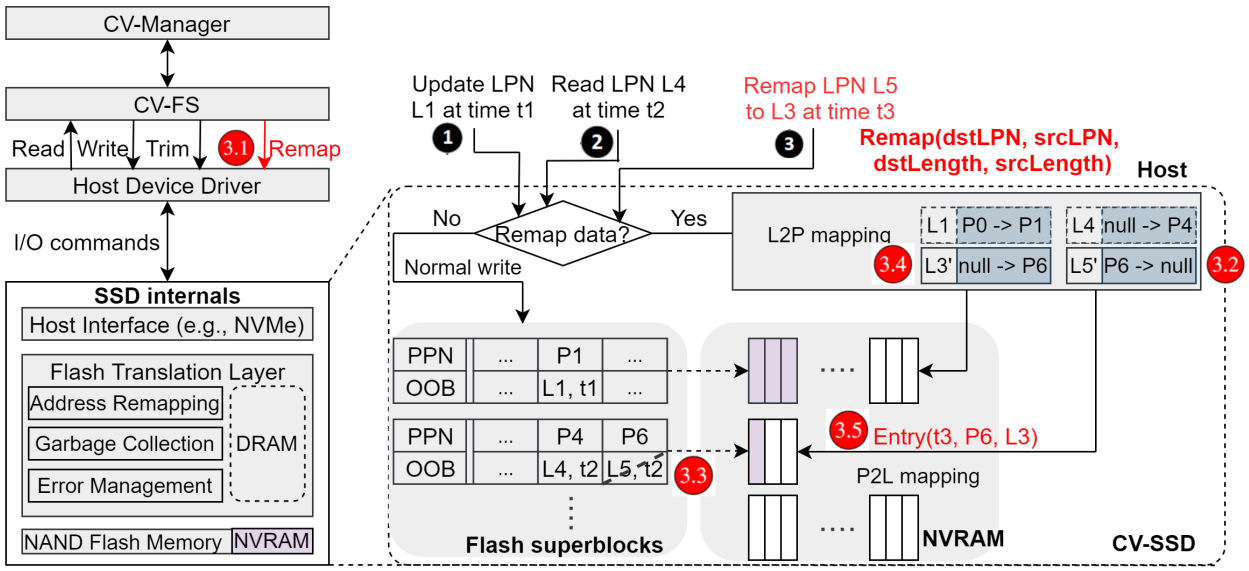


Figure 34: The `REMAP` command workflow for capacity variance: data in the range between `srcLPN` and `srcLPN + srcLength -1` are remapped to logical address starting from `dstLPN`. The third argument, `dstLength`, is optionally used for the file system to ensure I/O alignment.

takes 0.004 seconds, it exhibits significant performance degradation after the capacity reduction, for example, 13% for 16 KiB read and 50% for 4096 KiB read, due to increased fragmentation. We next present the design details of the proposed remapping interface and the capacity reduction process with that.

Interface Changes for Capacity Variance

To integrate the address remapping approach into CV-FS, we revise the interface proposed by prior works [127, 185], `REMAP(dstLPN, srcLPN, dstLength, srcLength)`, and tailor it for capacity-variant storage systems. Our modified command enables file systems to safely shrink the logical capacity with minimal overhead by remapping valid data from their old LPNs to new LPNs without the need for actual data rewriting [127, 185]. We extend the current NVMe interface to include remap as a vendor-unique command.

Figure 34 shows an example of the remap command used for shrinking capacity. Assuming the file system address space ranges from LBA0 – LBA47 at the beginning (i.e., LPN0 –

LPN5 with 512 bytes sector and 4 KiB page size) and LPN5 is mapped to PPN6 within the device. At time t, CV-FS initiates the capacity reduction and identifies that LBA40 – LBA47 (or LPN5) contains valid data. It then issues the remap command to move LPN5 data to LPN3 (i.e., `remap(LPN3, LPN5, 1, 1)`. Upon receiving the remap command, the FTL first finds the PPN associated with LPN5 (PPN6 in our case) and updates the logical-to-physical (L2P) mapping of L3 to PPN6. Finally, the old L2P mapping of L5 is invalidated, and the new physical-to-logical (P2L) mapping of PPN6 is recorded in the NVRAM of the SSD. Once the to-be-shrunk space is free, the file system states are validated and a new logical capacity size is updated.

In particular, the required size for NVRAM is small (for example, 1 MiB for a 1 TiB drive as suggested by the prior work [127]), as it is only used to maintain a log of the remapping metadata. Assuming the SSD capacity and page size are 1TB and 4KB, respectively, a single remapping entry requires no more than 8B space. The 1 MiB NVRAM would be sufficient to hold entries for 512 MiB remapped data during a capacity reduction event. Between capacity reduction events, the reclamation of a flash block/page will cause a passive recycle on the associated remapping entries. However, in cases where a larger buffer is needed, the log can perform an active cleaning or write part of the mappings to flash because the space allocated for internal metadata will be conserved with a smaller device capacity [47]. Alternatively, the need for NVRAM can be eliminated by switching to the data relocation method, but at the cost of a higher overhead for logical capacity adjustment.

As a result, this new interface does not require taking the file system and device offline to adjust address space since data are managed logically. Moreover, it does not complicate the existing file system consistency management scheme. Similar to other events such as `discard`, the file system periodically performs checkpoints to provide a consistent state.

The crash consistency is examined by manually crashing the system after initiating the remapping command and CrashMonkey [118] with its pre-defined workloads [117].

5.3.2. Capacity-Variant SSD

In this section, we outline design decisions and their leading benefits for building a capacity-variant SSD. We first discuss the necessity to forgo wear leveling in CV-SSD, and then describe its block management and life cycle management for extending lifetime and maintaining performance. Lastly, we introduce a degraded mode to handle the case where the remaining physical capacity becomes low.

Note that blocks in this subsection refer to physical flash memory blocks, different from the logical blocks managed by the file system. Furthermore, the flash memory blocks are grouped and managed as superblocks (again, different from the file system's superblock) to exploit the SSD's parallelism.

Wear Focusing

The goal of a capacity-variant SSD is to keep as much flash as possible at peak performance and mitigate the impact of underperforming and aged blocks. A capacity-variant SSD would maintain both performance and reliability by gracefully reducing its exported capacity so aged blocks can be mapped out earlier. Therefore, a capacity-variant SSD does not perform wear leveling (WL), as it degrades all of the blocks over time. WL is an artifact designed to maintain an illusion of a fixed-capacity device wherein its underlying storage components (i.e., flash memory blocks) either all work or fail, opposing our goal of allowing partial failure.

Moreover, static WL [20, 25, 42] incurs additional write amplification due to data relocation within an SSD. Dynamic WL [26, 27], on the other hand, typically combines with SSD internal tasks such as garbage collection, reducing the overall cleaning efficiency as

its victim selection considers both the valid ratio and wear state. A recent large-scale field study on millions of SSDs reveals that the WL techniques in modern SSDs present limited effectiveness [108] and an analysis study demonstrates that WL algorithms can even exhibit unintended behaviors by misjudging the lifetime of data in a block [65]. Such counterproductive results are avoided by forgoing WL and adopting capacity variance.

Block Management

A capacity-variant SSD exploits the characteristics of flash memory blocks to extend its lifetime and meet different performance and reliability requirements. Flash memory blocks in SSDs wear out at different rates and are marked as bad blocks by the bad block manager when they are no longer usable [67, 130]. This means that the physical capacity of the SSD naturally reduces over time, and for a fixed-capacity SSD, the entire storage device is considered to have reached the end of its life when the number of bad blocks exceeds its reserved space. On the other hand, the capacity-variant SSD's lifetime is defined by the amount of data stored in the SSD, rather than the initial logical capacity, making it a more reliable and efficient option.

The fail-slow symptoms and performance degradation in SSDs are caused by aged blocks with high error rates [15, 16, 106, 134]. Traditional SSDs consider blocks as either good or bad and such coarse-grained management fails to meet different performance and reliability requirements. On the other hand, the capacity-variant SSD defines three states of blocks: young, middle-aged, and retired, based on their operational characteristics. Young blocks have a relatively low erase count and a low RBER, while middle-aged blocks have higher errors and require advanced techniques to recover data. Retired blocks that are worn out or have a higher RBER than the configured threshold (5% by default) are excluded from storing data.

This block management scheme allows the capacity-variant SSD to map out underper-

forming and unreliable blocks earlier, effectively trading capacity for performance and reliability. In general, blocks start from a young state and transition to middle-aged and retired states. However, a block can also transition from a middle-aged state back to a young state since transient errors (i.e., retention and disturbance) are reset once the block is erased [79].

Life Cycle Management

A capacity-variant SSD requires wear focusing to mitigate the impact of aged flash memory blocks. However, simply avoiding wear leveling is insufficient as there are two processes affecting the life cycle of a flash memory block: **block allocation** and **garbage collection (GC)**. Traditional policies such as youngest-block-first for allocation and cost-benefit for GC work well on a traditional SSD, but are not suitable for CV-SSDs, since they aim to achieve a uniform wear state among blocks. Implementing these policies can cause a large number of blocks with the same erase count to map out simultaneously, leading to excessive capacity loss, and the device may suddenly fail. Figure 35 demonstrates this issue, where over 60% of the blocks aggregate to a particular wear state. Excessive capacity loss can increase the write amplification factor (WAF), particularly when the device utilization rate is high.

Allocation policy. In order to make wear accumulate in a small subset of blocks and allow capacity to shrink gradually, CV-SSD will prioritize middle-aged blocks to accommodate host writes and young blocks for GC writes. Since retired blocks are not used, there are four scenarios when considering data characteristics.

- Write-intensive data are written to a middle-aged block
- Write-intensive data are written to a young block
- Read-intensive data are written to a middle-aged block

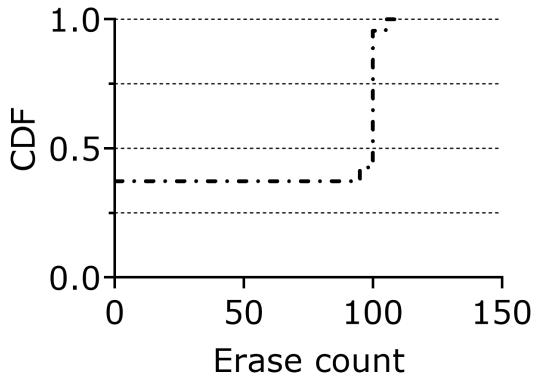


Figure 35: The wear distribution for a 256 GiB SSD under 100 iterations of MS-DTRS workload [78]. Traditional GC and block allocation policies cause a sudden capacity loss as too many blocks are equally aged.

- Read-intensive data are written to a young block

Type I and type IV are ideal cases as they help to converge the wear among blocks without affecting the performance. Type II will also not affect the performance when data are fetched by the host because of the low RBER of young blocks. Moreover, with CV-SSD's allocation policy, such write-intensive data are inevitably re-written by the file system to the middle-aged blocks and the type II blocks will be GC-ed due to their low valid ratio. This type of scenario also happens under the early stages of CV-SSD, in which most blocks are young. Lastly, type III is the case where we need to pay more attention: read-intensive data should be stored in young blocks; otherwise expensive error correction techniques are triggered more often.

Garbage collection. We modify the garbage collection policy to consider (in)valid ratio, aging status, and data characteristics to handle type III cases. The block with the highest

score will be selected as the victim based on the following formula:

$$\begin{aligned} Victim\ score &= W_{invalidity} \cdot invalid\ ratio \\ &\quad + W_{aging} \cdot aging\ ratio \\ &\quad + W_{read} \cdot read\ ratio \\ invalid\ ratio &= \frac{\#\ of\ invalid\ pages}{\#\ of\ valid\ pages + \#\ of\ invalid\ pages}, \\ aging\ ratio &= \frac{erase\ count}{endurance}, \\ read\ ratio &= \frac{\#\ of\ host\ read\ designated\ to\ the\ current\ block}{maximum\ host\ read\ among\ unretired\ blocks}. \end{aligned} \tag{5.1}$$

$W_{invalidity}$, W_{aging}, and W_{read} are weights to balance WAF, the aggressiveness of wear focusing, and the sensitivity of preventing type III scenarios, respectively. With that, read-intensive data stored in aged blocks are relocated by GC. Considering the read ratio could potentially affect the garbage collection efficiency. To avoid low GC efficiency, we set $W_{invalidity} = 0.4$, $W_{aging} = 0.3$, and $W_{read} = 0.3$, and their sensitivity analysis is shown in § 5.5.4. Increasing W_{read} is unfavorable not only because of adverse effects on WAF but also due to introducing unnecessary data movement. For example, a middle-aged block containing many valid pages but experiencing only a minimal number of reads is selected as the victim.

Degraded Mode

During normal conditions, CV-SSD intentionally unevens the wear state among blocks. As error-prone blocks retire, the physical capacity decreases gradually and performance is maintained. However, the physical capacity could decrease to a level where it will be insufficient to maintain current user data. Moreover, it can also cause high garbage collection overhead. In this case, $CV_{degraded}$ mode will be triggered and CV-SSD will slow down the further capacity loss. It is noteworthy that the triggering of degraded mode indicates a

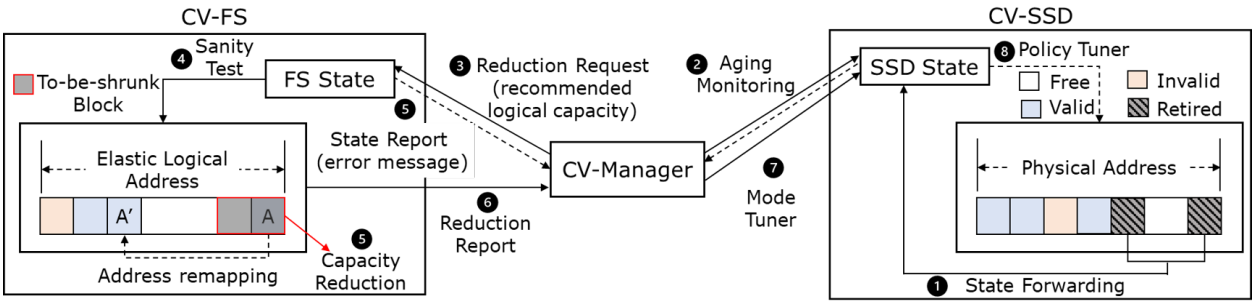


Figure 36: CV-manager design diagram. CV-manager monitors CV-SSD's aged state (Steps 1 and 2) and provides a recommended logical capacity to CV-FS (Step 3). After capacity reduction (Steps 4–6), CV-manager notifies CV-SSD (Step 7). The $CV_{degraded}$ mode will be triggered if the reduction fails (Step 8).

low remaining capacity to trade for performance, and storage administrators can gradually upgrade storage systems.

In particular, the $CV_{degraded}$ mode is triggered under two conditions: (1) when the effective over-provisioning (EOP), calculated as $EOP = (physical\ capacity - utilization)/utilization$, falls below the factory-set over-provisioning (OP), or (2) when the remaining physical capacity is less than a user-defined watermark.

Once $CV_{degraded}$ mode is set, GC only considers WAF and aging to slow down further capacity loss. This mode allows young blocks to be cleaned with a relatively higher valid ratio than aged blocks. Specifically, young blocks with a high invalid ratio are optimal candidates. Moreover, middle-aged blocks are used to accommodate GC-ed data, and young blocks are allocated for host writes. As a result, blocks are used more evenly than in the initial stage and a particular amount of physical capacity is maintained for the user. When EOP becomes greater than OP if the host decides to move or delete some data, $CV_{degraded}$ will be reset by CV-manager.

5.3.3. Capacity-Variant Manager

To improve usability, CV-manager is responsible for automatically managing the capacity of the whole storage system. As illustrated in Figure 36, CV-manager monitors the aged state of the underlying storage device and provides a recommended logical partition size to the kernel.

Specifically, when CV-SSD maps out blocks and its physical capacity is reduced, CV-manager will get notified (Steps 1 and 2). The CV-manager figures out a recommended logical capacity by checking the current bad capacity within the device and issues capacity reduction requests to CV-FS through a system call (Step 3). Upon request, CV-FS performs a sanity test. If the file system checkpoint functionality is disabled or the file system is not ready to shrink (i.e., frozen or read-only), the reduction will not continue (Step 4). Otherwise, CV-FS starts shrinking capacity as described in § 5.3.1 and returns the execution result (Steps 5 and 6). Lastly, CV-manager notifies CV-SSD whether logical capacity is reduced properly or not. If the reduction fails, the $CV_{degraded}$ is activated to slow down further capacity loss (Steps 7 and 8).

For user-level capacity management, CV-manager provides necessary interfaces for users to explicitly initiate capacity reduction and set performance and reliability requirements for the device. The CV-SSD would retire blocks based on the host requirement. Similar to the read recovery level (RRL) command [124] in the NVMe specification that limits the number of read retry operations for a read request, this configurable attribute limits the maximum amount of recovery applied to a request and thus balances the performance.

5.4. Implementation

The capacity-variant file system (CV-FS) is implemented upon the Linux kernel v5.15. CV-FS uses F2FS [90] as the baseline file system due to its virtue of being a log-structure

file system. We modify both CVSS and TrSS to employ a more aggressive discard policy than the baseline F2FS (i.e., 50ms interval if candidates exist and 10s max interval if no candidates) for better SSD garbage collection efficiency [84] (also shown in § 5.5.2).

To implement the `remap` command, we extend the block I/O layer. A new I/O request operation `REQ_OP_REMAP` is added to expose the remap command to the CV-FS. New attributes including `bio->bi_iter.bi_source_sector` and `bio->bi_iter.bi_source_size` are introduced in `bvec_iter`, which corresponds to the second and last parameter of the remap command. Functions related to `bio` splitting/merging procedure (e.g., `__blk_queue_split`) are modified to maintain added attributes (mainly in `/block/blk-merge.c`). Additionally, new `nvme_opcode` and related functions are added to support the remap command at the NVMe driver layer (mainly in `/block/blk-mq.c` and `/drivers/nvme/host/core.c`).

The capacity-variant SSD is built on top of the FEMU [95]. SSD reliability enhancement techniques such as ECC and read retry ensure data integrity. To implement the error model, we use the additive power-law model proposed in prior works [79, 101, 115] that considers wear, retention loss, and disturbance to quantify RBER, as shown in the following equation:

$$\begin{aligned} &RBER(cycles, time, reads) \\ &\quad = \varepsilon + \alpha \cdot cycles^{k} && (wear) \\ &\qquad + \beta \cdot cycles^{m} \cdot time^{n} && (retention) \\ &\qquad + \gamma \cdot cycles^{p} \cdot reads^{q} && (disturbance) \end{aligned} \tag{5.2}$$

The parameters used are particular to a real 2018 TLC flash chip [79], and the device internally keeps track of cycles, time, and reads for each block. During a read operation, read retry is applied if the error exceeds the ECC strength. We consider each read retry will lower the error rate by half [79, 134] and the maximum amount of recovery is limited

for a single read retry so that blocks have more fine-grained error states.

We modify five major software components to support capacity variance.

- We make changes to the Linux kernel v5.15 to provide an `ioctl`-based user-space API supporting logical partition reduction. Users can specify the shrinking size and issue capacity reduction commands through this API.
- We modify the F2FS to handle address remapping triggered by capacity variance and revise its discard scheme.
- We extend the `f2fs-tool` (f2fs format utility) to support the CV-specific functionalities, such as initializing a variable logical partition and updating the attributes that control discard policies.
- We implement CV-SSD mode in FEMU, adding flash reliability enhancement techniques, error models, wear leveling, bad block management, and device lifetime features.
- We modify NVMe device driver and introduce new commands to `NVMe-Cli` [125], to support capacity variance. The SMART [124] command is also extended to export more device statistics for capacity management.

5.5. Evaluation of Capacity Variance

We first describe our experimental setup and methodology, then present our evaluation results and demonstrate the effectiveness of capacity variance.

5.5.1. Experimental Setup and Methodology

Table 5 outlines the system configurations for our evaluation. For the traditional SSD, an adaptive WL, PWL [25], is used to even the wear among blocks. The error correction

Table 5: System configurations. Wear leveling (PWL [25]) and youngest block first allocation are used for traditional SSDs.

PC platform			
Parameter	Value	Parameter	Value
CPU name	Intel Xeon 4208	Frequency	2.10GHz
Number of cores	32	Memory	1TiB
Kernel	Ubuntu v5.15	ISA	X86_64
FEMU			
Parameter	Value	Parameter	Value
Channels	8	Physical capacity	128 GiB
Luns per channel	8	Logical capacity	120 GiB
Planes per lun	1	Over-provisioning	7.37%
Blocks per plane	512	Garbage collection	Greedy
Pages per block	1024	Program latency	500 μs
Page size	4 KiB	Read latency	50 μs
Superblock size	256 MiB	Erase latency	5 ms
Endurance	300	Wear leveling	PWL [25]
ECC strength	50 bits	Block allocation	Youngest first

code (ECC) for both Tr-SSD and CV-SSD is configured to tolerate up to 50-bit errors per 4 KiB data, and errors beyond the correction strength are subsequently handled by read retry. We use 17 different workloads in our evaluation: (1) 4 FIO [61] workloads (Zipfian and random, each with two different utilization); (2) 3 Filebench [161] workloads; (3) 2 YCSB workloads [12] (YCSB-A and YCSB-F); and (4) 8 key-value traces from Twitter [175].

We compare CVSS with three different techniques: (1) TrSS, a traditional storage system with vanilla F2FS plus a fixed-capacity SSD; (2) AutoStream [174]; (3) ttFlash [173]. The evaluation comparisons are selected based on their broader applicability and implementation simplicity of the multi-stream interface (represented by AutoStream [174]) and the

fast-fail mechanism (represented by ttFlash [173]). These approaches align with more general and widely used methods such as PCStream [85], LinnOS [55], and IODA [96]. Specifically, AutoStream [174] uses the multi-stream interface [76] and automatically assigns a stream ID to the data based on the I/O access pattern. The SSD then places data accordingly based on the assigned ID to reduce write amplification and thus, improve performance. On the other hand, ttFlash [173] reduces the tail latency of SSDs by utilizing a redundancy scheme (similar to RAID) to reconstruct data when blocked by GC. Since the original ttFlash is implemented on a simulator, we implement its logic in FEMU for a fair comparison.

To perform a more realistic evaluation, it is necessary to reach an aged FS and device state. Issuing workloads manually to the system is prohibitively expensive, as it takes years' worth of time. Moreover, this method lacks standardization and reproducibility, making the evaluation ineffective [1]. We extensively use aging frameworks in our evaluation. Prior to each experiment, we use impression [1] to generate a representative aged file system layout. After file system aging, the fast-forwardable SSD aging framework (FF-SSD) [67] is used to reach different aged states for SSD. The aging acceleration factor (AF) is strictly limited to 2 to maintain accuracy. Workloads will run until the underlying SSD fails.

We design the experiments with the following questions:

- Can CVSS maintain performance while the underlying storage device ages? (§ 5.5.2)
- Can CVSS extend the device lifetime under different performance requirements? (§ 5.5.3)
- What are the tradeoffs in CVSS design? (§ 5.5.4)

5.5.2. Performance Improvement

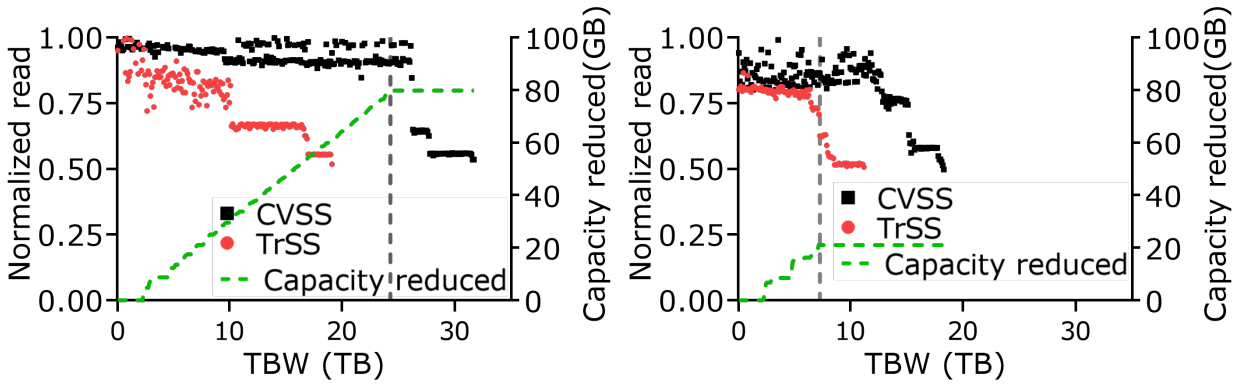
In this section, we evaluate the effectiveness of CVSS in maintaining performance and avoiding fail-slow symptoms under synthetic and real workloads.

FIO

We first examine the performance benefit of capacity variance under Zipfian workloads with two different workload sizes: 38GB (utilization of 30%) and 90GB (utilization of 70%). For this experiment, FIO continuously issues 16KB read and write requests to the device. We use the default setting of FIO and the read/write ratio is 0.5/0.5.

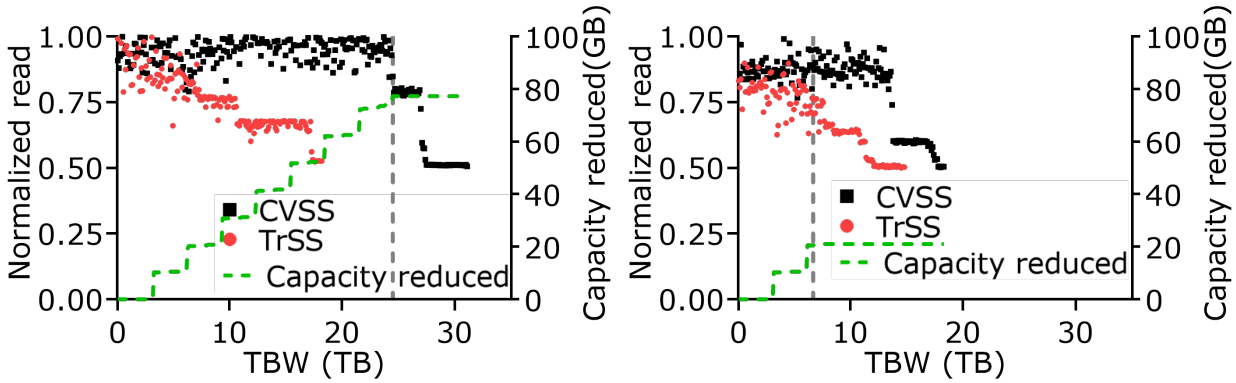
Zipfian. Figure 37 shows the read throughput under different aged states of TrSS and CVSS, in terms of terabytes written (TBW). We measure the performance until it drops below 50% of the initial value where no aging-related operations are performed. The green dotted line shows the amount of physical capacity that has been reduced within the CV-SSD and the straight vertical line represents the trigger of $CV_{degraded}$.

We observe that TrSS and CVSS behave similarly at first where both CV-SSD and Tr-SSD are relatively young. However, for TrSS, the read performance degrades gradually. As Tr-SSD gets aged, the amount of error corrected during each read operation increases and thus involves more expensive read retry processes. On the other hand, CV-SSD effectively trades the capacity for performance. The performance is maintained by excluding heavily aged blocks from use. Later, $CV_{degraded}$ is triggered to maintain a particular amount of capacity for the workloads. During this stage, blocks are used more evenly and the wear accumulates within the device. In this case, performance is traded for capacity in order to avoid data loss. However, even in this mode, CVSS delivers better performance compared to TrSS, thanks to the previous mapping out of most unreliable blocks. Overall, the read throughput of CVSS outperforms TrSS by up to 0.72× with the same amount of host



(a) Zipfian (utilization 30%) (b) Zipfian (utilization 70%)

Figure 37: Read throughput under FIO Zipfian workloads. In CVSS, the performance is maintained by trading capacity. The straight vertical line represents the trigger of the $CV_{degraded}$ mode. After $CV_{degraded}$, the future capacity reduction is slowed down but the performance is compromised.

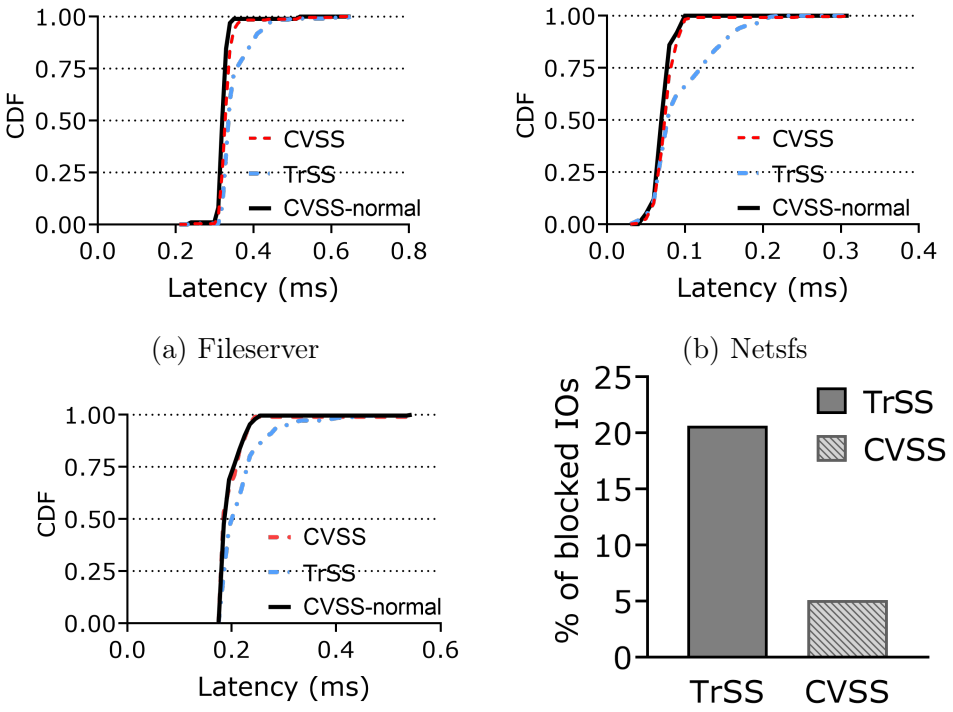


(a) Random (utilization 30%) (b) Random (utilization 70%)

Figure 38: Read throughput under FIO random workloads. CVSS delivers up to 0.6× (left) and 0.7× (right) higher performance compared to TrSS, under the same amount of host writes.

writes.

Random. Figure 38 shows the measured read performance under random I/O. The configuration is similar to the previous case. As in-used blocks get aged, the read performance of TrSS degrades gradually and the fail-slow symptoms manifest. With the same amount of host writes, CVSS delivers a 0.6× and a 0.7× higher throughput at most than TrSS un-



(a) Fileserver

(b) Netsfs

(c) Varmail

(d) Host I/Os blocked by error handling under varmail

Figure 39: Performance results under Filebench workloads. CVSS reduces the average latency by 8% under fileserver workload (Figure 39a), 24% under netsfs workload (Figure 39b), and 10% under varmail workload (Figure 39c) compared to TrSS throughout the devices' lifetime. Before $CV_{degraded}$ is triggered, CVSS–normal reduces the average latency by 32% under netsfs workload. Figure 39d shows the percentage of host I/Os blocked by read retry operations under varmail workload. Other workloads show a similar pattern.

der the utilization of 30% and 70%, respectively.

Figure 40 compares the average write performance over the measurement. For Tr-SSD, when WL is initiated, data are relocated within the device, which decreases the throughput by 0.6×. Without WL, CV-SSD provides a more stable and better write performance than Tr-SSD. Overall, the write throughput of CVSS outperforms TrSS by 0.12× on average.

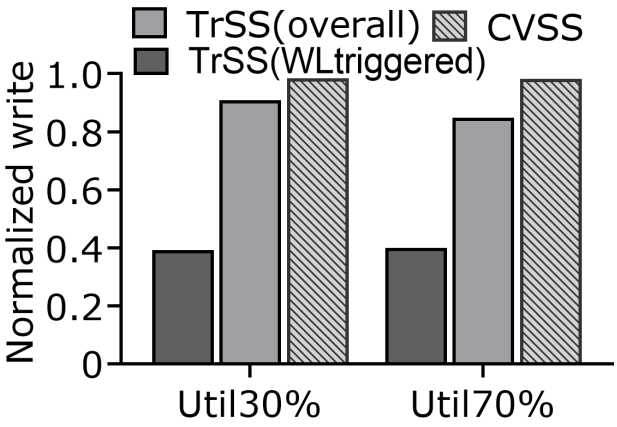


Figure 40: Average write throughput under FIO workloads. For TrSS, when wear leveling is triggered, the write throughput drops by 0.6×; on the other hand, by forgoing WL, CV-SSD provides a more stable and better write performance.

Filebench

We now use Filebench [161] to evaluate the capacity-variant system under file system metadata-heavy workloads. We use three pre-defined workloads in the benchmark, which exhibit differences in I/O patterns and fsync usage.

Figures 39a, 39b, and 39c show the CDF of operation latency under fileserver, netsfs, and varmail workloads throughout the devices' life. In particular, CVSS–normal represents the result before $CV_{degraded}$ is activated and CVSS represents the overall result. We use the default setting of Filebench, which measures the performance by running workloads for 60 seconds. Random writes are used to age CV-SSD and Tr-SSD. The measurement is performed after every 100GB of random data written until the device fails. The utilization for both TrSS and CVSS is 50%.

Compared to TrSS, CVSS reduces the average response time by 32% before the degraded mode is triggered and by 24% over the entire lifetime under netsfs workload, as shown in Figure 39b. The netsfs workload simulates the behavior of a network file server. It performs a more comprehensive set of operations such as application lunch, read-modify-

write, file appending, and metadata retrieving, and thus reflects the state of the underlying devices more intuitively. Overall, CVSS reduces the average latency by 8% in the fileserver case (Figure 39a), and 10% in the varmail case (Figure 39c).

CV-SSD maps out blocks once their RBER exceeds 5% by default, while Tr-SSD only maps them out when their erase counts exceed the endurance limit, leading to more expensive error correction operations. The increased error correction operations not only affect the latency of the ongoing host request but also create backlogs in IO traffic. Figure 39d shows the percentage of host I/Os blocked by read retry operations measured inside FEMU under the varmail workload. In TrSS, more than 20% of I/O requests are delayed by SSD internal read retry, while it is no more than 5% in CVSS.

Twitter Traces

The previous sections examine CVSS using block I/O workloads and file system metadata-heavy workloads. In this section, we evaluate CVSS and compare it with AutoStream [174] and ttFlash [173] at the overall application level. We use a set of key-value traces from Twitter production [175]. The Twitter workload contains 36.7 GB worth of key-value pairs in total. We first load the key-values pairs and then start and keep feeding the traces to RocksDB until the underlying SSD fails.

Figure 41 compares the average KIOPS over the entire device lifetime. Overall, capacity variance improves the throughput by 0.49× – 3.16× compared to TrSS; on the other hand, AutoStream and ttFlash present limited effectiveness in mitigating fail-slow symptoms. In particular, Trace38 highlights the benefits of capacity variance, achieving a 3.16× better throughput than the fixed-capacity storage. For RocksDB, point lookups may end up consulting all files in level 0 and at most one file from each of the other levels. Therefore, as the Tr-SSD ages, a single `Get()` request can cause multiple physical reads and each of them can trigger SSD read retry several times, degrading the read performance drastically.

Moreover, we find that traditional systems with the original discard policy show higher utilization inconsistency (i.e., $\frac{1}{n}\sum_{Observation=1}^{n} util_{SSD} - util_{FS}$) between FS and SSD, as shown in Figure 43. That is because of the high request rate during the experiments and F2FS only dispatches discard command when the device I/O is idle, which not only decreases SSD GC efficiency but also makes wear leveling more likely to misjudge data aliveness, limiting its effectiveness in maintaining capacity. During the experiments, the WAF of TrSS can be as high as 6.79, while only 1.12 for CVSS.

5.5.3. Lifetime Extension

In this section, we investigate how CVSS extends device lifetime given different performance requirements and thus leads to a longer replacement interval for SSD-based storage systems. We compare three different configurations: CVSS, TrSS, and AutoStream [174] in this evaluation since ttFlash introduces additional write (wear) overhead coming from RAIN (Redundant Array of Independent NAND) even for a small write [173]. The workloads used are similar to § 5.5.2.

Figure 42 shows the TBW before the device performance drops below 0.8, 0.6, 0.4, and 0 of the initial state. In particular, 0 represents the case where no performance requirement is applied so the workload runs until the underlying SSD is unusable. In cases of

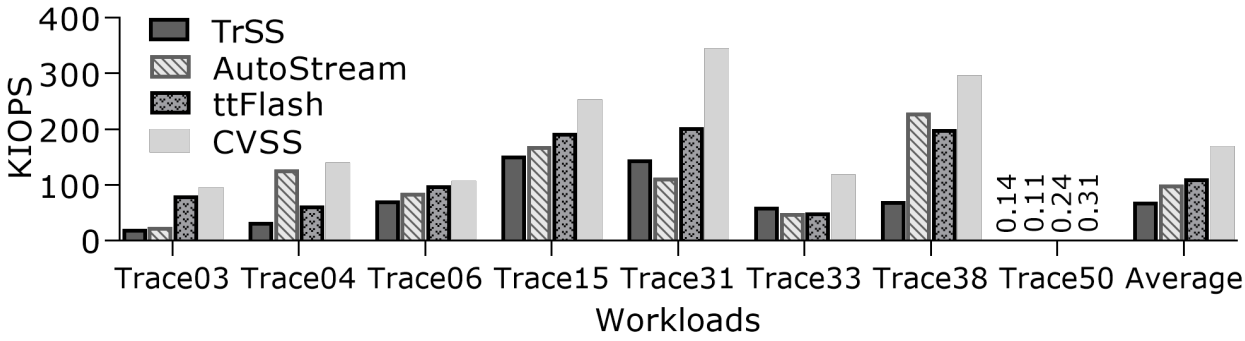


Figure 41: Performance results under Twitter traces. Capacity variance outperforms AutoStream and ttFlash and improves the throughput by 1.42× on average compared to TrSS.

lower device utilization (as shown in Figures 42a and 42c), CVSS effectively extends the device lifetime, even when high performance is required. In Figure 42a, the device fails after accommodating 10 TB host writes for TrSS and 18 TB for AutoStream, considering the performance requirement of 0.8. On the other hand, CVSS accommodates 28 TB host writes with the same performance requirement, outlasting TrSS by 180% and AutoStream by 55%. Similarly, in Figure 42c, CVSS outlasts TrSS by 270% and AutoStream by 50%.

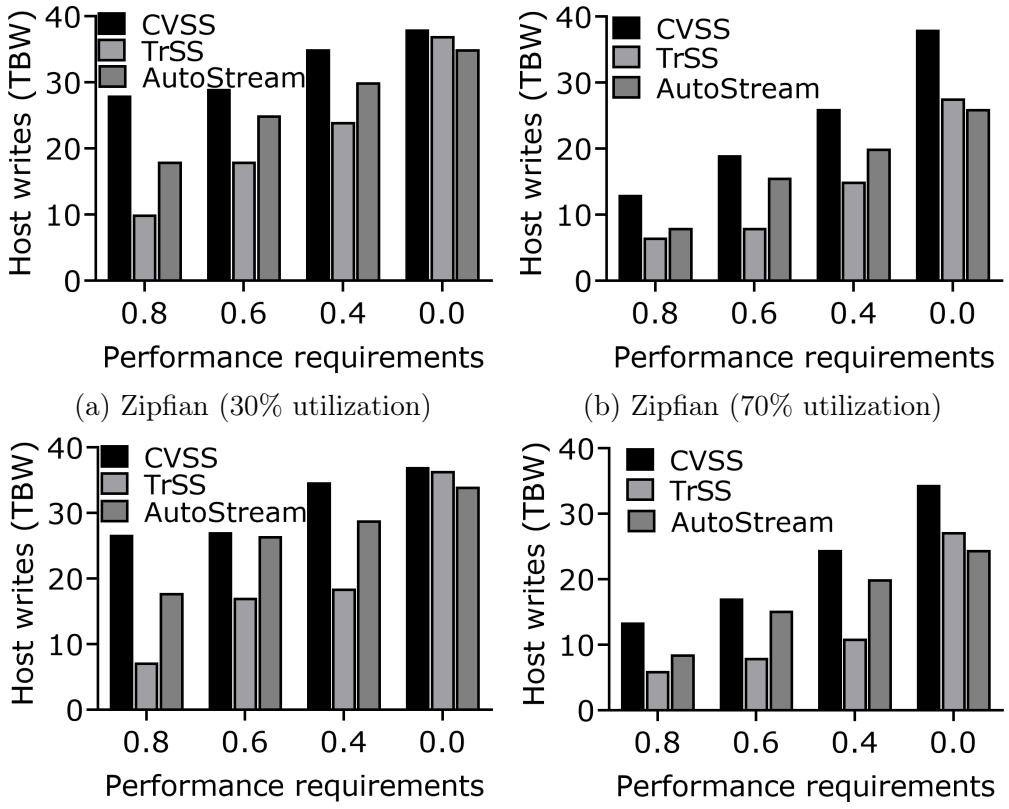
In the high device utilization cases (as shown in Figure 42b and 42d), CVSS outlasts TrSS by 123% and AutoStream by 55% on average with the highest performance requirement. In Figure 42b, before the device becomes unusable, CVSS accommodates 10.4 TB more in host writes compared to TrSS and 12 TB more compared to AutoStream. In our experiments, we found AutoStream achieves a longer lifetime than TrSS except for the no performance requirement case. In AutoStream, data are placed based on their characteristics, which in turn triggers more data relocation towards the end for wear leveling. Overall, with the highest performance requirement, CVSS ingests 168% host data more compared to TrSS and 57% more compared to AutoSteam on average, which in turn prolongs the replacement interval and reduces the cost.

5.5.4. Sensitivity Analysis

We next investigate the tradeoffs in CVSS regarding the block retirement threshold, the strength of ECC engine, and the impact of different GC formula weights.

Block Retirement Threshold

The mapping-out behavior for aged blocks in CV-SSD is controlled by a user-defined threshold. By default, blocks are mapped out and turn to a retired state once their RBER exceeds 5%. In this section, we investigate how this threshold affects the performance and device lifetime.



(a) Zipfian (30% utilization)

(b) Zipfian (70% utilization)

(c) Random (30% utilization)

(d) Random (70% utilization)

Figure 42: Terabytes written (TBW) with different performance requirements. Compared to TrSS and AutoStream, CVSS significantly extends the lifetime while meeting performance requirements.

We utilize YCSB-A and YCSB-F with their data set configured to have thirty million key-value pairs (10 fields, 100 bytes each, plus key). We compare three different configurations: (1) TrSS, vanilla F2FS plus a fixed-capacity SSD; (2) CVSS(4%), CVSS with a higher reliability requirement. Superblocks will be mapped out if RBER is greater than 4%; (3) CVSS(6%), CVSS with a lower reliability requirement. Superblocks will be mapped out if RBER is greater than 6%.

Figure 44 shows the latencies at major percentile values (p75 to p99) and the device lifetimes for each workload. As shown in Figures 44a and 44b, CVSS(4%) reduces p99 latency by 51% for the YCSB-A workload and by 53% for the YCSB-F workload compared

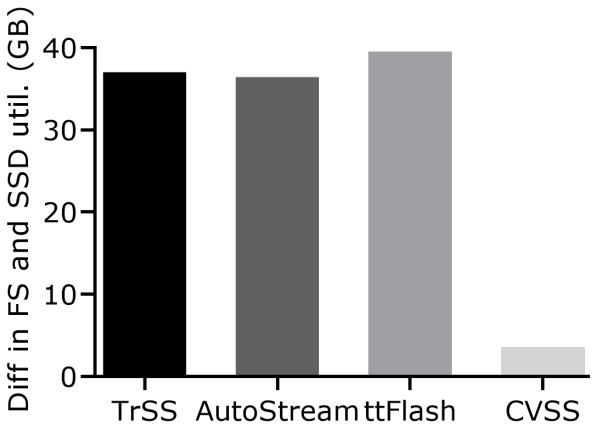
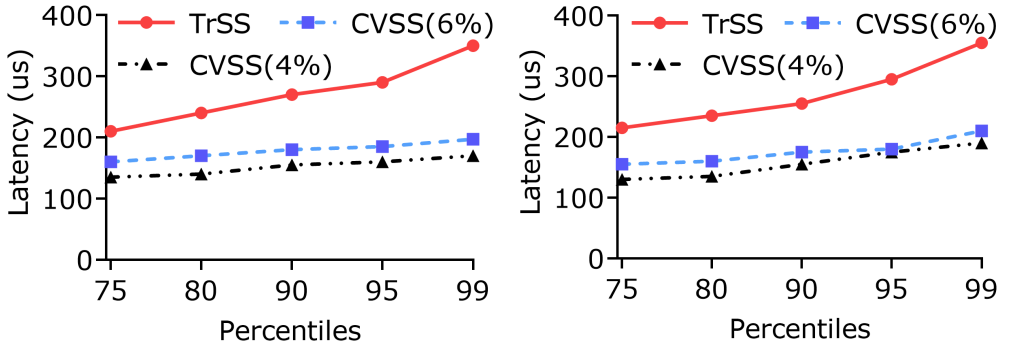
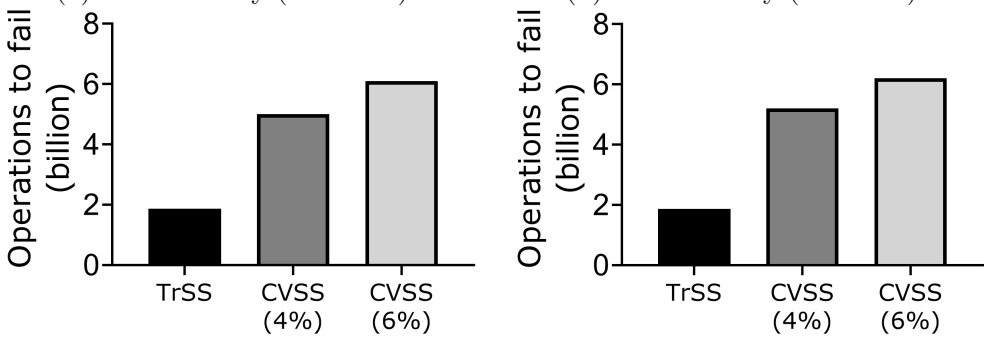


Figure 43: The average difference in FS and SSD utilization under Twitter traces. The original discard policy shows higher utilization inconsistency between FS and SSD, making data aliveness misjudged.



(a) Read latency (YCSB-A)

(b) Read latency (YCSB-F)



(c) Lifetime (YCSB-A)

(d) Lifetime (YCSB-F)

Figure 44: Sensitivity analysis on the mapping-out threshold in CVSS. CVSS with a higher reliability requirement, CVSS(4%), achieves better performance but with a relatively shorter lifetime compared to CVSS(6%) because blocks are retired earlier.

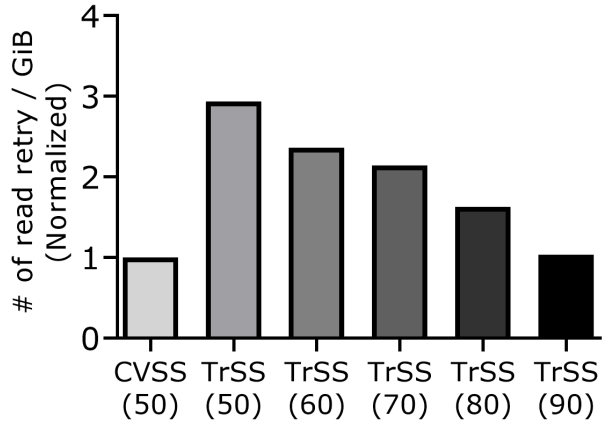


Figure 45: The average number of read retries triggered per GiB read over the device's lifetime. The x-axis represents different ECC strengths in bits.

to TrSS, which are 44% and 40% for CVSS(6%). With a higher reliability requirement, blocks are retired earlier in CVSS(4%), which in turn causes a relatively shorter device lifetime than CVSS(6%). As depicted in Figures 44c and 44d, CVSS(6%) and CVSS(4%) ingests 3.27× and 2.68× host I/O than TrSS on average, respectively.

ECC Strength

We now study the impact of ECC strength on SSD error handling and demonstrate the usefulness of capacity variance in simplifying SSD FTL design. As discussed earlier, CVSS excludes aged blocks from use and thus incurs fewer error correction operations. This further allows the CV-SSD to be equipped with a less robust error-handling mechanism without compromising reliability.

Figure 45 compares the average number of read retries triggered per GiB read over the device's lifetime for CVSS with ECC strength set as up to 50 bits corrected per 4KiB and TrSS with ECC strength set to 50 – 90 bits. The results are measured under the FIO Zipfian read/write workload with device utilization of 30%. We make two observations. First, with the same ECC capability, TrSS(50) performs 1.93× more read retry operations than CVSS(50). Second, TrSS requires a stronger ECC engine to improve the efficiency of the

error correction process, which complicates the FTL design in SSDs. On the other hand, with a weaker ECC engine, CVSS(50) achieves similar performance to TrSS(90).

GC Formula

As described in § 5.3.2, the GC formula consists of three parameters: $W_{invalidity}$, W_{aging}, and W_{read}. We analyze how different weights used in GC formula affect the performance of CVSS in this section. We compare the configured weights with three different configurations: (1) GC prioritizes blocks with more invalid pages, with $W_{invalidity} = 0.6$, $W_{read} = 0.2$, and $W_{aging} = 0.2$; (2) GC prioritizes blocks with more reads, with $W_{invalidity} = 0.2$, $W_{read} = 0.6$, and $W_{aging} = 0.2$; (3) GC prioritizes blocks with more erases, with $W_{invalidity} = 0.2$, $W_{read} = 0.2$, and $W_{aging} = 0.6$. FIO is used to generate Zipfian read/write workloads to the device. Figure 46 illustrates the measured WAF and read retry. Overall, the configured weights result in a lower WAF and fewer read retry operations.

In particular, compared to the configured case, the high $W_{invalidity}$ case achieves a lower WAF but involves 0.78× more read retry operations. This is because read-intensive data are stored in aged blocks. For the high W_{read} case, it triggers fewer read retry operations but decreases the cleaning efficiency of GC since the invalidity is not adequately considered during the victim selection. In the high W_{aging} case, GC always selects the most aged blocks, leading to a significant increase in WAF and faster device aging. In contrast, the configured weights balance WAF and read retry within the device.

5.6. Discussion

In this section, we discuss different use cases of capacity variance and its intersection with ZNS and RAID systems.

Use cases of capacity variance. CVSS aims to significantly outperform fixed-capacity systems in the best case, and perform at a similar level in the worst case. The degraded

mode serves the role of addressing the worst case by reserving a particular amount of capacity for the host. CVSS would be most useful for cases where IO performance is bottlenecked but has spare capacity. For instance, Haystack is the storage system specialized for new blobs (Binary Large Objects) and bottlenecks on IOPS but has spare capacity [131].

Moreover, for SSD vendors, capacity variance can simplify SSD design, as it allows for the tradeoff of performance and reliability with capacity. For data centers, introducing capacity variance can automatically exclude unreliable blocks and enable easy monitoring of device capacity, resulting in longer device replacement interval and mitigating SSD failure-related issues in data center environments. Lastly, for desktop users, capacity variance extends the lifetime of SSDs significantly and thus reduces the overall cost of storage.

ZNS-SSD. Capacity variance can be harmonious with ZNS. Specifically, due to a wear-out, a device may (1) choose to take a zone offline, or (2) report a new, smaller size for a zone after a reset. Both of these result in a shrinking capacity SSD. However, there is no software that can handle capacity reduction for ZNS-SSDs currently. The offline command simply makes a zone inaccessible and data relocation has to be done by users. Moreover, file systems are typically unaware of this change except for ZoneFS [89]. The capacity-variant SSD interface is a more streamlined solution where the software and the hardware cooperate to automate the process.

Capacity variance with RAID. The current CVSS does not support RAID systems. Existing RAID architectures require symmetrical capacity across devices and its overall capacity depends on the underlying minimal-capacity device. For parity RAID, the invalid data can not always be trimmed because it may be required to ensure the parity correctness. We will investigate the capacity-variant RAID system as our next direction, in which we consider modifying the disk layout and data placement scheme to support dynamically

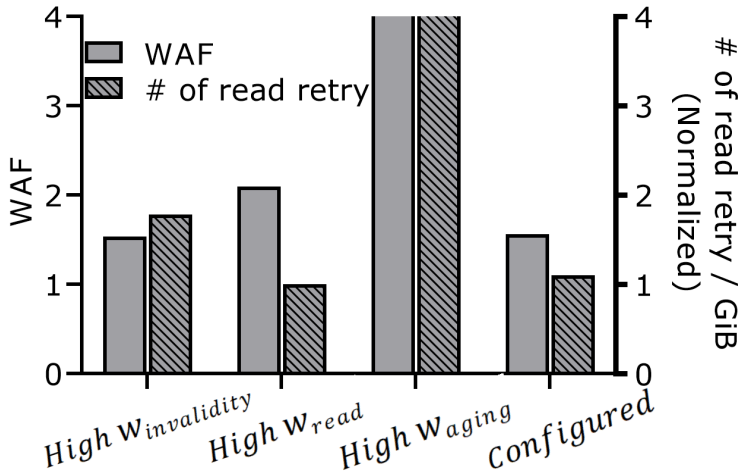


Figure 46: The WAF and read retries triggered under different weights used for GC formula.

changing asymmetrical capacity with multiple heterogeneous CV-SSDs.

5.7. Conclusion

The basic principle behind a capacity-variant storage system is simple: relax the fixed-capacity abstraction of the underlying storage device. We implement this idea and describe the key designs and implementation details of a capacity-variant storage system. Our evaluation result demonstrates how capacity variance leads to performance advantages and shows its effectiveness and usefulness in avoiding SSD fail-slow symptoms and extending device lifetime. We expect new optimizations and features will be continuously added to the capacity-variant storage system.

CHAPTER 6

paRAID: An Efficient and Sustainable Storage Architecture with Heterogeneous SSDs

6.1. Introduction

All-flash arrays (AFAs), leveraging the high performance, reliability, and compact form factor of solid-state drives (SSDs), have emerged as a promising solution to meet the growing storage demand in the big data era [64, 83]. Concurrently, data centers need to reduce carbon emissions, particularly for flash, which accounts for 40% of embodied carbon in servers [111]. However, decreasing carbon emissions from AFAs is difficult, especially since traditional AFA solutions assume homogeneous storage components and uniformly distribute IOs [64, 83, 121, 128, 164]. This architecture limits the opportunity to exploit the heterogeneity of modern SSDs [79] and the distinct data characteristics observed in real workloads [86], causing inefficiencies and exacerbating the carbon footprint of storage systems.

Specifically, modern SSDs exhibit significantly greater heterogeneity compared to traditional hard disk drives (HDDs), where variations between models are minimal [46, 67, 68, 103, 132]. As illustrated in Figure 47, advancements in flash density result in significant capacity variations across different models – even within the same product line. Consequently, upgrading a single disk in an AFA often requires replacing all disks and disposing of the old devices [71, 103, 170, 183], leading to increased embodied carbon emissions and cost. Moreover, the performance characteristics of modern SSDs vary significantly due to different factors such as the type of flash memory cells used, hardware configurations (e.g., number of channels/dies and block size), and firmware management (e.g., FTL algo-

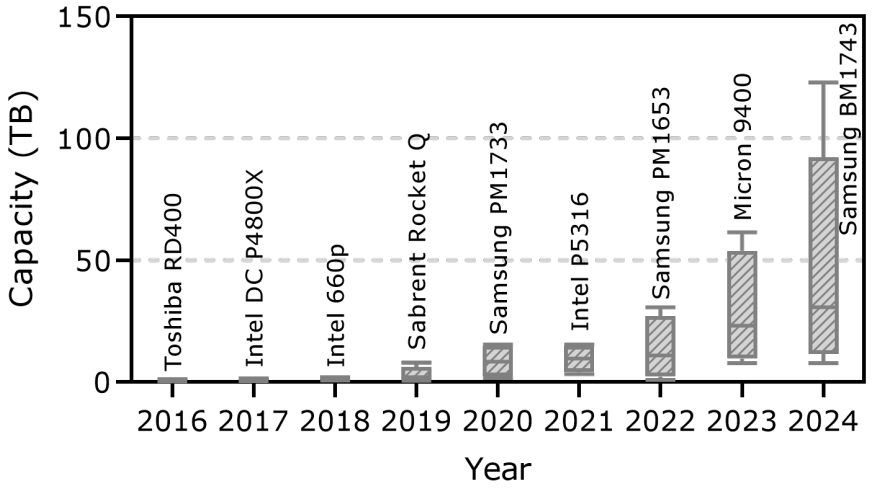


Figure 47: Trends in SSD capacity over the past decade. Each bar represents different capacity models for the same SSD. Variations in SSD capacity have doubled each year, and significant differences in device capacities can be observed for recent SSD products. These factors contribute to the growing asymmetry in storage device capacities.

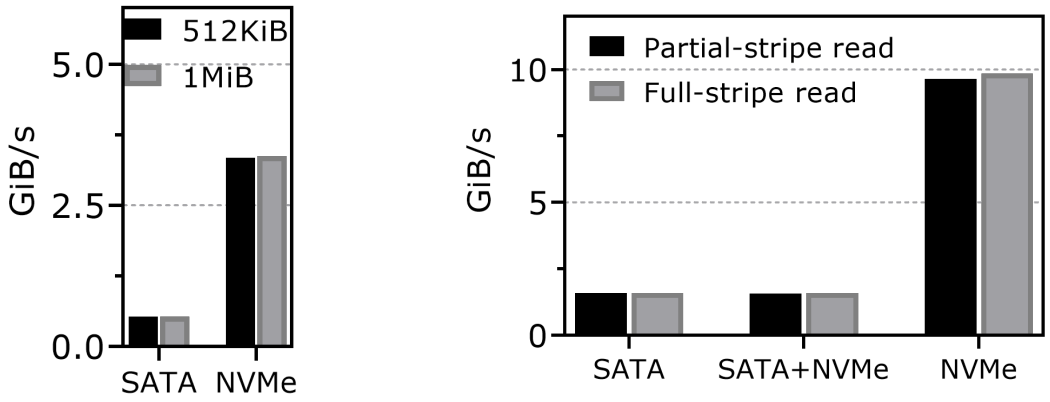
rithms).

On the other hand, the overall architecture of existing AFA systems is built around the assumption that the underlying storage components are homogeneous. A common N-disk RAID maintains a deterministic stripes-to-disk mapping that uniformly distributes data and parity chunks across all disks [179]. This architecture results in significant disk underutilization when considering heterogeneity among storage devices, particularly for modern SSDs. The balanced data layout requires the underlying storage devices to be roughly the same size. Otherwise, the aggregate capacity is determined by the minimal capacity device [121], making devices with larger capacity underutilized. This issue becomes worse with the increasing scale of AFAs and SSD capacities. Specifically, emerging AFAs, such as NetApp AFF [122] and EMC VMAX [33], contain 90 or more SSDs. For the latest SSD shown in Figure 47, capacity utilization drops below 50% when used with previous SSD products. Such underutilized capacity further leads to an additional 1843.2 kg em-

bodied CO_2 emissions and 491.52 kg annualized operational emissions, with its QLC flash cells [48, 111].

Moreover, by evenly spreading data across the disks, the overall system performance is bottlenecked by the poor-performing drives. We illustrate this by testing the popular Linux-MD software RAID5 array built on heterogeneous SSDs. Figure 48a shows the performance profile of two types of SSD products used for the experiments, measured by fio [61]. The read throughput of the NVMe SSD outperforms the SATA SSD by 5.31× and 5.35× with 512 KiB and 1 MiB request sizes, respectively. We then compare three RAID5 systems: (1) SATA: consisting of 3 SATA SSDs; (2) SATA+NVMe: consisting of 1 SATA SSD + 2 NVMe SSDs; and (3) NVMe: consisting of 3 NVMe SSDs. As shown in Figure 48b, although utilizing two NVMe SSDs, the performance of the SATA+NVMe RAID remains similar to the SATA RAID and achieves only 16.2% and 16.1% of the NVMe RAID performance, for partial-stripe read and full-stripe read, respectively. In this configuration, the spare device bandwidth leads to approximately 5.04 kg CO_2 per terabyte of device capacity annually [111]. As a result, with the conventional RAID solution, the performance is determined by the lowest-performance device, resulting in significant under-utilization and poor sustainability of storage resources.

As the latest Linux MD is capable of supporting arrays with 384 component devices [100], large cluster storage systems almost always include a heterogeneous mix of storage devices [107, 170]. Additionally, real-world data often exhibit varying lifetimes and temperatures, which are not uniformly accessed and affect overall system performance differently. For instance, data services at Google involve hot/warm/cold/frigid IO accesses [86], and modern filesystems internally categorize data into different types (e.g., metadata, three data logs, and three node logs for F2FS [90]), each with a different temperature. Consequently, such heterogeneous environments inevitably lead to utilization issues and poor



(a) Performance comparison of SATA and NVMe SSDs.

(b) Performance comparison of three RAID5 systems. SATA (3 SATA SSDs) vs. SATA+NVMe (1 SATA SSD and 2 NVMe SSDs) vs. NVMe (3 NVMe SSDs).

Figure 48: Performance bottleneck caused by disk heterogeneity with Linux MD. Figure 48a shows the read performance of a SATA and an NVMe SSD. Figure 48b demonstrates that the overall RAID performance is determined by the device with the lowest performance with conventional RAID solutions.

sustainability with existing RAID architectures that uniformly use underlying storage components. Moreover, while ongoing works [141] leverage the latest data placement technologies, enabled by NVMe Flexible Data Placement (FDP) interface, to enhance sustainability, these solutions are incompatible with conventional devices.

In response to these challenges, this paper presents paRAID (placement asymmetric RAID), a new RAID construction mechanism that exploits disk and data heterogeneity to improve performance and sustainability for modern AFAs. paRAID is designed to asymmetrically distribute data across the array to fully utilize the capacity of each SSD, and mathematically guarantees a maximum logical volume exported to the host. To prevent performance bottlenecks, paRAID maintains multiple stripe groups based on their performance characteristics and differentially exports the address space of each data stripe to the host, allowing for a performance-aware logical volume. To reduce carbon emissions and overhead from metadata, paRAID learns the mapping information between user ad-

dress space and device address spaces, which requires a minimum amount of DRAM and storage space for logical-to-physical (L2P) and physical-to-logical (P2L) address translation. The learned addressing approach further enables a more performance-optimized data placement, which adaptively tunes the data layout based on workloads, leading to better utilization of higher-performance devices.

The contributions of this paper are as follows.

- We study and identify the inefficiency of conventional RAID solutions when considering disk heterogeneity for modern SSDs through real system deployment. (§ 6.3)
- We present paRAID, to our knowledge, the first RAID architecture designed to leverage a mix of heterogeneous SSDs to improve overall system performance and sustainability by distributing data asymmetrically. The artifacts of paRAID will be open-sourced. (§ 6.4)
- We evaluate and quantitatively demonstrate the effectiveness of paRAID in managing heterogeneous SSDs, by comparing it against different state-of-the-art solutions under recent real-world I/O workloads. (§ 6.5)

6.2. Background

In this section, we first introduce the background for RAID and discuss the heterogeneous nature of modern SSDs. We then present the trend of carbon emissions for flash-based storage. Finally, we briefly review existing approaches for managing and optimizing AFA storage.

6.2.1. RAID

Redundant array of independent disks (RAID) is a classic solution that manages an array of SSDs to improve performance, reliability, and capacity simultaneously [164]. In RAID,

data is striped across the drives based on the required level of redundancy and performance, referred to as RAID levels. The RAID driver maintains metadata that converts logical block addresses (LBAs) from the host into physical block addresses (PBAs) on the actual disks in the array.

RAID-5 and RAID-6 are the most commonly used parity-based RAID levels to balance throughput, redundancy, and space overhead [147]. For an N-drive RAID-5 array, each data stripe consists of $N-1$ data chunks and one parity chunk, which rotates for each stripe to eliminate any parity bottlenecks in the array [9]. As a result, RAID-5 offers redundancy against one drive failure with $1/(N-1)$ space overhead. In theory, RAID-5 can deliver read throughput that scales up to N times and write throughput that scales up to $N-1$ times compared to the throughput of a single device [147].

However, in practice, this throughput can hardly be achieved even without drive failures [64, 96, 179]. There are three write modes in parity-based RAID: full-stripe write, read-modify-write, and reconstruct write. For a partial-stripe write (read-modify-write and reconstruct write) where only a subset of the data chunks of a stripe is written, parity chunks or old data chunks need to be read from corresponding drives to generate new parity chunks, thus amplifying the amount of I/Os. In terms of read, achieving ideal performance relies on the assumption that the underlying drives can simultaneously deliver consistent performance. However, this assumption does not hold true in practice, particularly with modern SSDs. We will discuss this impact in the subsequent section.

6.2.2. SSD Heterogeneity

NAND flash memory density continues to scale to meet the growing storage demands of data-intensive applications. Over the past decade, the density of planar flash increases by more than 1000×, driven by advancements in both manufacturing technology and cell density [68, 106]. With that, SSD capacity has been doubling annually, and recent models

such as Nimbus ExaDrive DC [123] and Samsung BM1743 [143] offer capacities of 100 TiB or more.

In addition to capacity, SSDs inherently exhibit performance variability from the time of manufacturing due to differences in the NAND flash memory cells and variations in the firmware [24, 144]. Modern SSDs composed of persistent memory (PM), triple-level cell (TLC), or quadruple-level cell (QLC) flash memory present very different performance characteristics. A recent study has shown that the performance differential between two commodity SSD devices can be as high as 5.1× [164]. Moreover, even using the same type of flash memory cells, the overall performance of an SSD varies depending on the device configuration (e.g., number of channels/dies, block size, FTL algorithms). Recent SSDs have shown performance variations of up to 22% among devices within the same product line [113].

A recent study has shown that the performance differential between two commodity SSD devices can be as high as 5.1× [164]. Thus, to enhance overall resource utilization, it is often recommended to construct RAID systems using disks of the same make and model [71, 99]. Nevertheless, the issue of heterogeneous performance persists even among disks of identical models. Prior research has shown that SSDs can suffer from a permanent performance degradation caused by flash memory errors [46, 67, 132].

6.2.3. Carbon Footprint of Flash

Data centers are projected to emit more than 33% global emissions by 2050 [111] and flash-based storage is responsible for 33-80% of a computer's carbon footprint [186]. The carbon footprint of an SSD consists of embodied emissions from hardware manufacturing and infrastructure activities [111] and operational emissions from device usage. As flash annual capacity production exceeds 765 Exabytes, the associated embodied emissions amount to 122M metric tonnes of CO_2 [186], equivalent to the average annual CO_2 emis-

Table 6: Key features of existing AFA systems.

	Data layout	Issue tackled	Disk hetero
mdraid [121]	RAID	—	✗
FusionRAID [64]	MOLS-based	I/O determinism	✗
StRAID [164]	RAID	I/O concurrency	✗
LogRAID [128]	Log-structured	Small writes	✗
SWAN [83]	2D Array	GC overhead	✗
Tiger [71]	Pool	System reliability	✗
paRAID	Adaptive	Storage utilization	✓

sions of 28M people [155] . By 2030, this figure is expected to rise to the equivalent of over 150M people [38]. Such an exponential growth must require the development of more sustainable designs.

To reduce carbon emissions, companies including Meta [112] and Microsoft [114] are adopting renewable energy sources such as solar and wind [49, 111]. Google [43] and AWS [4] aim to complete their transition to renewable energy by 2030. While these strategies are effective at reducing operational emissions, they do not alleviate the embodied emissions of data centers. In particular to flash-based storage, improving overall utilization and reducing DRAM consumption, whose embodied carbon per bit is 12× higher, have been identified as key measures to amortize and mitigate flash storage's carbon footprint [48, 111, 186].

6.2.4. Related Works

As summarized in Table 6, AFAs have been extensively studied, with approaches primarily categorized into three groups: (1) enhancing performance by reducing software overhead [64, 164, 179]; (2) taming tail-latency by alleviating GC and small writes overhead [83, 96, 128]; (3) improving reliability by distributing parity unevenly across the devices [9] or adopting disk-adaptive data redundancy scheme [71, 72, 74]. Approaches in the first two groups typically assume that disk components have the same capacity and comparable

performance, and most in the third group only focus on the reliability perspective of the system.

Reducing software overhead. Both RAID+ [179] and FusionRAID [64] utilize the mathematical properties of mutually orthogonal Latin squares (MOLS) [179] to spread I/O to a larger disk pool in a balanced manner. Unfortunately, MOLS requires each device to have exactly the same size [109]. FusionRAID relies on latency spike detection and requests redirection to avoid drives experiencing degraded performance, causing additional system overhead. Moreover, to reduce mapping overhead, it restricts a relatively large block size. On the other hand, StRAID [164] addresses the lock contentions in Linux-MD [121] by assigning a dedicated thread for each stripe write. It also adopts a two-phase write mechanism to opportunistically aggregate I/Os.

Maintaining performance consistency. LogRAID [128] employs a log-structured design to optimize the performance of small partial stripe writes. SWAN [83] organizes SSDs into foreground and background groups. The foreground group is used to accommodate host writes and only SSDs in background groups are allowed to perform garbage collection (GC). It aims to alleviate GC interference without considering disk heterogeneity. IODA [96], on the other hand, proposes a busy time model to coordinate SSD internal tasks and achieve I/O determinism. The model considers devices to have the same capacity and performance.

Improving system reliability. Diff-RAID [9] distributes parity blocks unevenly across the array and maintains an age differential to lower correlated failures in SSDs. However, it concerns the case where the array has operated in the degraded mode — data can not be reconstructed because of correlated failures once a drive fails.

Several works [71, 72, 74] propose disk-adaptive redundancy schemes to ensure data reli-

ability. For example, Tiger [71] tailors the redundancy scheme of each data stripe based on disks' annualized failure rates (AFRs). These approaches contain disks from only one make/model. Works within this group mainly focus on addressing disk heterogeneity in terms of reliability rather than capacity and performance.

6.3. Experimental Study on RAID with Heterogeneous SSDs

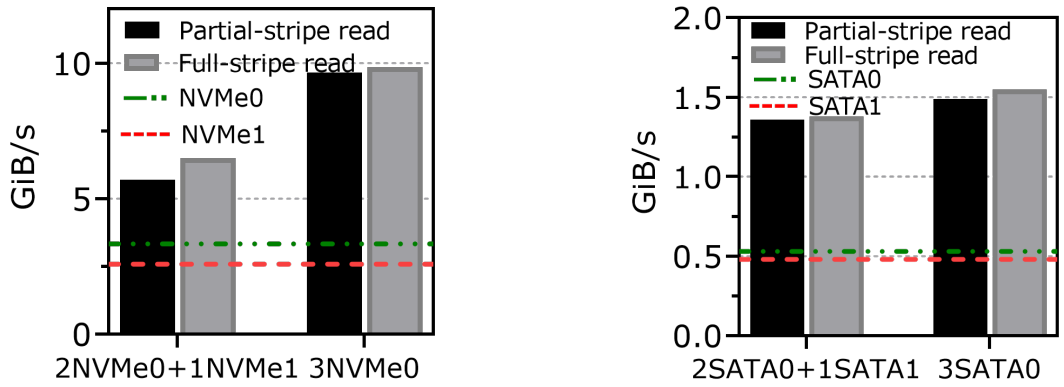
As introduced earlier, conventional RAID solutions assume the underlying storage components are homogeneous and distribute I/O requests from applications evenly across the array. In this section, we begin with an experimental study to evaluate the performance of RAID with heterogeneous SSDs and demonstrate that the existing RAID architecture fails to effectively leverage the heterogeneity inherent in modern SSDs. We then analyze the effect of under-utilization caused by disk heterogeneity on the system's overall sustainability.

6.3.1. Performance Analysis

Experiments setup. We focus on sequential read workloads to mitigate the impact of SSD background operations (e.g., garbage collection) and analyze the performance of conventional RAID architectures with heterogeneous SSDs. We configure a set of (2+1) RAID5 (i.e., 2 data chunks and 1 parity chunk) arrays with various types of SSD products (9 off-the-shelf SSDs in total) using the default Linux mdadm [121] software RAID controller. Table 7 summarizes the characteristics and the usages of the experimented SSDs. To show

Table 7: Performance characteristics and usage of devices used in the experimental study. *Degraded* refers to that the device remains operational but experiences reduced performance.

Device (# used)	4K/64K read BW (GiB/s)	Disk condition	Usage
NVMe0 (3)	0.73 / 3.33	Normal	Homogeneous NVMe RAID
SATA0 (3)	0.44 / 0.53	Normal	Homogeneous SATA RAID
NVMe1 (1)	0.73 / 2.58	Normal	Heterogeneous NVMe RAID
SATA1 (1)	0.33 / 0.48	Normal	Heterogeneous SATA RAID
NVMe0-degraded (1)	0.65 / 1.34	Degraded	RAID with degraded device



(a) NVMe-only SSD RAID

(b) SATA-only SSD RAID

Figure 49: In Figure 49a, using one NVMe1 SSD results in a 41% degradation in overall performance with the conventional RAID architecture. In Figure 49b, with only 0.05 GiB/s bandwidth difference between SATA0 and SATA1 SSD, it leads to an 11% reduction in the overall RAID performance.

the inefficiency of conventional RAID architectures in managing heterogeneous SSDs, we consider the following three different configurations.

Case #1: RAID with a mix of NVMe and SATA SSDs. As discussed in Section § 6.1 (Figure 48b), the conventional RAID architecture fails to deliver performance improvements even when two NVMe SSDs are utilized. In particular, for full-stripe read workloads, the total RAID bandwidth reaches only 1.48 GiB/s, whereas the aggregated bandwidth of the devices is 7.19 GiB/s, resulting in an NVMe device utilization rate of less than 17%.

Case #2: RAID with NVMe-only SSDs or SATA-only SSDs. We next investigate the scenario where various SSDs are used, but all belonging to the same type (i.e., either NVMe or SATA). This experiment eliminates the performance discrepancies arising from different storage interfaces.

Figure 49a compares the performance of the heterogeneous NVMe RAID (2 NVMe0 SSDs + 1 NVMe1 SSD) against the homogeneous NVMe RAID (3 NVMe0 SSDs). The lines il-

lustrate the read performance of individual devices in their steady state. Therefore, the gap between the two lines indicates the difference in aggregated device bandwidth between the heterogeneous and homogeneous RAID configurations.

Notably, the total device bandwidth of the heterogeneous NVMe-SSD RAID reaches up to 9.24 GiB/s, while the configured RAID only achieves 6.5 GiB/s under full-stripe reads and 5.7 GiB/s under partial-stripe reads. The bandwidth utilization is only 57% for a single NVMe0 SSD and 61% for the overall RAID under partial-stripe read workloads. Additionally, using one NVMe1 SSD results in a 41% degradation in overall performance with the conventional RAID architecture.

As shown in Figure 49b, the performance degradation caused by device heterogeneity also occurs in the SATA-SSD RAID, but to a smaller degree than in the NVMe-SSD RAID. This is primarily due to the relatively small performance difference between SATA0 SSD and SATA1 SSD, which is only 0.05 GiB/s. However, it still leads to an 11% reduction in the overall RAID performance. Compared with the NVMe-SSD RAID, it indicates that the impact of performance heterogeneity in RAID will become increasingly evident as SSD technology continues to advance.

Case #3: RAID with SSDs of the same model. The previous cases demonstrate the inefficiency of traditional RAID solutions when different SSD products are deployed. In this experiment, we show that the challenge of heterogeneous performance persists even when disks of identical models are used, due to their different disk conditions. Specifically, we configure a RAID5 array using three NVMe0 products. Among these, two SSDs are relatively new, with no more than 5% percentage used (in terms of endurance) and without any errors logged according to their S.M.A.R.T. [124] attributes, while one SSD has 63 device errors logged. This scenario can happen in large-scale storage clusters, where same model SSDs are deployed, but some exhibit anomalous performance [46, 68, 107, 170].

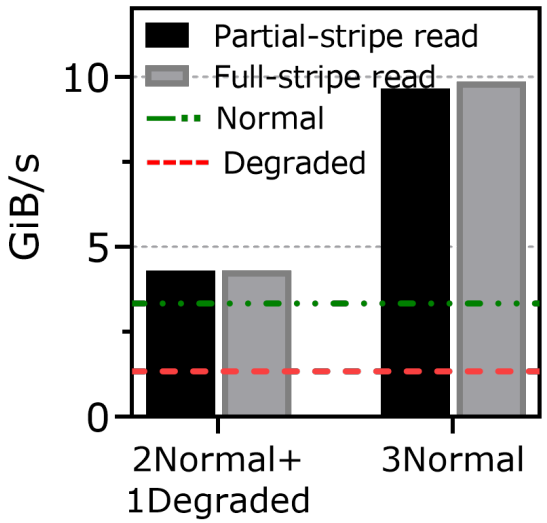


Figure 50: The challenge of performance heterogeneity persists even when disks of identical models are used, due to different disk conditions.

As illustrated in Figure 50, the RAID array consisting of identical SSD models, with one experiencing degraded performance (NVMe0-degraded), performs 58% worse than the RAID array with all normal SSDs. The overall throughput is bottlenecked by the NVMe0-degraded SSD, which achieves only 40% of the read performance compared to the normal NVMe0 SSDs. This problem is not specific to Linux software RAID: our measurement also shows a hardware controller (Broadcom MegaRAID SAS) producing very similar results. Therefore, using SSDs of the same model cannot eliminate performance heterogeneity in RAID. As devices wear out and are replaced, it can lead to aging differentials and performance heterogeneity among devices.

6.3.2. Sustainability Cost from Underutilization

Operational emissions

We examine the sustainability efficiency of previous RAID configurations. Specifically, we use the model developed in recent work [111] to estimate the operational carbon emissions attributed to the unutilized device bandwidth. The model parameters are based on specifi-

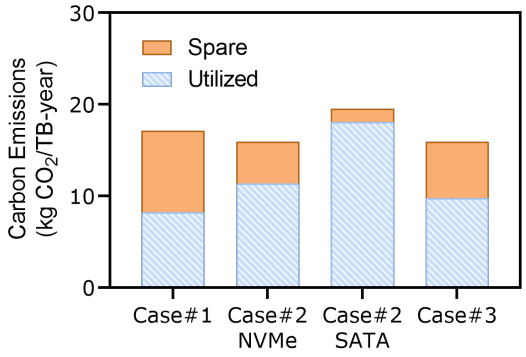


Figure 51: Operational emissions generated by the unutilized device bandwidth.

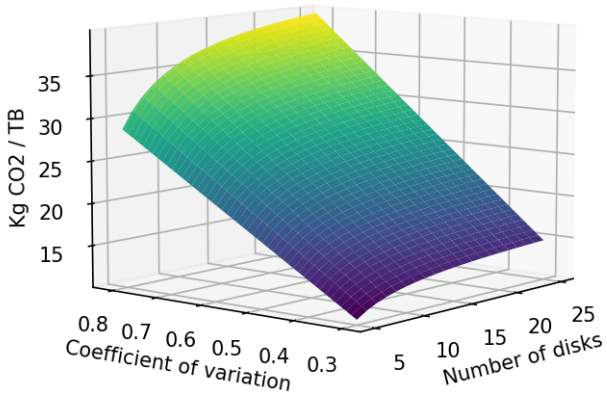


Figure 52: Embodied emissions generated by the unutilized device capacity.

cations from manufacturers.

Figure 51 presents the results. For case #1, the unused RAID bandwidth (denoted as spare) generates 8.904 kg CO_2 per terabyte of capacity annually, which accounts for 53% of the total operational emissions. The SATA RAID in case #2 achieves better sustainability (1.43 kg CO_2 /TB-year) because of lower performance discrepancies. For case #3, although the RAID is built on SSDs of the same model, it still leads to poor sustainability with extra 6.148 kg CO_2 /TB-year. As a result, conventional RAID architecture demonstrates limited effectiveness in managing heterogeneous SSDs, which in turn results in low storage utilization and poor sustainability.

Embodied emissions

We next analyze the embodied carbon emissions attributed to the unutilized device capacity with heterogeneous SSDs. In a RAID system, the capacity utilization (CU) can be calculated using the formula:

$$CU = \frac{Minimum\ disk\ capacity \times \#\ of\ disks}{Aggregate\ disk\ capacity}$$

For simplification, we assume that the disk capacities are independently and identically distributed (i.i.d) random variables following a uniform distribution. Therefore, the expected value of the minimum disk capacity ($E[Min]$) for a RAID built with N disks uniformly distributed over [a, b] is:

$$E[Min] = a + \frac{b-a}{N+1}$$

Particularly, a and b can be expressed in terms of mean (μ) and standard deviation (σ) of disk capacities:

$$a = \mu - \sqrt{3}\sigma \quad \text{and} \quad b = \mu + \sqrt{3}\sigma$$

By substituting back into $E[Min]$, we get:

$$E[\text{Min}] = \mu - \sqrt{3}\sigma + \frac{2\sqrt{3}\sigma}{N+1} = \mu - \sqrt{3}\sigma\left(\frac{N-1}{N+1}\right)$$

Thus, the expected unutilized capacity for a given RAID can be derived as:

$$1 - Expected\ CU = 1 - \frac{E[\text{Min}] \times N}{N\mu} = \sqrt{3}\left(\frac{\sigma}{\mu}\right)\left(\frac{N-1}{N+1}\right)$$

where $\frac{\sigma}{\mu}$ represents the relative capacity variation.

Figure 52 plots the embodied carbon emissions resulting from unutilized device capacity for each terabyte of aggregated RAID capacity, given the number of disks (N) and coefficient of variation ($\frac{\sigma}{\mu}$). The emission value is calculated based on a 30nm NAND chip [48]. Overall, greater capacity variation and a higher number of disks result in more wasted embodied emissions, indicating that increased heterogeneity among storage devices exacerbates the inefficiency of conventional RAID systems.

6.4. Design of paRAID

Inspired by the observations and analysis above, we introduce paRAID (placement asymmetric RAID). In this section, we first present the overview of the proposed design and outline the key challenges (§ 6.4.1). We then describe the methodology for building the heterogeneity-aware data layout (§ 6.4.2) and the performance-aware logical volume (§ 6.4.3). Lastly, we explore the learning-based addressing approach (§ 6.4.4) and its enabled adaptive data placement scheme (§ 6.4.5).

6.4.1. Overview

The proposed paRAID is designed to be heterogeneity-aware for both disks and workload characteristics. The core idea is to asymmetrically distribute data across a larger disk pool based on data temperature, as well as each device's capacity and performance. As a result, paRAID improves storage performance and sustainability by adaptively using storage resources because (1) more data will be allocated to larger SSDs and less to smaller ones, and (2) hot data will be placed on fast SSDs while cold data will be stored on regular SSDs.

Figure 53 illustrates the overall architecture of the paRAID, for a (2+1) RAID5 configuration from an N-disk array. Given the disk pool and RAID configuration, paRAID builds a three-dimensional logical address space, an internal logical block layer between the user-perceived logical and the SSD logical address spaces, where the depth corresponds to the capacity of each individual device (Steps ❶ – ❹). paRAID then separates each address space into one or more stripe groups and differentially exports them into a one-dimensional logical address space to the host based on the device characteristics (Steps ❺ – ❻). Lastly, paRAID learns and adaptively tunes the mapping between host and device address spaces so that the performance-sensitive data are placed in fast SSDs and cold data are placed in

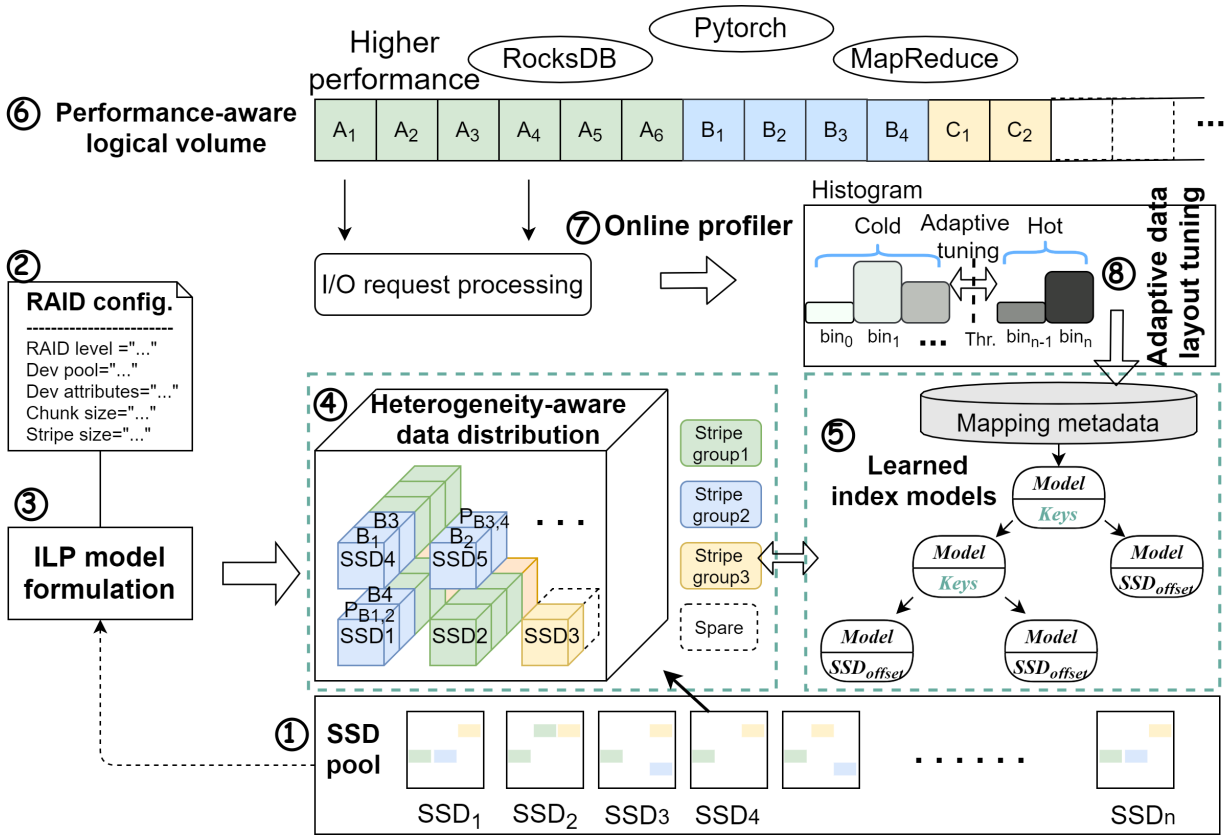


Figure 53: Overview of paRAID: a (2+1) RAID5 configuration. Given the disk pool and RAID configuration, paRAID constructs a mathematical model to optimize data organization and maximize the address space (Steps 1–4). It differentially exports each data stripe's address space with learned logical-to-physical mapping (Steps 5–6). Finally, paRAID adaptively tunes the addressing models based on access patterns to optimize the usage of higher-performance devices (Steps 7–8).

slow SSDs for improved performance and disk utilization (Steps ❼ – ❽).

The novel challenges here are to (1) derive an efficient data organization to utilize the total aggregate capacity given a disk array where each device has a different capacity; (2) achieve address translation between user space and AFA space with minimal overhead; as well as (3) design a lightweight mechanism that allows the host to optimize the usage of devices with higher performance based on workloads adaptively. We first explain the data distribution model that stripes data unevenly across devices and the initial logical volume layout. We then describe the learning-based addressing approach. Finally, we present the

performance-optimized data placement scheme enabled by paRAID.

6.4.2. Heterogeneity-aware Data Distribution

For conventional N-disk RAID arrays, each data stripe consists of exactly N chunks, which are uniformly distributed across N disks. To distribute data across a disk pool that exceeds the typical size of RAID arrays, one straightforward approach is to physically partition the disk pool into two RAID groups [179], where each disk belongs to one RAID array. RAID-50, for instance, adheres to this approach. However, while this approach achieves good disk isolation and efficient stripes-to-disks mapping, only a fixed amount of capacity is used for each device.

Alternatively, paRAID unevenly distributes data across the SSDs, with the goal of fully utilizing the capacity of each SSD and maximizing the available logical capacity exported to the host. Consequently, SSDs with larger capacity will be assigned more data and parity chunks within the array.

In particular, this can be formulated as an optimization problem as follows: given disk pool size N, size of the i^{th} disk S_i (where i ranges from 1 to N), data stripe width k ($k < N$), and chunk size C. Let x_{ijk} be a binary decision variable representing whether chunk k of data stripe j is assigned to disk i. The objective function is to maximize the number of complete k-width data stripes, denoted by D. There are three constraints inherited from the RAID to ensure data reliability: (1) each chunk of each data stripe must be assigned to exactly one disk; (2) any two chunks within a data stripe are placed on different disks; and (3) each disk can only accommodate a certain number of chunks based on its

size. Thus, we have:

$$\begin{aligned} &\text{Maximize} \quad D \\ &\sum_{i=1}^{N} x_{ijk} = 1, \quad \forall j, \forall k \\ &x_{ijk} + x_{ijk'} \leq 1, \quad \forall i, \forall j, \forall k' \neq k \\ &\sum_{j=1}^{D} \sum_{k=1}^{K} x_{ijk} \cdot C \leq S_i, \quad \forall i \end{aligned} \tag{6.1}$$

Improve the tractability: The analytical model above outlines a preliminary representation by embedding the objective function within the constraints. To make it more tractable

Table 8: Summary of parameters used in the data organization model.

Notation	**Description**
N	Disk pool size
S_i	Size of the i^{th} disk S_i $(1 \leq i \leq N)$
k	Data stripe width k $(k < N)$
C	Data chunk size
D_{max}	The theoretical upper bound of the number of complete stripes of the given disk array
x_{ijk}	Binary decision variable indicating whether chunk k of stripe j is placed in device i
z_j	Binary decision variable indicating whether stripe j is placed $(1 \leq j \leq D_{max})$
D	Objective function, the derived number of complete stripes $(D = \sum_{j=1}^{D_{max}} z_j)$

in practice, we introduce two additional variables: D_{max}, which represents the theoretical upper bound of the number of complete stripes of the given disk array, and z_j, which is another binary decision variable indicating whether stripe j is placed (where j ranges from 1 to D_{max}). Table 8 summarizes the parameters used in the model.

Therefore, by linking x_{ijk} with z_j, we can decouple the objective function from the constraints, and the formulations can be refined as follows:

$$\begin{aligned}
\text{Maximize } D &= \sum_{j=1}^{D_{max}} z_j \\
\sum_{i=1}^{N} x_{ijk} &= z_j, \quad \forall j, \forall k \\
x_{ijk} + x_{ijk'} &\leq z_j, \quad \forall i, \forall j, \forall k \neq k' \\
\sum_{j=1}^{D_{\max}} \sum_{k=1}^{K} x_{ijk} \cdot C &\leq S_i, \quad \forall i \\
x_{ijk} &\leq z_j, \quad \forall i, \forall j, \forall k
\end{aligned} \tag{6.2}$$

At this point, this combinatorial optimization problem can be solved using standard optimization techniques, for example, by taking advantage of integer linear programming (ILP) techniques [138]. Specifically, the value of D_{max} can be set as the total aggregate capacity of the disk array divided by stripe size (i.e., $\left\lfloor \frac{\sum S}{k \cdot C} \right\rfloor$). Figure 53 presents a simple example of the derived stripe organization, assuming $N = 5$, $k = 3$, $S = \{6, 4, 2, 5, 2\}$, and $C = 1$. The derived stripe organization mathematically guarantees the maximum logical address space given a disk array while allowing paRAID to asymmetrically place data to fully utilize the aggregate capacity.

6.4.3. Performance-aware Logical Volume

The formulated analytical model provides an efficient data organization to utilize the total aggregate capacity from a heterogeneous disk pool. However, there is another issue with how to integrate the address space of each stripe into a single logical volume and export it to the upper layers (e.g., file systems and host). Simply concatenating them will not yield performance benefits, as logical blocks are equally treated and accessed from the perspective of the file system.

To tackle this, paRAID profiles the performance characteristics of each stripe group, which involves fixed disk components derived from § 6.4.2. Based on the identified performance, the disk components within stripe groups are adjusted if the performance of the intra-group disks can be better aligned. paRAID then concatenates and exports the address space of each stripe group, starting from the higher-performing stripes and continuing to the lower-performing ones. Consequently, the basic layout of the logical volume is as follows: the LBA0 perceived by the user will be mapped to the most performant disks, while subsequent LBAs will be allocated to the progressively less performant disks. As illustrated in Figure 53, the green blocks correspond to the most performant stripe group, which consists of SSD 1, 2, and 4. The subsequent blue and yellow blocks are mapped to less performant disks.

With that, it opens the door for the system to differentially use logical blocks, leading to disks with higher performance being better utilized. As a result, performance-sensitive data (i.e., metadata and hot data chunks) can be placed in the lower LBA space for optimized performance, and cold data can be placed in the higher LBA space for better capacity utilization.

On the other hand, this basic layout requires the file system and applications to be aware

of such performance asymmetry across the address space. The need for alignment between logical blocks and their allocation policy limits the ability to integrate with existing systems. We address this challenge by proposing an adaptive data placement scheme in § 6.4.5, based on this basic layout.

6.4.4. Learned Model for Address Mapping

Given the derived data organization and logical volume layout, one intuitive approach to translating a user-space address to a disk address is to adopt a deterministic mapping table [64, 83, 179]. However, the chunk-level mapping table size is large, as device offsets need to be stored for each logical data chunk. Among the data structures in an RAID superblock, the address mapping table has the largest memory footprint [83, 153], which could take approximately 1% of the aggregated capacity for an AFA.

Moreover, the mapping table can significantly affect both sustainability and performance in RAID, as it not only determines the efficiency of L2P addressing, but also affects the utilization of host DRAM [153, 163], which generates 12× more embodied carbon emissions per bit [111]. As shown in Figure 54, a host LBA first needs to be translated to an AFA-level address and then it can be converted to an SSD offset. This challenge becomes even worse with the increasing flash memory capacity in an SSD, as larger address space usually requires a larger mapping table for addressing [153].

To improve the mapping efficiency and reduce memory footprint, we propose a learning-based addressing scheme. Unlike the conventional mapping table approach, the key idea is to learn the correlation between a set of logical data chunks and their corresponding device offsets, based on which it can build several space-efficient index segments. As presented in Figure 54, paRAID adopts the piecewise linear regression models to learn the host-to-device mappings for their simplicity and high computation efficiency compared to neural network-based learning approaches [154].

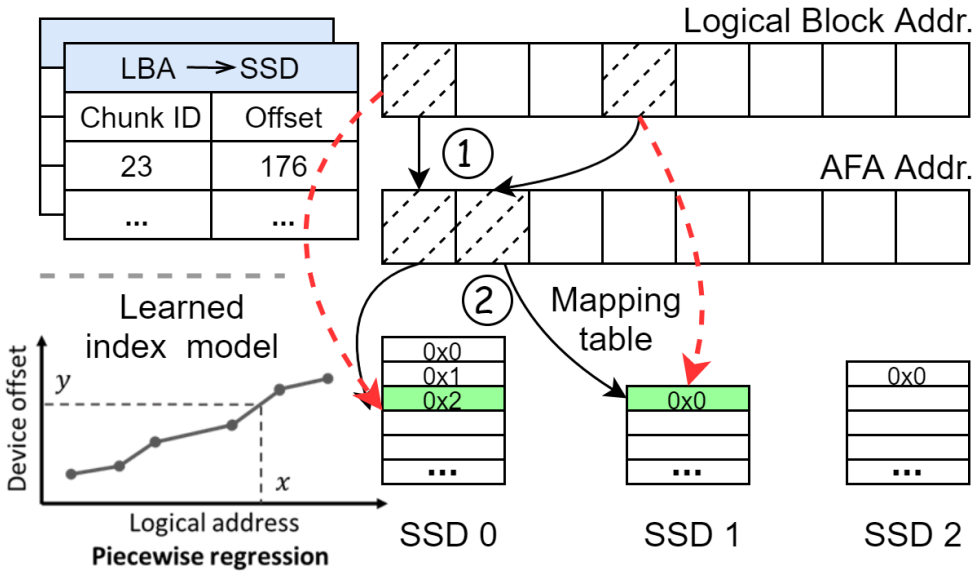


Figure 54: Mapping table-based v.s. learning-based approach for addressing in paRAID.

Since the learned index segment can be represented with (S, L, K, I) [153], where S denotes the start data chunk, L is the interval, K is the slope, and I is the intercept of the index segment, each segment will take 14 bytes: 4 bytes for S, 4 bytes for L, 2 bytes for K, and 4 bytes for I. Therefore, the disk offset of an LBA can be obtained with the expression: $offset_{SSD} = f(LBA) = [K * LBA + I], where\ LBA \in [S, S+L]$. Additionally, the LBA of a given disk offset can be derived from the expression: $LBA = f(offset_{SSD}) = [(offset_{SSD} - I)\ /\ K]$, which is useful to determine the stripe ID for parity computations. However, although this approach effectively reduces the mapping metadata overhead compared to the mapping table approach, it still requires a hashing mechanism to index the segments that have been learned from the data layout [164], which introduces additional overhead and the issue of hash collisions in practical use.

To further optimize the metadata overhead, paRAID builds a tree of index segments, structured into two types of learned nodes: model nodes (intermediate nodes) and data nodes (leaf nodes). The model nodes predict the index model in the subsequent level, and the

data nodes generate the corresponding physical device offset of the referenced logical data chunk. The number of levels and the number of models in each level are determined based on the cost model [35] during the RAID initialization process. By doing so, it improves the training and reference efficiency by narrowing down the search space across layers. Moreover, the data node can now be simply represented with (K, I) (slope and intercept) without the need for S (the start chunk ID) and L (the interval), as those are automatically handled by model nodes. This hierarchical approach reduces the space overhead for mapping metadata by ~57% (i.e., 6 bytes instead of 14 bytes for each data model).

When a user-level LBA is provided, paRAID calculates its logical chunk and stripe IDs. It then traverses the index tree, refining the location at each model node. Once the leaf node is reached, it provides the corresponding device offset for the given LBA and the IO request is sent to the underlying SSD.

6.4.5. Adaptive Data Placement

As previously mentioned, paRAID imbues performance information into logical blocks. By default, it maps stripes consisting of higher-performance devices to lower LBA spaces and stripes on lower-performance devices to higher LBA spaces. To manage such performance asymmetry across logical blocks, it is essential to dynamically adapt the data placement based on workload characteristics. Specifically, the hot data chunks identified in user space should be automatically mapped to devices with higher performance, while cold data chunks should be assigned to lower-performance devices.

Therefore, paRAID introduces an online workload profiler to monitor access patterns and feed this information to addressing models, which enables dynamical adjustments to the host-to-device mapping by updating index segments. In particular, paRAID profiles access patterns by sampling device I/Os at the granularity of the configured RAID chunk size, which is typically 64 KiB or larger [64, 71, 83, 96]. Similar to prior works [94, 139, 169],

the number of accesses to a chunk is accumulated using the exponential moving average (EMA). For example, assuming that the number of accesses to a chunk in the i^{th} profiling interval is x, and the EMA at the $(i-1)^{th}$ interval is EMA_{i-1}, then the updated EMA_i can be computed as follows:

$$EMA_i = d * x + (1 - d) * EMA_{i-1} \tag{6.3}$$

where d is a decay factor (0.5 by default [94]) to balance the contribution of previous and current profiling records.

The computed $EMAs$ are then used to build an access histogram, in which the n^{th} bin contains all distinct chunks with the EMA value ranging from 2^n to 2^{n+1}. The hot bin threshold, denoted as $Hot_{threshold}$, is decided based on the access histogram and the aggregated capacity of higher-performance SSDs. The accumulated size of hot chunks in the [bin_{hot}, bin_{max}] is just smaller than the fast device capacity.

Therefore, once $Hot_{threshold}$ is determined, data chunks that fall within [bin_{hot}, bin_{max}] but are not stored on fast SSDs are marked for promotion. Conversely, data chunks that are stored on fast SSDs but do not fall within [bin_{hot}, bin_{max}] are considered for demotion. paRAID assigns a separate thread to manage this process. To reduce overhead, paRAID prioritizes the hottest and coldest chunks in devices without outstanding I/Os and opportunistically migrates target chunks when they are modified by the host so that they can be updated for free. The addressing models are also updated to reflect the new locations for the corresponding chunks. With that, paRAID achieves better disk utilization by automatically redirecting frequently accessed data chunks to fast SSDs and cold chunks to slower SSDs without requiring any application-level changes.

6.5. Evaluation

In this section, we evaluate paRAID and compare it against other designs under real-world workloads. We first describe our experimental setup and then present the results.

6.5.1. Experimental Setup

Implementation and platform. paRAID requires no device level modifications and is implemented upon the Linux kernel v5.15 based on StRAID [164] as a user-space AFA engine, with 9667 lines of code changed. We use PuLP [137] to implement the data organization model and the solver is built based on the IBM CPLEX optimization library [59]. We run all experiments on a Dell PowerEdge T640 Server, with 32-core Intel Xeon Silver 4208 CPU, running Ubuntu 20.04 LTS. We utilize ten SSDs on two different RAID configurations: (1) 2+1 RAID5 with 6 devices and (2) 4+2 RAID6 with 10 devices (described in Table 9). The chunk size is set to 64 KiB by default.

Workloads. We use 21 different workloads for evaluation: (1) 7 production traces from Meta on different regions [167]. Each workload sampled production traffic from a Tec-

Table 9: System configuration and device characteristics.

RAID	Types	Read/Write BW (GiB/s)	Devices (Capacity)
Config 1 (2+1) RAID5	NVMe	3.33/1.35	NVMe0 - NVMe2 (150, 120, 100 GiB)
	SATA	0.48/0.43	SATA0 - SATA2 (80, 60, 20 GiB)
Config 2 (4+2) RAID6	NVMe	3.33/1.35	NVMe0 - NVMe3 (894, 894, 894 GiB)
	NVMe	3.33/1.28	NVMe4 (2969 GiB)
	NVMe	2.58/1.30	NVMe5 (894 GiB)
	SATA	0.48/0.43	SATA0 - SATA3 (447,447,447,1740 GiB)

tonic [131] cluster, spanning an entire data center; (2) 10 FIU traces [88] collected from an all-flash array based HPC testbed [18]; and (3) 4 synthetic workloads, including sequential full-stripe read/write and random partial-stripe read/write.

Methodology. We evaluate two of our designs, `paRAID` and `paRAID-s` (static data layout without adaptive placement), against five state-of-the-art systems.

We provide a brief description for each compared design:

1. `mdraid` [121] is the default Linux RAID utility.
2. `StRAID` [164] assigns a dedicated worker thread for each stripe-write to improve parallelism and reduce lock contentions. It also opportunistically aggregates write IOs.
3. `SWAN` [83] adopts a spatial separation approach that partitions SSDs into front-end and back-end groups to alleviate GC interference.
4. `FusionRAID` [64] utilizes RAID declustering and request redirection to avoid latency spikes and improve performance consistency.

We answer the following questions in the evaluation:

- Can paRAID manage heterogeneous devices and improve performance under real-world workloads? (§ 6.5.2)
- Can paRAID improve system sustainability? (§ 6.5.3)
- Can the data organization model utilize disk capacity efficiently? (§ 6.5.4)
- Is the learning-based addressing approach efficient? (§ 6.5.5)
- How does paRAID perform under homogeneous RAID configurations? (§ 6.5.6)
- How does paRAID perform under degraded mode? (§ 6.5.7)

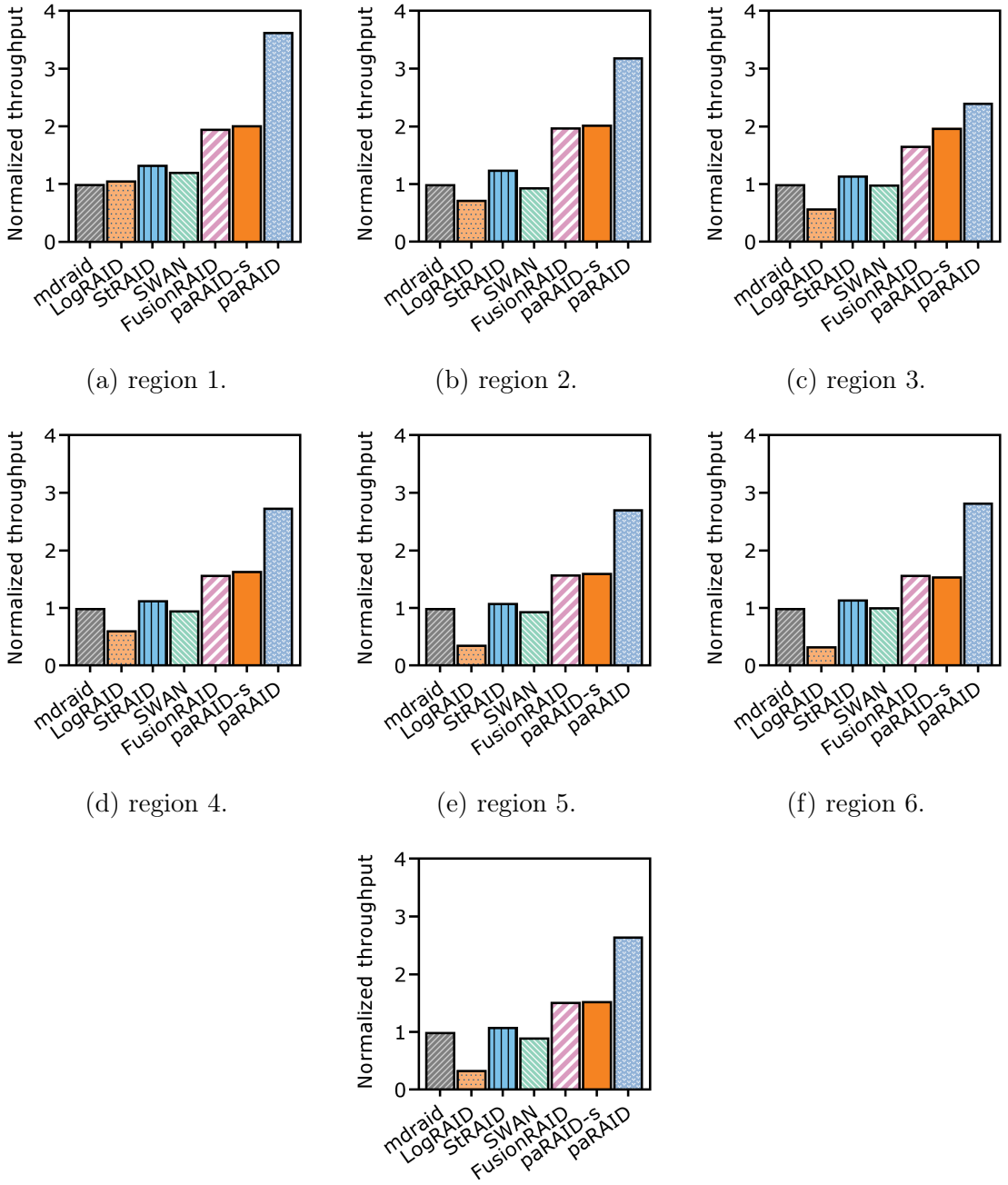
6.5.2. Overall Performance

In this section, we evaluate the performance of paRAID with two configurations under Meta and FIU workloads.

RAID5

We first configure a disk pool with 3 NVMe SSDs and 3 SATA SSDs, each having a different capacity (Table 9, config 1). Upon that, we build a RAID5 array where each stripe consists of 2 data chunks and 1 parity chunk. For paRAID and paRAID-s, the initial data layout is determined based on the data organization model and the profiled device performance, which takes 0.09 seconds in this experiment. For the conventional RAID, we manually partition the disks and aggregate each sub-RAID using Linux device-mapper [158]. The traces are issued by 8 worker threads concurrently.

Meta traces. Figure 55 shows the throughput of paRAID and other designs across 7 workloads from Meta. paRAID delivers significantly better performance than conventional solutions, where device heterogeneity is not adequately considered in their designs. In particular, paRAID outperforms mdraid by 2.63× under region 1 workload and by 1.9× on average across all workloads. We observe that LogRAID delivers a lower performance than mdraid on average, which is because LogRAID targets to optimize write performance while the traces are read/write mixed. It not only introduces an additional layer of indirection between the host and devices, but also requires AFA-level garbage collection, which is also bottlenecked by the poor-performing devices. paRAID-s achieves a similar performance with FusionRAID, due to its optimized logical volume layout and more aligned device performance within stripes. By adaptively mapping hot and cold data chunks, paRAID outperforms conventional solutions by up to 2.63× and achieves better utilization of disk components, as shown in Figure 56. Figure 57 plots latency results at major percentiles. Compared to conventional approaches, paRAID decreases the 99th and 99.9th latency by



(a) region 1. (b) region 2. (c) region 3.

(d) region 4. (e) region 5. (f) region 6.

(g) region 7.

Figure 55: Performance result of RAID5 under Meta traces.

46% and 48%, respectively.

FIU traces. Figure 58 compares the performance results under FIU workloads (showing

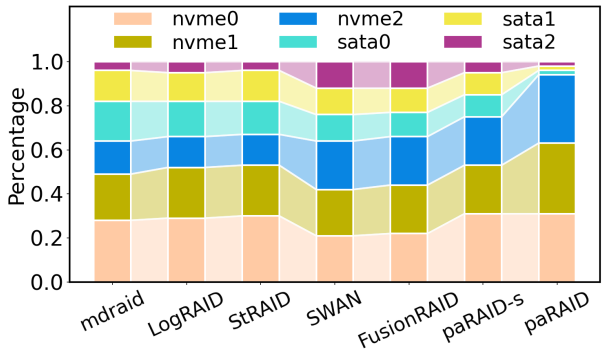


Figure 56: IO distribution under Meta workloads.

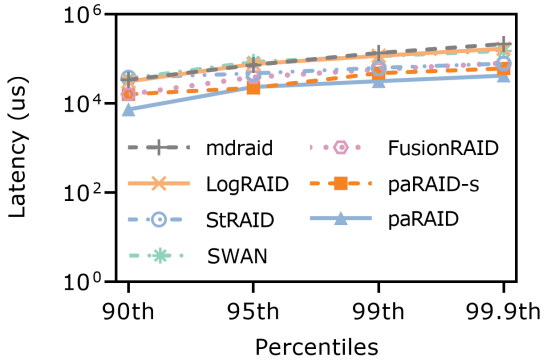
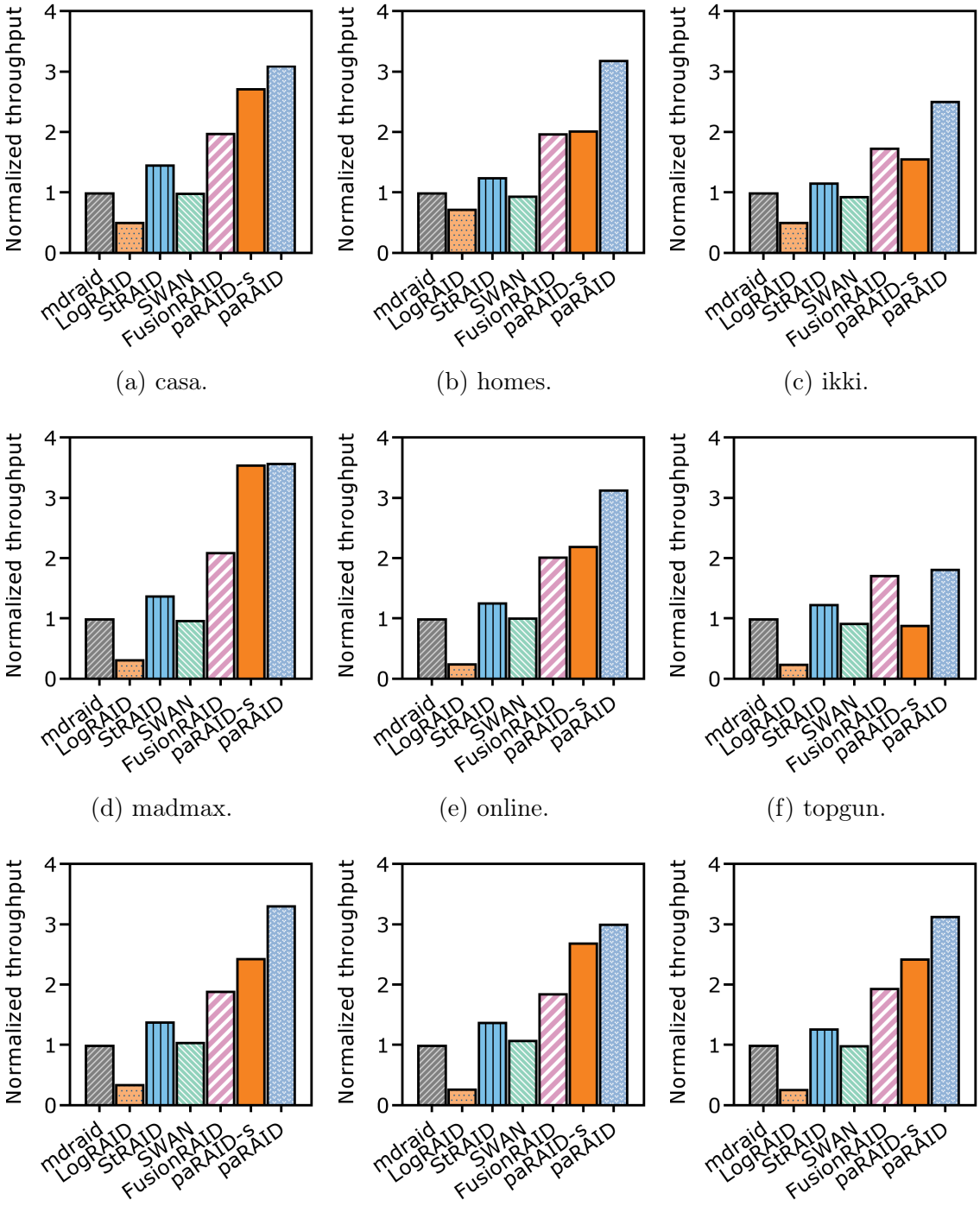


Figure 57: IO latency under Meta workloads.

9 traces due to limited space). Overall, paRAID outperforms mdraid by 1.99× on average and by up to 2.52× under the madmax workload, as shown in Figure 58d. Based on the results, we make the following observations. First, the conventional approaches fail to manage heterogeneous SSDs efficiently. The throughput is bottlenecked by the SATA SSDs. Second, although techniques like spatial partition (SWAN) and RAID declustering (FusionRAID) improve RAID performance with homogeneous devices, they present limited effectiveness under heterogeneous environments. Specifically, FusionRAID only achieves 60% of the paRAID performance on average. Third, to optimize the usage of NVMe SSDs, it is necessary to tune the data layout adaptively, as indicated by the performance difference between paRAID-s and paRAID.

RAID6

We next investigate the performance benefits of paRAID under RAID6 configuration. For this experiment, the underlying disk pool consists of 10 SSDs with different characteristics (Table 9, config 2). Each data stripe contains 4 data chunks and 2 parity chunks, a common RAID6 configuration [177]. The formulated data organization model takes 20.48 seconds to solve on our server with this configuration, due to the more complex search space compared to configuration 1. However, such overhead only happens once during the RAID

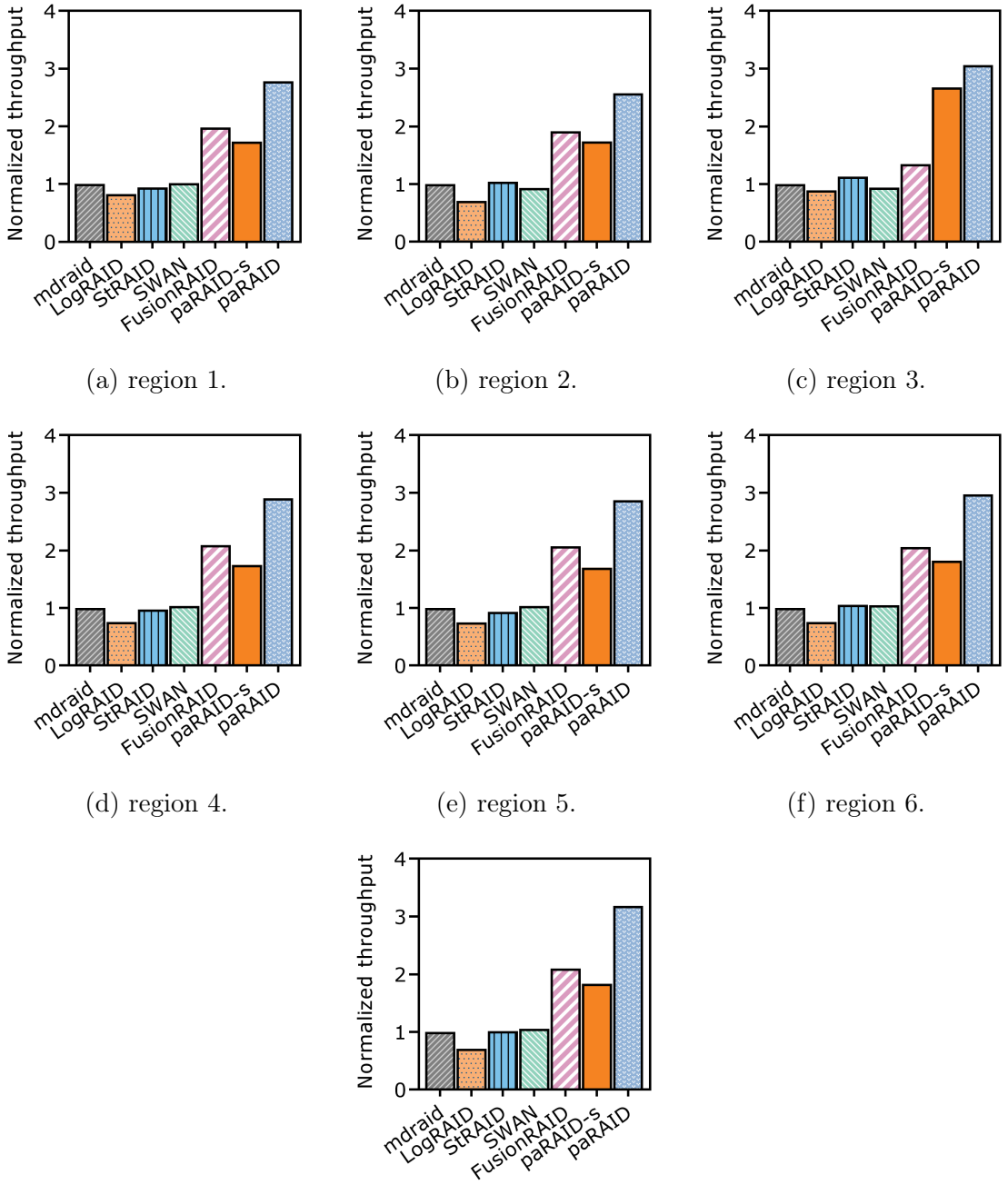


(a) casa. (b) homes. (c) ikki.

(d) madmax. (e) online. (f) topgun.

(g) webmail. (h) webresearch. (i) webusers.

Figure 58: Performance result of RAID5 under FIU traces.



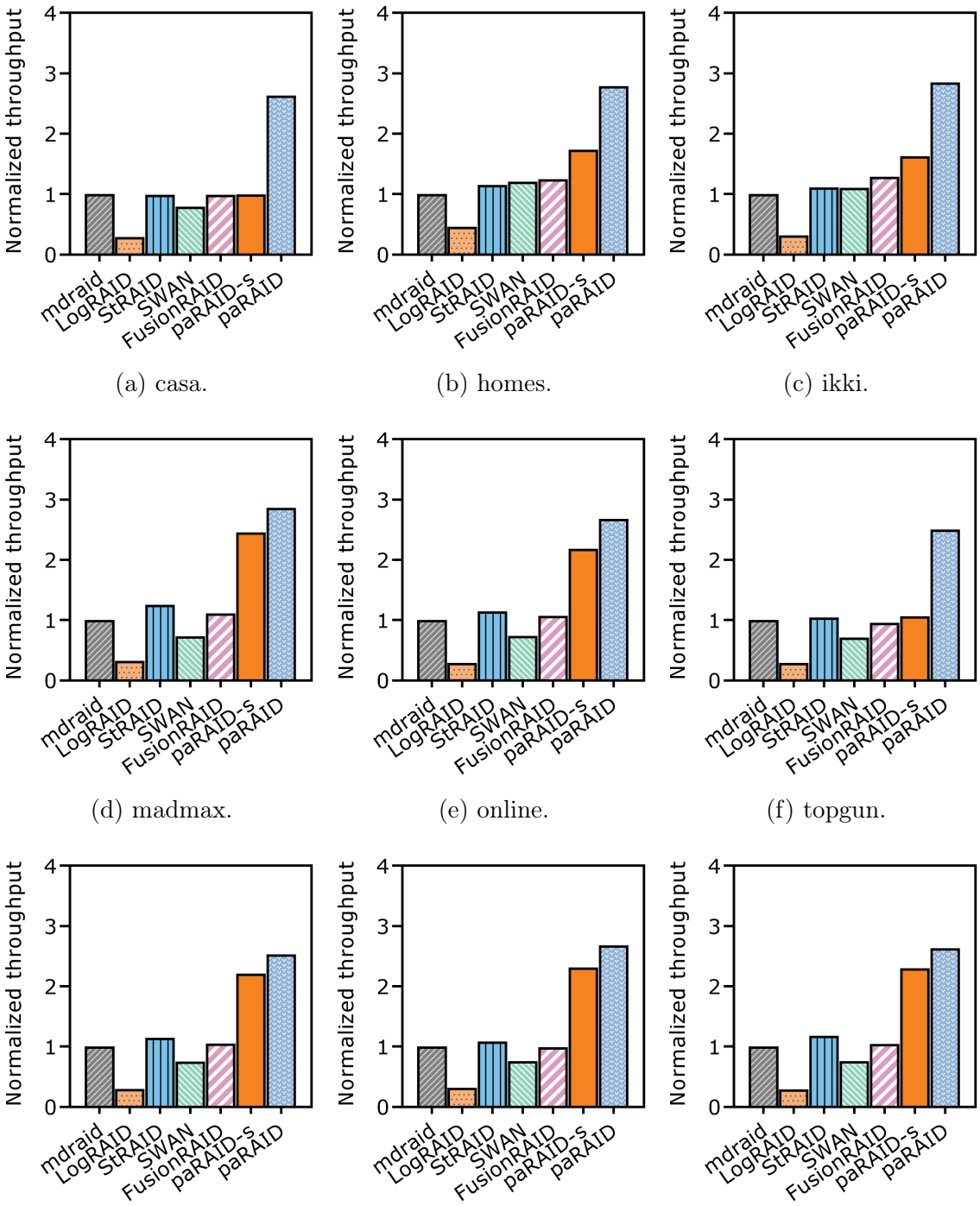
(a) region 1. (b) region 2. (c) region 3.

(d) region 4. (e) region 5. (f) region 6.

(g) region 7.

Figure 59: Performance result of RAID6 under Meta traces.

initialization stage. Moreover, solver parameters (e.g., `gapRel` or `timeLimit`) can be tuned to optimize model solving process.



(a) casa. (b) homes. (c) ikki.

(d) madmax. (e) online. (f) topgun.

(g) webmail. (h) webresearch. (i) webusers.

Figure 60: Performance result of RAID6 under FIU traces.

Meta traces. Figure 59 shows the performance results of the configured RAID6 under Meta workloads. With a larger data stripe size, the overall performance is more likely to be bottlenecked by SATA SSDs. Under the heterogeneous setting, LogRAID performs worse than mdraid due to its addressing and garbage collection overhead. StRAID and SWAN deliver negligible benefits. For SWAN, its partitioning algorithm requires devices to have similar performance, limiting its effectiveness in heterogeneous environments. FusionRAID performs better than mdraid thanks to its request redirection scheme. By mitigating intro-stripe performance discrepancy, paRAID-s achieves a comparable performance to FusionRAID on average. Finally, with adaptive data placement, paRAID outperforms mdraid by 1.91× across the workloads.

FIU traces. Figure 60 plots the results of RAID6 under FIU traces. Since the FIU workloads contain more small write IOs, StRAID performs better than previous cases and benefits more from its run-to-complete I/O processing and parity cache design. On average, the performance of paRAID is higher than mdraid by 1.68×.

6.5.3. Sustainability

To assess sustainability, we investigate the carbon emissions of different AFA systems for a 5-year deployment. We use the ACT model [48] to estimate operational and embodied emissions from flash and DRAM, and optimistically assume that different flash densities will have the same cost and emissions per cell [111]. The embodied emission values are based on Seagate Nytro 3331 for flash devices and a 10nm DDR4 for DRAM. We calculate all results based on the average performance of each system across workloads.

Figure 61 shows the estimated emission per year for each system. The RAID6 configuration generates more emissions than RAID5 due to the larger aggregated capacity. LogRAID achieves poor sustainability because of its relatively lower performance and higher DRAM overhead for metadata (i.e., roughly 0.51% of the total storage capacity). On the

other hand, FusionRAID requires lower memory for L2P mapping (5 bytes per 64 KiB) and achieves better sustainability. paRAID achieves the lowest emission and improves sustainability by 65% compared to mdraid due to the higher overall performance and low metadata overhead.

6.5.4. Data Organization Model

We evaluate the data organization model under RAID configurations with varying disk pool sizes (N) and capacity variation factors ($\frac{\sigma}{\mu}$). The average disk capacity and data stripe width are set as 10 TiB and 6 $\times$ 64KiB (i.e., six 64KiB chunks) respectively. We compare the derived logical capacity from the model with the aggregated physical capacity of disks.

As shown in Figure 62, the derived data organization efficiently utilizes the capacity of each disk, leading to the utilization rate of 100% in most cases. On average, it outperforms the manually configured data layout by 19%, which always prioritizes the device with the largest available capacity to maximize the AFA address space. In the case of $\frac{\sigma}{\mu} = 0.5$ and $N = 10$, paRAID achieves a utilization rate of 93%. This reduction is due to (1) the high capacity variation factor and (2) the large data stripe width (6 chunks over 10 SSDs).

6.5.5. Performance of Learned Addressing

As shown in Table 10, the learned index models take 1,328 bytes and 14,288 bytes of memory under the RAID5 and RAID6 configurations, which is only 2.1 bytes per GiB storage capacity. The overhead for reference and update is low, thanks to the simplicity of linear models. Specifically, the reference operation takes 72 ns and 76 ns on average, while the

Table 10: Performance of the linear addressing models.

System	AFA Capacity	Model Size	Reference	Update
RAID5	528 GiB	1,328 bytes	72 ns	145 ns
RAID6	8.51 TiB	14,288 bytes	76 ns	183 ns

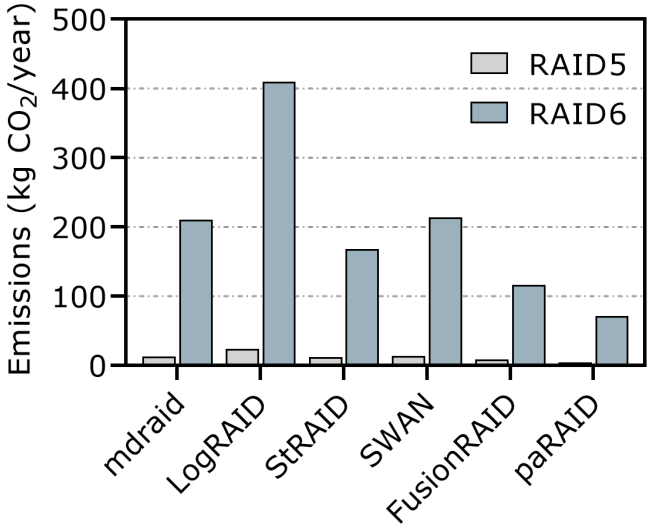


Figure 61: Yearly carbon emissions for different AFA systems. paRAID reduces carbon emissions by 65% compared to mdraid, due to the higher overall performance and low metadata overhead.

update takes 145 ns and 183 ns for RAID5 and RAID6.

6.5.6. Homogeneous Environment

We use 3 NVMe SSDs of the same model to build a homogeneous (2+1) RAID5 array with 64 KiB chunk size. Figure 63 plots the results under four synthetic workloads: (1) P-write: 64 KiB random partial-stripe writes; (2) F-write: 192 KiB sequential full-stripe writes; (3) P-read: 64 KiB random partial-stripe reads; and (4) F-read: 192 KiB sequential full-stripe reads. Overall, paRAID achieves a comparable performance to StRAID and outperforms mdraid by 2.68× for large sequential writes and by 3.1× for small random writes. FusionRAID outperforms paRAID by a small margin under small random writes due to its two-phase writes mechanism. The performance of paRAID and paRAID-s is similar, as there is no room for data layout tuning. The performance of read workloads is similar across different designs.

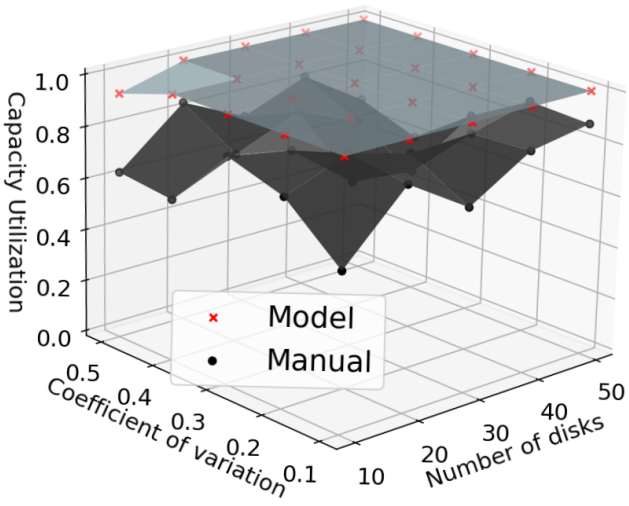


Figure 62: Capacity utilization under different settings.

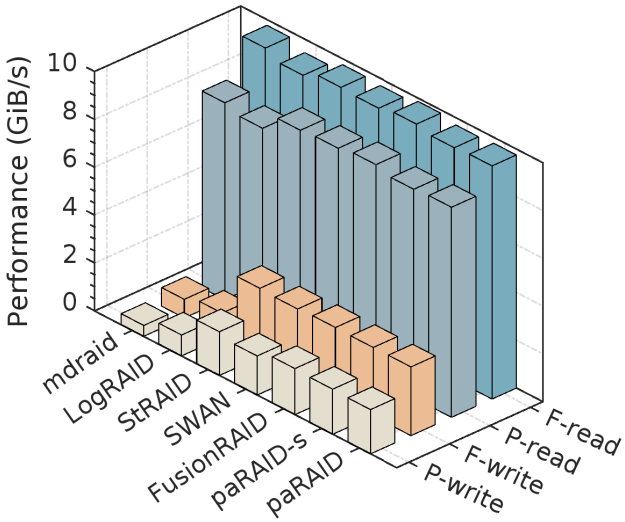
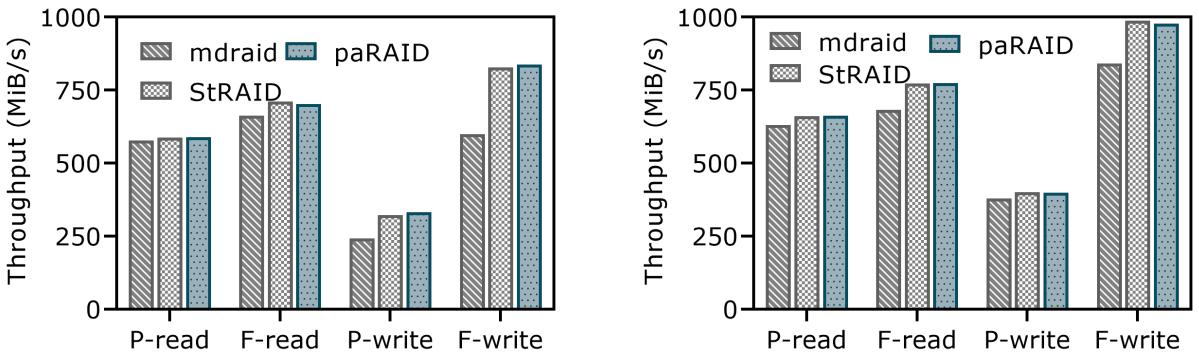


Figure 63: Performance comparison with same SSDs.

6.5.7. Degraded Mode

We compare the degraded mode performance of paRAID, StRAID, and mdraid in a RAID5 setup consisting of two NVMe SSDs and one SATA SSD. We examine two configurations: (1) One random NVMe SSD in the RAID array is set as failed and (2) The SATA SSD is set as failed.

The results in Figure 64 indicate that the read throughput of degraded paRAID and Linux MD differ by less than 5% on average. On the other hand, the write performance of degraded paRAID is 24% higher than that of Linux MD on average, due to its optimized write path and lower lock contention. The performance of degraded paRAID and StRAID is similar. We also observe that Figure 64b shows better performance than Figure 64a, which is because more I/Os are issued to the NVMe SSD when the SATA SSD fails. Reliability in paRAID is primarily managed through host-instructed RAID levels, similar to the existing RAID architectures. The data recovery and degraded mode work in the same way as the conventional RAID. Specifically, paRAID reconstructs data blocks by reading from



(a) NVMe SSD failure.

(b) SATA SSD failure.

Figure 64: paRAID degraded mode performance.

available disks and comparing their calculated parity results with the on-disk parity data.

6.6. Limitations

6.6.1. Dynamic Heterogeneity

The proposed architecture addresses the static disk heterogeneity when the system is initialized. However, the performance of SSDs can gradually degrade throughout their lifetime, which needs automatically optimizing data layout to adapt to dynamic disk heterogeneity. The challenge here is to minimize the data re-distribution overhead. One way to solve this problem is via data-remapping as discussed in Chapter 5. However, we believe that this approach may not be optimal as in RAID, data is located on different disks.

6.6.2. Reliability and Fault Tolerance.

At the current stage, paRAID operates in the same way as the legacy MD under disk failures. The missing data needs to be reconstructed from the remaining disks in the array. We plan to consider the case where the reliability of the SSDs is also different. Unlike existing disk-adaptive redundancy schemes [71, 72], we will exploit machine-learning techniques to predict SSD reliability changes and imbue this information into the block I/O layer for more efficient data management and fault handling.

6.6.3. Disk Replacement.

In paRAID, each drive can contribute to multiple stripe groups. Different from conventional RAID, when a larger capacity device is used, the added capacity can potentially combine with spare capacity from other disks to create new data stripes. By inserting them into existing stripe groups or developing new ones, these data stripes extend the storage capacity. We consider investigating recent works [68, 84] to coordinate paRAID and file systems for efficient online address space adjustment.

6.7. Conclusion

paRAID improves all-flash arrays' performance, utilization, and sustainability by exploiting the heterogeneity among modern SSDs. This requires a redesign of RAID to transition from conventional symmetric architectures to an asymmetric architecture. Experimental results show that paRAID outperforms state-of-the-art solutions in managing heterogeneous SSDs, allowing for higher disk utilization that reduces carbon emissions by up to 65%.

CHAPTER 7

Summary and conclusion

The explosive growth of data, driven by the era of big data, places significant stress on storage infrastructures of computing systems. As data centers evolve to meet increasing computational and storage requirements, efficiently managing this vast amount of data becomes a critical challenge for system architects. This thesis aims to optimize performance, reliability, and sustainability across the I/O stack, spanning device firmware, the block I/O layer, the file system, and RAID management, in NAND flash memory-based storage systems.

In particular, FF-SSD (Chapter 3) is a learning-based SSD aging framework that generates representative future wear-out states. The analysis of modern SSD internals (Chapter 4) uncovers the inefficiencies and unintended drawbacks of wear leveling algorithms in today's environments. CVSS (Chapter 5) addresses the fail-slow symptoms in solid-state drives (SSDs) by developing a software/hardware co-design solution. paRAID (Chapter 6) is a heterogeneity-aware RAID architecture for all-flash-array systems that optimizes storage utilization and sustainability.

7.1. Flash Memory Characterization

The proposed system, known as the *Fast-Forwardable SSD* (FF-SSD), is the first ML-based SSD aging framework of its kind. It achieves up to 99% accuracy in simulating aged states, significantly accelerates simulation times, and is adaptable across multiple SSD development platforms. The open-source artifacts of this work have been used by research groups at institutions such as Columbia University and Boston University.

7.2. Emerging Storage Device Internals

We quantify the performance of wear leveling (WL) in the context of modern flash memory with reduced endurance. Unfortunately, our findings show that existing WL exhibits limited effectiveness and may generate counter-productive results. Instead of designing a new wear leveling algorithm that patches these issues, we fundamentally ask if wear leveling is worth the trouble. Our findings shed light on the design of next-generation storage devices and reveal that WL may become an artifact of the past. This work receives the best paper award in HotStorage 2022 from Samsung Semiconductors.

7.3. File System and Block I/O

By enabling storage systems to dynamically adapt their capacity over time, the capacity-variant storage system (CVSS) for SSDs provides a practical solution to the growing challenges of flash storage reliability and performance stability. Experimental results have demonstrated practical benefits, including up to a 53% reduction in latency, a 316% improvement in throughput, and a 327% increase in device lifetime under real workloads.

7.4. RAID and All-Flash Array

paRAID redesigns RAID to transition from conventional symmetric architectures to a more flexible, asymmetric architecture that aligns with the unique characteristics of each SSD. Experiments with this system reveal a potential reduction in carbon emissions by up to 65%, making it a more sustainable storage architecture for managing emerging storage devices.

CHAPTER 8

Future research directions

I am excited to keep exploring new directions and to contribute solutions that address the evolving challenges of our data-intensive world. My research will continue to focus on optimizing storage systems through three key aspects: performance, reliability, and sustainability. Specifically, the following research directions are planned to be initiated.

8.1. HeteroRAID

8.1.1. Overview

We propose to improve the performance and sustainability of all-flash array (AFA) storage under heterogeneous device environments for modern solid-state drives (SSDs). The key insight behind this project is to adaptively and differentially use underlying storage components based on their unique capacity, performance, and reliability characteristics. Our prior work [68] has shown the presence of dynamic heterogeneity in modern SSDs. Building on this, we aim to design a heterogeneity-aware AFA system that transitions from traditional symmetric architectures to an asymmetric architecture, enabling the full utilization of all storage devices for optimized performance and sustainability.

8.1.2. Motivation

All-flash arrays (AFAs), leveraging the high performance, reliability, and compact form factor of solid-state drives (SSDs), have emerged as a promising solution to meet the growing storage demand in the big data era [64, 83]. However, the overall architecture of existing AFA systems is built around the assumption that the underlying storage components are homogeneous. This architecture results in significant disk under-utilization when considering heterogeneity among storage devices, particularly for modern SSDs [66]. The

balanced data layout requires the underlying storage devices to be roughly the same size. Otherwise, the aggregate capacity is determined by the minimal capacity device, making devices with larger capacity underutilized. Moreover, by evenly spreading data across the disks, the overall system performance and reliability are bottlenecked by the poor-performing drives.

This inefficiency not only impacts performance but also exacerbates environmental challenges. As flash annual capacity production exceeds 765 Exabytes, the associated embodied emissions amount to 122M metric tonnes of CO2 [186], equivalent to the average annual CO2 emissions of 28M people [155]. By 2030, this figure is expected to rise to the equivalent of over 150M people [38]. Addressing these sustainability concerns necessitates the development of more efficient and sustainable storage designs that can fully exploit device heterogeneity [186].

8.1.3. Proposed Work

Existing all-flash array designs fail to manage heterogeneous SSDs efficiently. Moreover, with the advancement in flash memory technology, modern SSDs, even for the same model, exhibit very different characteristics and degrade over time, leading to severe disk underutilization and reduced system sustainability. This work will build a redundant array of independent disks (RAID) variant that factors in dynamic disk heterogeneity. The proposed project has the following planned phases, each building on top of the previous.

• *Phase 1: Fail-slow in RAID.* The first phase aims to characterize the fail-slow symptoms in RAID under various configurations. Since there is no framework that allows for a full-stack study on SSD's fail-slow symptoms in the RAID environment, we start by building a RAID emulator capable of supporting various system settings while accurately emulating the aging behaviors of the underlying SSDs. We will then build lightweight ML models for

predicting the future performance and reliability characteristics of an SSD given different system configurations. We will focus on extracting the internal overheads such as the write amplification factor, and use this to generate representative future device states.

• *Phase 2: Metrics for Sustainability in AFA.* One of the key prerequisites for sustainable systems is the ability to define and measure sustainability in a holistic manner [40]. As business management thinker Peter Drucker rightly said "if you can't measure it, you can't improve it" [31]. In this phase, the central question we aim to address is how can we establish comprehensive metrics and cost functions that accurately capture the sustainability of each component within AFA systems. Achieving true "sustainable" storage systems will require taking into account all factors that contribute to the sustainability footprint. We will present new candidate metrics for sustainability beyond traditional power/energy metrics.

Existing solutions often focus on a single, possibly incomplete metric. For example, Power Usage Effectiveness (PUE) [5] focuses on total energy consumption but ignores the energy source (i.e., clean energy vs. grid energy) and the wear-and-tear of storage components. Other metrics such as Carbon- [7] or WaterUsage Effectiveness [6] or Green PUE [37] are too coarse-grained and cannot be applied to individual requests. Instead, our candidate metric will incorporate the following aspects [40]: (1) operational energy, (2) energy source cleanliness, (3) embodied costs, (4) device wear, (5) recycling costs, and (6) material consumption and human costs. The proposed metrics have the potential to significantly impact sustainable storage practices by incentivizing end-users and developers to engage in sustainable computing initiatives within the storage ecosystem.

• *Phase 3: Sustainability under heterogeneity.* Building on the holistic sustainability metrics developed in Phase 2, this phase focuses on improving the sustainability of all-flash arrays by integrating data and device characteristics into the RAID management layer.

Specifically, we aim to optimize data placement and utilization of heterogeneous storage components to address the inefficiencies of traditional RAID architectures.

Unlike conventional RAID systems that rely on a balanced data layout and fixed redundancy across all data, our approach leverages insights from our prior work [66] to avoid the underutilization and inefficiency caused by such rigid designs. We propose a dynamic and adaptive data management strategy guided by the following principles:

- Data Lifetime: Long-lived data or read-intensive data requires higher reliability and should be placed on more reliable storage devices. This minimizes the risk of data loss while improving the sustainability of the storage system.

- Storage Capacity, Performance, and Reliability: The available capacity of each device will be used to ensure efficient utilization, particularly for devices with larger capacities, which are underutilized in traditional systems. The performance and reliability profiles of each storage component will also be considered to optimize placement for workloads requiring specific performance guarantees.

- SSD Garbage Collection: Traditional RAID-like systems use a fixed-size data chunks, which fails to exploit the tradeoffs between SSD internal parallelism and GC efficiency. This rigid chunking strategy can lead to suboptimal performance and increased write amplification. In contrast, we propose a disk- and workload-adaptive chunking scheme that dynamically adjusts chunk sizes based on the underlying device characteristics and workload profiles. By tailoring chunk sizes, we can better align with SSD internal parallelism to maximize throughput, while simultaneously minimizing GC overhead through more efficient write patterns.

- Sustainability Cost: Placement decisions must balance overall per-data sustainability cost (in dollars per lifetime per size) based on the workloads.

We will develop a mathematical model that formalizes the placement problem as an optimization task. Additionally, we will explore machine learning techniques (e.g., clustering + KNN) to facilitate decision-making by predicting optimal configurations based on historical data and workload patterns.

The effectiveness of the proposed system will be evaluated using the metrics from Phase 2 to ensure that the design delivers measurable improvements in energy efficiency, storage utilization, and overall sustainability. This iterative evaluation process will also provide feedback for refining the design, ensuring alignment with sustainability goals.

• *Phase 4: AFA over disaggregated storage.* This phase focuses on extending the scope to disaggregated storage using the NVMe-over-Fabrics (NVMe-oF) protocol. While aggregating multiple SSDs can significantly increase storage bandwidth, it often exceeds the capacity of even high-end RDMA NICs (e.g., 100 Gbps), presenting a challenge to fully leverage the performance potential of modern SSDs. To address this, we will explore approaches that utilize SSD internal hardware resources and implement a host/device co-design. This co-design will enable SSDs to communicate directly with their peers, bypassing the need for continuous intervention from the RAID controller, thereby optimizing performance and sustainability in disaggregated environments.

APPENDIX

Mathematical Model for Capacity Utilization in a RAID System

A.1. Capacity Utilization with Heterogeneous SSDs

The capacity utilization (CU) of a RAID system can be calculated using the formula:

$$\text{CU} = \frac{\text{Minimum Disk Capacity} \times N}{\sum_{i=1}^{N} C_i}$$

where:

- N is the number of disks.
- C_i is the capacity of the i-th disk.

A.2. Assumptions

1. **Disk Capacity Variable**: Disk capacities (C_i) are independently and identically distributed random variables.
2. **Uniform Distribution**: For simplification, we assume that disk capacities follow a uniform distribution over a range $[a, b]$.

A.3. Statistical Measures

Mean Disk Capacity (μ):

$$\mu = \frac{a + b}{2}$$

Variance of Disk Capacities (σ^2):

$$\sigma^2 = \frac{(b-a)^2}{12}$$

Standard Deviation (σ):

$$\sigma = \frac{b-a}{2\sqrt{3}}$$

Coefficient of Variation (CV):

$$\text{CV} = \frac{\sigma}{\mu}$$

A.4. Expected Minimum Disk Capacity

The expected value of the minimum disk capacity among N disks uniformly distributed over $[a, b]$ is:

$$E[\text{Min}] = a + \frac{(b-a)}{N+1}$$

Express a and b in terms of μ and σ:

$$b - a = 2\sqrt{3}\sigma$$

$$a = \mu - \sqrt{3}\sigma$$

$$b = \mu + \sqrt{3}\sigma$$

Substitute back into $E[\text{Min}]$:

$$\begin{aligned} E[\text{Min}] &= \mu - \sqrt{3}\sigma + \frac{2\sqrt{3}\sigma}{N+1} \\ &= \mu - \sqrt{3}\sigma\left(1 - \frac{2}{N+1}\right) \\ &= \mu - \sqrt{3}\sigma\left(\frac{N-1}{N+1}\right) \end{aligned}$$

A.4.1. Calculating Expected Capacity Utilization

Substitute $E[\text{Min}]$ into the capacity utilization formula:

$$\text{Expected CU} = \frac{E[\text{Min}] \times N}{N\mu} = \frac{E[\text{Min}]}{\mu}$$

Thus:

$$\text{Expected CU} = \frac{\mu - \sqrt{3}\sigma\left(\frac{N-1}{N+1}\right)}{\mu} = 1 - \sqrt{3}\left(\frac{\sigma}{\mu}\right)\left(\frac{N-1}{N+1}\right)$$

Since $\text{CV} = \frac{\sigma}{\mu}$:

$$\text{Expected CU} = 1 - \sqrt{3} \times \text{CV} \times \left(\frac{N-1}{N+1}\right)$$

A.4.2. Final Model

$$\boxed{\text{Capacity Utilization (CU)} = 1 - \sqrt{3} \times \text{CV} \times \left(\frac{N-1}{N+1}\right)}$$

Alternatively,

$$\text{Capacity Underutilization (1 - CU)} = \sqrt{3} \times \text{CV} \times \left(\frac{N-1}{N+1}\right)$$

A.5. Example Calculations

1. Low Variance Scenario

Let CV $= 0.1$ and $N = 5$:

$$\begin{aligned}
\text{CU} &= 1 - \sqrt{3} \times 0.1 \times \left(\frac{5-1}{5+1}\right) \\
&= 1 - \sqrt{3} \times 0.1 \times \left(\frac{4}{6}\right) \\
&\approx 1 - 0.1155 \\
&\approx 0.8845
\end{aligned}$$

2. High Variance Scenario

Let CV $= 0.3$ and $N = 5$:

$$\begin{aligned}
\text{CU} &= 1 - \sqrt{3} \times 0.3 \times \left(\frac{5-1}{5+1}\right) \\
&= 1 - \sqrt{3} \times 0.3 \times \left(\frac{4}{6}\right) \\
&\approx 1 - 0.3464 \\
&\approx 0.6536
\end{aligned}$$

BIBLIOGRAPHY

[1] Nitin Agrawal, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Generating Realistic Impressions for File-System Benchmarking. In *Conference on File and Storage Technologies (FAST)*, pages 125–138. USENIX Association, 2009. URL: http://www.usenix.org/events/fast09/tech/full%5C_papers/agrawal/agrawal.pdf.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference (ATC)*, pages 57–70. USENIX Association, 2008. URL: https://www.usenix.org/conference/atc08/design-tradeoffs-ssd-performance.

[3] Michael Allison, Arun George, Javier Gonzalez, Dan Helmick, Vikash Kumar, Roshan Nair, and Vivek Shah. Towards Efficient Flash Caches with Emerging NVMe Flexible Data Placement SSDs. In *European Conference on Computer Systems (EuroSys)*, pages 1142–1160. ACM, 2025. URL: https://dl.acm.org/doi/10.1145/3689031.3696091.

[4] Amazon. How Amazon Achieves Near-Real-Time Renewable Energy Plant Monitoring to Optimize Performance Using AWS. https://aws.amazon.com/blogs/industries/amazon-achieves-near-real-time-renewable-energy-plant-monitoring-to-optimize-performance-using-aws/, 2024.

[5] Victor Avelar, Dan Azevedo, Alan French, and Emerson Network Power. PUE: A Comprehensive Examination of the Metric, 2012.

[6] Dan Azevedo, Symantec Christian Belady, and Jack Pouchet. Water Usage Effectiveness (WUE): A Green Grid Datacenter Sustainability Metric, 2011.

[7] Dan Azevedo, Michael Patterson, Jack Pouchet, and Roger Tipley. Carbon Usage Effectiveness (CUE): A Green Grid Data Center Sustainability Metric, 2010.

[8] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Conference on File and Storage Technologies (FAST)*, pages 223–238. USENIX Association, 2008. URL: https://www.usenix.org/conference/fast-08/analysis-data-corruption-storage-stack.

[9] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential RAID: Rethinking RAID for SSD Reliability. In *European Conference on Computer Systems (EuroSys)*, pages 15–26. ACM, 2010. URL: https://dl.acm.org/doi/10.1145/1807060.1807061.

[10] 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 *Annual Technical Conference (ATC)*, pages 689–703. USENIX Association, 2021. URL: https://www.usenix.org/conference/atc21/presentation/bjorling.

[11] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Conference on File and Storage Technologies (FAST)*, pages 359–374. USENIX Association, 2017. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling.

[12] Brian Cooper. Yahoo Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/, 2010.

[13] Cactus Technologies. SOLID STATE DRIVES 101. https://www.cactus-tech.com/wp-content/uploads/2019/04/Solid-State-Drives-101-EBook.pdf, 2019.

[14] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives. In *Proceedings of the IEEE*, pages 1666–1704. IEEE, 2017. URL: https://ieeexplore.ieee.org/document/8013174.

[15] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. Threshold Voltage Distribution in MLC NAND Flash Memory: Characterization, Analysis, and Modeling. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1285–1290. ACM, 2013. URL: https://ieeexplore.ieee.org/document/6513712.

[16] Yu Cai, Yixin Luo, Erich F. Haratsch, Ken Mai, and Onur Mutlu. Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 551–563. IEEE, 2015. URL: https://ieeexplore.ieee.org/document/7056062.

[17] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrián Cristal, Osman S. Ünsal, and Ken Mai. Flash Correct-and-refresh: Retention-aware Error Management for Increased Flash Memory Lifetime. In *International Conference on Computer Design (ICCD)*, pages 94–101. IEEE Computer Society, 2012. URL: https://ieeexplore.ieee.org/document/6378623.

[18] CAMEL. CAMEL's High Performance Computing Systems. https://camelab.org/pmwiki.php?n=Main.Resource, 2024.

[19] Chandranil Chakraborttii and Heiner Litz. Improving the Accuracy, Adaptability, and Interpretability of SSD Failure Prediction Models. In *Symposium on Cloud Computing (SOCC)*, pages 120–133. ACM, 2020. URL: https://dl.acm.org/doi/10.1145/3419111.3421300.

[20] Li-Pin Chang. On Efficient Wear Leveling for Large-scale Flash-memory Storage Systems. In *Symposium on Applied Computing (SAC)*, pages 1126–1130. ACM, 2007. URL: https://dl.acm.org/doi/10.1145/1244002.1244120.

[21] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time Garbage Collection for Flash-memory Storage Systems of Real-time Embedded Systems. In *ACM Transactions on Embedded Computing Systems (TECS)*, pages 837–863. ACM, 2004. URL: https://dl.acm.org/doi/10.1145/1027794.1027801.

[22] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance Enhancement of Flash-Memory Storage, Systems: An Efficient Static Wear Leveling Design. In *Design Automation Conference (DAC)*, pages 212–217. ACM, 2007. URL: https://dl.acm.org/doi/10.1145/1278480.1278533.

[23] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Improving Flash Wear-Leveling by Proactively Moving Static Data. In *IEEE Transactions on Computers (TOC)*, pages 53–65. IEEE, 2010. URL: https://ieeexplore.ieee.org/document/5255228.

[24] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 181–192. ACM, 2009. URL: https://dl.acm.org/doi/10.1145/2492101.1555371.

[25] Fu-Hsin Chen, Ming-Chang Yang, Yuan-Hao Chang, and Tei-Wei Kuo. PWL: A Progressive Wear Leveling to Minimize Data Migration Overheads for NAND Flash Devices. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 1209–1212. IEEE, 2015. URL: https://ieeexplore.ieee.org/document/7092571.

[26] Zhe Chen and Yuelong Zhao. DA-GC: A Dynamic Adjustment Garbage Collection Method Considering Wear-leveling for SSD. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 475–480. ACM, 2020. URL: https://dl.acm.org/doi/10.1145/3386263.3406943.

[27] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. Using Data Clustering to Improve Cleaning Performance for Flash Memory. In *Software: Practice and Experience*, pages 267–290. ACM, 1999. URL: https://dl.acm.org/doi/10.1002/%28SICI%291097-024X%28199903%2929%3A3%253C267%3A%3AAID-SPE233%253E3.0.CO%3B2-T.

[28] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. Software Orchestrated Flash Array. In *International Conference on Systems and Storage (SYSTOR)*, pages 1–11. ACM, 2014. URL: https://dl.acm.org/doi/10.1145/2611354.2611360.

[29] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File Systems Fated for Senescence? Nonsense, Says Science! In *Conference on File and Storage Technologies (FAST)*, pages 45–58. USENIX Association, 2017. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/conway.

[30] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald E. Porter, and Martin Farach-Colton. Filesystem Aging: It's more Usage than Fullness. In *Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 15–21. ACM, 2019. URL: https://www.usenix.org/conference/hotstorage19/presentation/conway.

[31] Dave Lavinsky. The Two Most Important Quotes in Business. https://www.growthink.com/content/two-most-important-quotes-business, 2024.

[32] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. In *Commun. ACM*, pages 74–80. ACM, 2013. URL: https://doi.org/10.1145/2398356.2398364.

[33] Dell. VMAX All Flash Storage. https://www.dell.com/en-us/dt/storage/vmax-all-flash.htm, 2024.

[34] Peter Desnoyers. Analytic Modeling of SSD Write Performance. In *International Systems and Storage Conference (SYSTOR)*, pages 1–10. ACM, 2012. URL: https://doi.org/10.1145/2367589.2367603.

[35] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. ALEX: An Updatable Adaptive Learned Index. In *Conference on Management of Data (SIGMOD)*, pages 969–984. ACM, 2020. URL: https://dl.acm.org/doi/10.1145/3318464.3389711.

[36] Yajuan Du, Siyi Huang, Yao Zhou, and Qiao Li. Towards LDPC Read Performance of 3D Flash Memories with Layer-induced Error Characteristics. In *ACM Transactions on Design Automation of Electronic Systems (TDAE)*, pages 1–25. ACM, 2023. URL: https://doi.org/10.1145/3585075.

[37] AL-Hazemi Fawaz, Alaelddin Fuad Yousif Mohammed, Lemi Isaac Yoseke Laku, and Rayan Alanazi. PUE or GPUE: A Carbon-aware Metric for Data Centers. In *International Conference on Advanced Communication Technology (ICACT)*, pages 38–41. IEEE, 2019. URL: https://ieeexplore.ieee.org/document/8701895.

[38] Forbes. Flash Memory Summit Announcements. https://www.forbes.com/sites/tomcoughlin/2022/08/13/2022-flash-memory-summit-announcements/?sh=2ffa73a11bf0, 2022.

[39] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. In *ACM Computing Surveys (CSUR)*, pages 138–163. ACM, 2005. URL: https://dl.acm.org/doi/10.1145/1089733.1089735.

[40] Anshul Gandhi, Kanad Ghose, Kartik Gopalan, Syed Hussain, Dongyoon Lee, David Liu, Zhenhua Liu, Patrick McDaniel, Shuai Mu, and Erez Zadok. Metrics for Sustainability in Data Centers. In *Workshop on Sustainable Computer Systems Design and Implementation (HotCarbon)*, pages 1–10. ACM, 2022. URL: https://www3.cs.stonybrook.edu/~anshul/hotcarbon22_sassy.pdf.

[41] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Conference on File and Storage Technologies (FAST)*, pages 149–165. USENIX Association, 2017. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan.

[42] Thomas Gleixner, Frank Haverkamp, and Artem Bityutskiy. UBI - Unsorted Block Images. http://linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf, 2006.

[43] Google. Advancing Clean Energy Commitments. https://blog.google/outreach-initiatives/sustainability/new-ways-were-advancing-our-clean-energy-commitments-in-the-us/, 2024.

[44] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber: Enabling Precise Full-system Simulation with Detailed Modeling of All SSD Resources. In *International Symposium on Microarchitecture (MICRO)*, pages 469–481. IEEE, 2018. URL: https://doi.org/10.1109/MICRO.2018.00045.

[45] Laura M. Grupp, John D. Davis, and Steven Swanson. The Bleak Future of NAND Flash Memory. In *Conference on File and Storage Technologies (FAST)*, pages 1–8. USENIX Association, 2012. URL: https://www.usenix.org/system/files/conference/fast12/grupp.pdf.

[46] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Conference on File and Storage Technologies (FAST)*, pages 1–14. USENIX Association, 2018. URL: https://www.usenix.org/system/files/conference/fast18/fast18-gunawi.pdf.

[47] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In *Conference on File and Storage Technologies (FAST)*, pages 91–103. USENIX Association, 2011. URL: https://www.usenix.org/legacy/event/fast11/tech/full_papers/Gupta.pdf.

[48] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. ACT: Designing Sustainable Computer Systems with An Architectural Carbon Modeling Tool. In *International Symposium on Computer Architecture (ISCA)*, pages 784–799. ACM, 2022. URL: https://dl.acm.org/doi/10.1145/3470496.3527408.

[49] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing Carbon: The Elusive Environmental Footprint of Computing. In *Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867. IEEE, 2021. URL: https://dl.acm.org/doi/abs/10.1109/MM.2022.3163226.

[50] Keonsoo Ha, Jaeyong Jeong, and Jihong Kim. An Integrated Approach for Managing Read Disturbs in High-Density NAND Flash Memory. In *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 1079–1091. IEEE, 2016. URL: https://ieeexplore.ieee.org/document/7342903.

[51] 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 *Annual Technical Conference (ATC)*, pages 759–771. USENIX Association, 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/hahn.

[52] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–162. USENIX Association, 2021. URL: https://www.usenix.org/conference/osdi21/presentation/han.

[53] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Symposium on Operating Systems Principles (SOSP)*, pages 168–183. ACM, 2017. URL: https://ucare.cs.uchicago.edu/pdf/sosp17-mittos.pdf.

[54] Mingzhe Hao, Gokul Soundararajan, Deepak R. Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Conference on File and Storage Technologies (FAST)*, pages 263–276. USENIX Association, 2016. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/hao.

[55] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–190. USENIX Association, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/hao.

[56] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *European Conference on Computer Systems (EuroSys)*, pages 127–144. ACM, 2017. URL: https://pages.cs.wisc.edu/~jhe/eurosys17-he.pdf.

[57] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman A. Pletka. Write Amplification Analysis in Flash-Based Solid State Drives. In *Annual International Conference on Systems and Storage (SYSTOR)*, pages 1–9. ACM, 2009. URL: https://dl.acm.org/doi/10.1145/1534530.1534544.

[58] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Conference on File and Storage Technologies (FAST)*, pages 375–390. USENIX Association, 2017. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/huang.

[59] IBM. ILOG CPLEX Optimization Studio. https://www.ibm.com/products/ilog-cplex-optimization-studio, 2024.

[60] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. Rethinking WOM Codes to Enhance the Lifetime in New SSD Generations. In *Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 1–8. USENIX Association, 2020. URL: https://www.usenix.org/conference/hotstorage20/presentation/jaffer.

[61] Jens Axboe. Flexible I/O Tester. https://github.com/axboe/fio/, 2005.

[62] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. The Pitfalls of Deploying Solid-state Drive RAIDs. In *Annual International Conference on Systems and Storage (SYSTOR)*, pages 1–13. ACM, 2011. URL: https://dl.acm.org/doi/10.1145/1993686.1993704.

[63] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. File Fragmentation in Mobile Devices: Measurement, Evaluation, and Treatment. In *IEEE Transactions on Mobile Computing (TMC)*, pages 2062–2076. IEEE, 2019. URL: https://ieeexplore.ieee.org/document/8462795.

[64] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In *Conference on File and Storage Technologies (FAST)*, pages 355–370. USENIX Association, 2021. URL: https://www.usenix.org/system/files/fast21-jiang.pdf.

[65] Ziyang Jiao, Janki Bhimani, and Bryan S. Kim. Wear leveling in SSDs considered harmful. In *Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 72–78. ACM, 2022. URL: https://dl.acm.org/doi/10.1145/3538643.3539750.

[66] Ziyang Jiao and Bryan S. Kim. Asymmetric RAID: Rethinking RAID for SSD Heterogeneity. In *Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 101–107. ACM, 2024. URL: https://dl.acm.org/doi/10.1145/3655038.3665952.

[67] Ziyang Jiao and Bryan S. Kim. Generating Realistic Wear Distributions for SSDs. In *Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 65–71. ACM, 2022. URL: https://dl.acm.org/doi/10.1145/3538643.3539757.

[68] Ziyang Jiao, Xiangqun Zhang, Hojin Shin, Jongmoo Choi, and Bryan S. Kim. The Design and Implementation of a Capacity-Variant Storage System. In *Conference on File and Storage Technologies (FAST)*, pages 159–176. USENIX Association, 2024. URL: https://www.usenix.org/system/files/fast24-jiao.pdf.

[69] Xavier Jimenez, David Novo, and Paolo Ienne. Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance. In *Conference on File and Storage Technologies (FAST)*, pages 47–59. USENIX Association, 2014. URL: https://www.usenix.org/conference/fast14/technical-sessions/presentation/jimenez.

[70] Myoungsoo Jung and Mahmut T. Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 203–216. ACM, 2013. URL: https://dl.acm.org/doi/10.1145/2465529.2465548.

[71] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, K. V. Rashmi, and Gregory R. Ganger. Tiger: Disk-Adaptive Redundancy Without Placement Restrictions. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 413–429. USENIX Association, 2022. URL: https://www.usenix.org/conference/osdi22/presentation/kadekodi.

[72] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, K. V. Rashmi, and Gregory R. Ganger. PACEMAKER: Avoiding HeART Attacks in Storage Clusters with Disk-adaptive Redundancy. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 369–385. USENIX Association, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/kadekodi.

[73] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. Geriatrix: Aging What You See and What You Don't See. A Dile System Aging Approach For Modern Storage Systems. In *Annual Technical Conference (ATC)*, pages 691–704. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/kadekodi.

[74] Saurabh Kadekodi, K. V. Rashmi, and Gregory R. Ganger. Cluster Storage Systems Gotta Have HeART: Improving Storage Efficiency by Exploiting Disk-reliability Heterogeneity. In *Conference on File and Storage Technologies (FAST)*, pages 345–358. USENIX Association, 2019. URL: https://www.usenix.org/conference/fast19/presentation/kadekodi.

[75] Aarati Kakaraparthy, Jignesh M. Patel, Kwanghyun Park, and Brian Kroth. Optimizing Databases by Learning Hidden Parameters of Solid State Drives. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 519–532. ACM, 2019. URL: https://dl.acm.org/doi/abs/10.14778/3372716.3372724.

[76] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed Solid-State Drive. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 13–17. USENIX, 2014. URL: https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang.

[77] Won-Kyung Kang, Dongkun Shin, and Sungjoo Yoo. Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD. In *ACM Transactions on Embedded Computing Systems (TECS)*, pages 1–20. ACM, 2017. URL: https://dl.acm.org/doi/10.1145/3126537.

[78] Swaroop Kavalanekar, Bruce L. Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *Symposium on Workload Characterization (IISWC)*, pages 119–128. IEEE, 2008. URL: 10.1109/IISWC.2008.4636097.

[79] Bryan S. Kim, Jongmoo Choi, and Sang Lyul Min. Design Tradeoffs for SSD Reliability. In *Conference on File and Storage Technologies (FAST)*, pages 281–294. USENIX Association, 2019. URL: https://www.usenix.org/system/files/fast19-kim-bryan.pdf.

[80] Bryan S. Kim, Eunji Lee, Sungjin Lee, and Sang Lyul Min. CPR for SSDs. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 201–208. ACM, 2019. URL: https://dl.acm.org/doi/10.1145/3317550.3321437.

[81] Bryan Suk Kim, Hyun Suk Yang, and Sang Lyul Min. AutoSSD: an Autonomic SSD Architecture. In *Annual Technical Conference (ATC)*, pages 677–690. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/kim.

[82] Han-joon Kim and Sang-goo Lee. A New Flash Memory Management for Flash Storage System. In *International Computer Software and Applications Conference (COMPSAC)*, pages 284–289. IEEE, 1999. URL: https://ieeexplore.ieee.org/document/812717.

[83] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *Annual Technical Conference (ATC)*, pages 799–812. USENIX Association, 2019. URL: https://www.usenix.org/conference/atc19/presentation/kim.

[84] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and Youjip Won. IPLFS: Log-Structured File System without Garbage Collection. In *Annual Technical Conference (ATC)*, pages 739–754. USENIX Association, 2022. URL: https://www.usenix.org/conference/atc22/presentation/kim-juwon.

[85] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Joo Young Hwang, Jongyoul Lee, and Jihong Kim. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *Conference on File and Storage Technologies (FAST)*, pages 295–308. USENIX Association, 2019. URL: https://www.usenix.org/conference/fast19/presentation/kim-taejin.

[86] Timothy Kim, Sanjith Athlur, Saurabh Kadekodi, Francisco Maturana, Dax Delvira, Arif Merchant, Gregory R Ganger, and KV Rashmi. Morph: Efficient File-Lifetime Redundancy Management for Cluster File Systems. In *Symposium on Operating Systems Principles (SOSP)*, pages 330–346. ACM, 2024. URL: https://dl.acm.org/doi/pdf/10.1145/3694715.3695981.

[87] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David Dillow, and Feiyi Wang. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-State Drives. In *Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2011. URL: https://ieeexplore.ieee.org/document/5937224.

[88] Miryeong Kwon, Jie Zhang, Gyuyoung Park, Wonil Choi, David Donofrio, John Shalf, Mahmut Kandemir, and Myoungsoo Jung. Tracetracker: Hardware/software Co-evaluation for Large-scale I/O Workload Reconstruction. In *Symposium on Workload Characterization (IISWC)*, pages 87–96. IEEE, 2017. URL: https://arxiv.org/abs/1709.04806.

[89] Damien Le Moal and Ting Yao. Zonefs: Mapping POSIX File System Interface to Raw Zoned Block Device Accesses. In *Linux Storage and Filesystems Conference (VAULT)*, pages 1–19. USENIX Association, 2020. URL: https://www.usenix.org/conference/vault20/presentation/lemoal.

[90] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Conference on File and Storage Technologies (FAST)*, pages 1–15. USENIX Association, 2015. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[91] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding Storage Traffic Characteristics on Enterprise Virtual Desktop Infrastructure. In *International Systems and Storage Conference (SYSTOR)*, pages 1–11. IEEE, 2017. URL: https://dl.acm.org/doi/10.1145/3078468.3078479.

[92] Jaeyong Lee, Myungsuk Kim, Wonil Choi, Sanggu Lee, and Jihong Kim. TailCut: Improving Performance and Lifetime of SSDs Using Pattern-aware State Encoding. In *Design Automation Conference (DAC)*, pages 409–414. ACM, 2022. URL: https://dl.acm.org/doi/10.1145/3489517.3530471.

[93] Sungjin Lee, Taejin Kim, Kyungho Kim, and Jihong Kim. Lifetime Management of Flash-based SSDs Using Recovery-aware Dynamic Throttling. In *Conference on File and Storage Technologies (FAST)*, pages 1–14. USENIX Association, 2012. URL: https://www.usenix.org/conference/fast12/lifetime-management-flash-based-ssds-using-recovery-aware-dynamic-throttling.

[94] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Symposium on Operating Systems Principles (SOSP)*, pages 17–34. ACM, 2023. URL: https://dl.acm.org/doi/10.1145/3600006.3613167.

[95] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminatahan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *Conference on File and Storage Technologies (FAST)*, pages 83–90. USENIX Association, 2018. URL: https://www.usenix.org/conference/fast18/presentation/li.

[96] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Symposium on Operating Systems Principles (SOSP)*, pages 263–279. ACM, 2021. URL: https://dl.acm.org/doi/10.1145/3477132.3483573.

[97] Shuwen Liang, Zhi Qiao, Sihai Tang, Jacob Hochstetler, Song Fu, Weisong Shi, and Hsing-Bung Chen. An Empirical Study of Quad-Level Cell (QLC) NAND Flash SSDs for Big Data Applications. In *International Conference on Big Data (Big Data)*, pages 3676–3685. IEEE, 2019. URL: https://ieeexplore.ieee.org/document/9006406.

[98] Jianwei Liao, Fengxiang Zhang, Li Li, and Guoqiang Xiao. Adaptive Wear Leveling in Flash-Based Memory. In *Computer Architecture Letters (LCA)*, pages 1–4. IEEE, 2015. URL: https://ieeexplore.ieee.org/document/6853308.

[99] Linux RAID. Linux RAID Setup. https://raid.wiki.kernel.org/index.php/RAID_setup, 2011.

[100] Linux RAID. Linux RAID Superblock. https://raid.wiki.kernel.org/index.php/RAID_superblock_formats, 2011.

[101] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing NAND Flash-based SSDs via Retention Relaxation. In *Conference on File and Storage Technologies (FAST)*, page 11. USENIX Association, 2012. URL: https://www.usenix.org/conference/fast12/optimizing-nand-flash-based-ssds-retention-relaxation.

[102] Shijun Liu and Xuecheng Zou. QLC NAND Study and Enhanced Gray Coding Methods for Sixteen-level-based Program Algorithms. In *Microelectronics Journal*, pages 58–66. ACM, 2017. URL: https://dl.acm.org/doi/abs/10.1016/j.mejo.2017.05.019.

[103] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. Perseus: A Fail-Slow Detection Framework for Cloud Storage Systems. In *Conference on File and Storage Technologies (FAST)*, pages 49–64. USENIX Association, 2023. URL: https://www.usenix.org/conference/fast23/presentation/lu.

[104] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. NVMe SSD Failures in the Field: the Fail-Stop and the Fail-Slow. In *Annual Technical Conference (ATC)*, pages 1005–1020. USENIX Association, 2022. URL: https://www.usenix.org/conference/atc22/presentation/lu.

[105] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. WARM: Improving NAND Flash Memory Lifetime with Write-hotness Aware Retention Management. In *Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015. URL: https://ieeexplore.ieee.org/document/7208284.

[106] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In *International Symposium on High-Performance Computer Architecture (HPCA)*, pages 504–517. IEEE, 2018. URL: https://ieeexplore.ieee.org/document/8327033.

[107] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. A Study of SSD Reliability in Large Scale Enterprise Storage Deployments. In *Conference on File and Storage Technologies (FAST)*, pages 137–149. USENIX Association, 2020. URL: https://www.usenix.org/conference/fast20/presentation/maneas.

[108] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Operational Characteristics of SSDs in Enterprise Storage Systems: A Large-Scale Field Study. In *Conference on File and Storage Technologies (FAST)*, pages 165–180. USENIX Association, 2022. URL: https://www.usenix.org/conference/fast22/presentation/maneas.

[109] Henry B. Mann. The Construction of Orthogonal Latin Squares. In *The Annals of Mathematical Statistics*, pages 418–423. JSTOR, 1942. URL: https://www.jstor.org/stable/2235844.

[110] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The New EXT4 Filesystem: Current Status and Future Plans. In *Ottawa Linux symposium*, pages 21–34. The Linux Kernel Archives, 2007. URL: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-21-34.pdf.

[111] Sara McAllister, Benjamin Berg, Daniel S Berger, George Amvrosiadis, Nathan Beckmann, Gregory R Ganger, et al. FairyWREN: A Sustainable Cache for Emerging Write-Read-Erase Flash Interfaces. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 745–764. USENIX Association, 2024. URL: https://www.usenix.org/conference/osdi24/presentation/mcallister.

[112] Meta. Meta 2023 Path to Net Zero. https://sustainability.fb.com/wp-content/uploads/2023/07/Meta-2023-Path-to-Net-Zero.pdf, 2023.

[113] Micron. Micron 9400 NVMe SSD. https://www.micron.com/content/dam/micron/global/public/products/product-flyer/9400-nvme-ssd-product-brief.pdf, 2024.

[114] Microsoft. Microsoft Environmental Sustainability Report. https://blogs.microsoft.com/on-the-issues/2023/05/10/2022-environmental-sustainability-report/, 2023.

[115] Neal R. Mielke, Robert E. Frickey, Ivan Kalastirsky, Minyan Quan, Dmitry Ustinov, and Venkatesh J. Vasudevan. Reliability of Solid-State Drives Based on NAND Flash Memory. In *Proceedings of the IEEE*, pages 1725–1750. IEEE, 2017. URL: https://ieeexplore.ieee.org/document/7998601.

[116] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Conference on File and Storage Technologies (FAST)*, pages 1–12. USENIX Association, 2012. URL: https://www.usenix.org/system/files/conference/fast12/min.pdf.

[117] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. CrashMonkey. https://github.com/utsaslab/crashmonkey/tree/master/code/tests/seq1, 2018.

[118] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–50. USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/mohan.

[119] Sangwhan Moon and A. L. Narasimha Reddy. Does RAID Improve Lifetime of SSD Arrays? In *ACM Transactions on Storage (TOS)*, pages 1–29. ACM, 2016. URL: https://dl.acm.org/doi/10.1145/2902180.

[120] Muthukumar Murugan and David Hung-Chang Du. Rejuvenator: A Static Wear Leveling Algorithm for NAND Flash Memory with Minimized Overhead. In *Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2011. URL: https://ieeexplore.ieee.org/document/5937225.

[121] NeilBrown. Multiple Device Driver. https://github.com/neilbrown/mdadm, 2020.

[122] NetApp. NetApp AFF Solutions. https://www.netapp.com/data-storage/, 2024.

[123] NIMBUSDATA. ExaDrive DC Specifications. https://nimbusdata.com/docs/ExaDrive-DC-Datasheet.pdf, 2024.

[124] NVM Express. NVM Express Base Specification 2.0. https://nvmexpress.org/developers/nvme-specification/, 2021.

[125] NVM Express. NVMe Command Line Interface. https://github.com/linux-nvme/nvme-cli, 2021.

[126] A. O'Hagan and Tom Leonard. Bayes Estimation Subject to Uncertainty about Parameter Constraints. In *Biometrika*, pages 201–203. JSTOR, 1976. URL: https://doi.org/10.1093/biomet/63.1.201.

[127] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *International Conference on Management of Data (SIGMOD)*, pages 343–354. ACM, 2016. URL: https://dl.acm.org/doi/10.1145/2882903.2882910.

[128] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. Improving Performance and Lifetime of the SSD RAID-based Host Cache through A Log-structured Approach. In *Special Interest Group on Operating Systems (SIGOPS)*, pages 90–97. ACM, 2014. URL: https://dl.acm.org/doi/10.1145/2626401.2626419.

[129] Aleph One. Yet Another Flash File System. http://www.yaffs.net.

[130] Open NAND Flash Interface. ONFI 5.0 Spec. http://www.onfi.org/specifications/, 2021.

[131] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P., Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, J. R. Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's Tectonic Filesystem: Efficiency from Exascale. In *Conference on File and Storage Technologies (FAST)*, pages 217–231. USENIX Association, 2021. URL: https://www.usenix.org/conference/fast21/presentation/pan.

[132] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *Annual Technical Conference (ATC)*, pages 47–62. USENIX Association, 2019. URL: https://www.usenix.org/conference/atc19/presentation/panda.

[133] Changhyun Park, Seongjin Lee, Youjip Won, and Soohan Ahn. Practical Implication of Analytical Models for SSD Write Amplification. In *International Conference on Performance Engineering (ICPE)*, pages 257–262. ACM, 2017. URL: https://dl.acm.org/doi/10.1145/3030207.3030219.

[134] Jisung Park, Myungsuk Kim, Myoungjun Chun, Lois Orosa, Jihong Kim, and Onur Mutlu. Reducing Solid-state Drive Read Latency by Optimizing Read-retry. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 702–716. ACM, 2021. URL: https://dl.acm.org/doi/10.1145/3445814.3446719.

[135] Daniel Christopher Powell and Björn Franke. Using Continuous Statistical Machine Learning to Enable High-Speed Performance Prediction in Hybrid Instruction-/Cycle-Accurate Instruction Set Simulators. In *Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 315–324. ACM, 2009. URL: https://doi.org/10.1145/1629435.1629478.

[136] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2005. URL: https://research.cs.wisc.edu/wind/Publications/iron-sosp05.pdf.

[137] PuLP. An Linear and Mixed Integer Programming Modeler. https://coin-or.github.io/pulp/, 2009.

[138] Arvind Rajan. Theory of Linear and Integer Programming. In *Networks*, pages 1–471. Wiley, 1990. URL: https://colab.ws/articles/10.1002%2Fnet.3230200608.

[139] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *European Conference on Computer Systems (EuroSys)*, pages 803–817. ACM, 2024. URL: https://dl.acm.org/doi/10.1145/3627703.3650075.

[140] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Symposium on Operating System Principles (SOSP)*, pages 26–52. ACM, 1991. URL: https://dl.acm.org/doi/10.1145/146941.146943.

[141] Roshan Nair and Arun George. The Promise of NVMe Flexible Data Placement in Data Center Sustainability. https://www.sniadeveloper.org/events/agenda/session/698, 2025.

[142] Hesham Saadawi, Gabriel Wainer, and German Pliego. DEVS Execution Acceleration with Machine Learning. In *Symposium on Theory of Modeling and Simulation (TMS-DEVS)*, pages 1–6. IEEE, 2016. URL: 10.23919/TMS.2016.7918816.

[143] Samsung. Samsung BM1743 QLC V-NAND. https://semiconductor.samsung.com/news-events/tech-blog/next-generation-qlc-v-nand-increases-data-center-profitability/, 2024.

[144] Seagate. BarraCuda 120 SSD Data Sheet. https://www.seagate.com/content/dam/seagate/migrated-assets/www-content/datasheets/pdfs/barracuda-120-sata-DS2022-2-2104US-en_US.pdf, 2021.

[145] Xin Shi, Fei Wu, Shunzhuo Wang, Changsheng Xie, and Zhonghai Lu. Program Error Rate-based Wear Leveling for NAND Flash Memory. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*, pages 1241–1246. IEEE, 2018. URL: https://ieeexplore.ieee.org/document/8342205.

[146] Youngseop Shim, Myungsuk Kim, Myoungjun Chun, Jisung Park, Yoona Kim, and Jihong Kim. Exploiting Process Similarity of 3D Flash Memory for High Performance SSDs. In *International Symposium on Microarchitecture (MICRO)*, pages 211–223. ACM, 2019. URL: https://dl.acm.org/doi/10.1145/3352460.3358311.

[147] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. Disaggregated RAID Storage in Modern Datacenters. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 147–163. ACM, 2023. URL: https://dl.acm.org/doi/10.1145/3582016.3582027.

[148] Keith A. Smith and Margo I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 203–213. ACM, 1997. URL: https://doi.org/10.1145/258612.258689.

[149] SNIA. Solid State Storage (SSS) Performance Test Specification (PTS) Enterprise Version 1.1. https://www.snia.org/sites/default/files/technical-work/pts/release/SNIA-SSS-PTS-Enterprise-v1.1.pdf, 2013.

[150] Seungwoo Son and Jaeho Kim. Differentiated Protection and Hot/Cold-Aware Data Placement Policies through k-Means Clustering Analysis for 3D-NAND SSDs. In *Electronics*, pages 398–412. MDPI, 2022. URL: https://doi.org/10.3390/electronics11030398.

[151] Esther Spanjer and Easen Ho. The Why and How of SSD Performance Benchmarking - SNIA. https://www.snia.org/sites/default/education/tutorials/2011/fall/SolidState/EstherSpanjer_The_Why_How_SSD_Performance_Benchmarking.pdf, 2011.

[152] 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 *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 144–151. ACM, 2021. URL: https://dl.acm.org/doi/10.1145/3458336.3465300.

[153] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. LeaFTL: A Learning-based Flash Translation Layer for Solid-state Drives. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 442–456. ACM, 2023. URL: https://jianh.web.engr.illinois.edu/papers/leaftl-asplos2023.pdf.

[154] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. Learned index: A Comprehensive Experimental Evaluation. In *Proceedings of the VLDB Endowment (PVLDB)*, pages 1992–2004. ACM, 2023. URL: https://dl.acm.org/doi/10.14778/3594512.3594528.

[155] Swamit Tannu and Prashant J Nair. The Dirty Secret of SSDs: Embodied Carbon. In *SIGENERGY Energy Informatics Review (EIR)*, pages 4–9. ACM, 2023. URL: https://dl.acm.org/doi/10.1145/3630614.3630616.

[156] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *Conference on File and Storage Technologies (FAST)*, pages 49–65. USENIX Association, 2018. URL: https://www.usenix.org/conference/fast18/presentation/tavakkol.

[157] Yentl Van Tendeloo and Hans Vangheluwe. Discrete Event System Specification Modeling and Simulation. In *Winter Simulation Conference (WSC) 2018*, pages 162–176. IEEE, 2018. URL: https://ieeexplore.ieee.org/document/8632372.

[158] The kernel development community. Linux Device Mapper. https://docs.kernel.org/admin-guide/device-mapper/index.html, 2024.

[159] The SSD Guy. Comparing Wear Figures on SSDs. https://thessdguy.com/comparing-wear-figures-on-ssds/, 2017.

[160] Theodore Ts'o. Ext2/3/4 File System Utilities. https://e2fsprogs.sourceforge.net/, 2009.

[161] Vasily Tar Asov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf, 2016.

[162] Haobo Wang. Optimizing Flash-Based Storage Systems, 2018. URL: https://escholarship.org/uc/item/4h29w76.

[163] Shengzhe Wang, Zihang Lin, Suzhen Wu, Hong Jiang, Jie Zhang, and Bo Mao. LearnedFTL: A Learning-based Page-level FTL for Improving Random Reads in Flash-based SSDs. In *Symposium on High-Performance Computer Architecture (HPCA)*, pages 616–629. IEEE, 2024. URL: https://arxiv.org/abs/2303.13226.

[164] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, and Yuanyuan Dong. StRAID: Stripe-threaded Architecture for Parity-based RAIDs with Ultra-fast SSDs. In *Annual Technical Conference (ATC)*, pages 915–932. USENIX Association, 2022. URL: https://www.usenix.org/conference/atc22/presentation/wang-shucheng.

[165] Western Digital. Zoned Namespaces (ZNS) SSDs. https://zonedstorage.io/introduction/zns/, 2020.

[166] Ellis Herbert Wilson, Myoungsoo Jung, and Mahmut T. Kandemir. ZombieNAND: Resurrecting Dead NAND Flash for Improved SSD Longevity. In *Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 229–238. IEEE, 2014. URL: https://ieeexplore.ieee.org/document/6888765.

[167] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Baleen:ML Admission & Prefetching for Flash Caches. In *Conference on File and Storage Technologies (FAST)*, pages 347–371. USENIX Association, 2024. URL: https://www.usenix.org/conference/fast24/presentation/wong.

[168] Michael Wu and Willy Zwaenepoel. ENVy: A Non-Volatile, Main Memory Storage System. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 86–97. ACM, 1994. URL: https://doi.org/10.1145/195473.195506.

[169] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *Annual Technical Conference (ATC)*, pages 817–833. USENIX Association, 2024. URL: https://www.usenix.org/conference/atc24/presentation/xu-dong.

[170] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures. In *Annual Technical Conference (ATC)*, pages 961–976. USENIX Association, 2019. URL: https://www.usenix.org/conference/atc19/presentation/xu.

[171] Fan Xu, Shujie Han, Patrick P. C. Lee, Yi Liu, Cheng He, and Jiongzhou Liu. General Feature Selection for Failure Prediction in Large-scale SSD Deployment. In *International Conference on Dependable Systems and Networks (DSN)*, pages 263–270. IEEE, 2021. URL: https://ieeexplore.ieee.org/document/9505157.

[172] Gala Yadgar, Moshe Gabel, Shehbaz Jaffer, and Bianca Schroeder. SSD-based Workload Characteristics and Their Performance Implications. In *ACM Transactions on Storage (TOS)*, pages 1–26. ACM, 2021. URL: https://dl.acm.org/doi/10.1145/3423137.

[173] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Conference on File and Storage Technologies (FAST)*, pages 15–28. USENIX Association, 2017. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan.

[174] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. AutoStream: Automatic Stream Management for Multi-streamed SSDs. In *International Systems and Storage Conference (SYSTOR)*, pages 1–11. ACM, 2017. URL: https://dl.acm.org/doi/10.1145/3078468.3078469.

[175] Juncheng Yang, Yao Yue, and K. V. Rashmi. A lLrge Scale Analysis of Hundreds of In-memory Cache Clusters at Twitter. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 191–208. USENIX Association, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/yang.

[176] Yudong Yang, Vishal Misra, and Dan Rubenstein. On the Optimality of Greedy Garbage Collection for SSDs. In *SIGMETRICS Performance Evaluation Review*, pages 63–65. ACM, 2015. URL: https://dl.acm.org/doi/10.1145/2825236.2825261.

[177] Shushu Yi, Xiurui Pan, Qiao Li, Qiang Li, Chenxi Wang, Bo Mao, Myoungsoo Jung, and Jie Zhang. ScalaAFA: Constructing User-Space All-Flash Array Engine with Holistic Designs. In *Annual Technical Conference (ATC)*, pages 141–156. USENIX Association, 2024. URL: https://www.usenix.org/conference/atc24/presentation/yi-shushu.

[178] Kong-Kiat Yong and Li-Pin Chang. Error Diluting: Exploiting 3-D NAND Flash Process Variation for Efficient Read on LDPC-Based SSDs. In *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pages 3467–3478. IEEE, 2020. URL: https://dl.acm.org/doi/10.1145/3078468.3078469.

[179] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. RAID+: Deterministic and Balanced Data Distribution for Large Disk Enclosures. In *Conference on File and Storage Technologies (FAST)*, pages 279–294. USENIX Association, 2018. URL: https://www.usenix.org/conference/fast18/presentation/zhang.

[180] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *Annual Technical Conference (ATC)*, pages 87–100. USENIX Association, 2016. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/zhang.

[181] Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao. Optimizing Deterministic Garbage Collection in NAND Flash Storage Systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 14–23. IEEE, 2015. URL: https://ieeexplore.ieee.org/document/7108392.

[182] Tao Zhang, Aviad Zuck, Donald E. Porter, and Dan Tsafrir. Flash Drive Lifespan is a Problem. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 42–49. ACM, 2017. URL: https://dl.acm.org/doi/10.1145/3102980.3102988.

[183] Yuqi Zhang, Wenwen Hao, Ben Niu, Kangkang Liu, Shuyang Wang, Na Liu, Xing He, Yongwong Gwon, and Chankyu Koh. Multi-view Feature-based SSD Failure Prediction: What, When, and Why. In *Conference on File and Storage Technologies (FAST)*, pages 409–424. USENIX Association, 2023. URL: https://www.usenix.org/conference/fast23/presentation/zhang.

[184] Hao Zhou, Zhiheng Niu, Gang Wang, Xiaoguang Liu, Dongshi Liu, Bingnan Kang, Hu Zheng, and Yong Zhang. A Proactive Failure Tolerant Mechanism for SSDs Storage Systems based on Unsupervised Learning. In *International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2021. URL: https://ieeexplore.ieee.org/document/9521302.

[185] 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 *Conference on File and Storage Technologies (FAST)*, pages 187–202. USENIX Association, 2021. URL: https://www.usenix.org/conference/fast21/presentation/zhou.

[186] Aviad Zuck, Donald Porter, and Dan Tsafrir. Degrading Data to Save the Planet. In *Workshop on Hot Topics in Operating Systems (HotOS)*, pages 61–69. ACM, 2023. URL: https://dl.acm.org/doi/10.1145/3593856.3595896.

VITA

Ziyang Jiao was born in 1998 in China. He received his Bachelor of Engineering degree in 2019. He pursued a Master of Science in Computer Science at Washington University in St. Louis from 2019 to 2020. In 2020, he began his doctoral studies in Computer and Information Science and Engineering at Syracuse University. This dissertation was defended in April 2025 at Syracuse University.

Ziyang's research focuses on file and storage systems, solid-state drives, operating systems, green computing, and sustainability. His research philosophy is to imbue minimal yet meaningful knowledge into various layers of the I/O stack, thereby enabling a more efficient, synergistic, and adaptive storage ecosystem.

During his doctoral studies, Ziyang's work has been published at top-tier systems venues, including the USENIX Conference on File and Storage Technologies (FAST) and the ACM Workshop on Hot Topics in Storage and File Systems (HotStorage), where he received the Best Paper Award in 2022.

In addition to his research contributions, Ziyang has demonstrated a strong commitment to academic service and mentorship. He has served on the Artifact Evaluation Committee for FAST 2024 and the Shadow Program Committee for EuroSys 2025. He has also worked as a teaching assistant for both undergraduate and graduate-level courses, including Computer Organization and Programming Systems and Introduction to Machine Learning, supporting student learning in areas such as digital logic, instruction set architecture, memory systems, and supervised learning techniques.

Ziyang's research has been recognized with multiple honors, including the Best Paper Award in HotStorage, the ECS Research Day Honorable Award, and a Ph.D. Fellowship from SU.