Improving Reliability and Performance of Storage Stacks on next generation Solid State Drives

by

Shehbaz Jaffer

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy

Graduate Department of Computer Science
University of Toronto

Improving Reliability and Performance of Storage Stacks on next generation Solid State Drives

Shehbaz Jaffer
Doctor of Philosophy

Graduate Department of Computer Science
University of Toronto
2023

Abstract

Solid State Drives (SSDs) are increasingly replacing HDDs as the main source of a persistent medium in both datacenters and personal devices. SSDs offer higher performance and consume less power. They also bring two challenges in the storage space.

First, SSDs have different reliability characteristics than HDDs. Unlike HDDs that are based on magnetic media, SSDs store data in the form of an electronic charge. This significantly increases bit-level errors in SSDs. However, there exists no expansive study of the impact of such errors on the overlying software stack. Furthermore, as the demand for high capacity SSDs is increasing, more bits are being stored within the same voltage range of the SSD medium. This further reduces SSD drive reliability as there is a lower margin of error for each data bit written to the drive.

Second, with the end of Moore's law, there is increased emphasis on disaggregated computing where device-specific computation takes place on a processor within the device. To achieve this goal in storage, SSDs are embedded with CPUs. Applications can offload computation from the host to the device and save host CPU cycles, lower host-device latency, and reduce bandwidth utilization. However, such devices are expensive, hard to program, and may not be suitable for all application computation offloads.

We address these reliability and performance challenges for SSDs as follows:

First, we evaluate file system reliability in the presence of SSD errors. File Systems are the main component of the storage software stack which interacts with the underlying storage media. We notice that real-world file systems often make performance assumptions for SSDs that impact their reliability. We find that real world file systems incur severe failures such as unmountable drives in 16% of our extensive test suite of over 7000 file system experiments.

Second, we invent a novel voltage-based Write Once Memory (WOM-v) coding scheme that reduces the number of erase operations for every overwrite on a SSD. This directly improves the SSD reliability for high-density SSDs where only a limited number of program-erase operations are possible. We show that for QLC SSD, we can reduce the number of erase operations by $4.4x - 11.1x$ on real-world workloads.

Third, we build Nescafe, a computational storage framework to evaluate performance gains of offloading computational tasks to an emulated near storage device. We show that real-world, compute-intensive applications like zlib compression, jbd2 checksum computation and LSM Tree compaction are easy to offload to a computational device using our framework. We also show that surprisingly, not all computationally intensive applications perform better by offloading tasks to an on-disk processor. We show that Nescafe is generic and simple to extend to other block, stream and application based workloads.

It takes an entire village to raise a child, and an entire University to graduate a Ph.D. student

Ancient Proverb

Acknowledgements

I am grateful to my advisor, Prof Bianca Schroeder for taking on the enduring task of advising me during my Ph.D. and supporting me throughout this memorable journey. Her enthusiasm for teaching and guiding, and her unwavering support towards research ideas on both the good and not-so-good days motivated me to give my best to this thesis. I would like to thank her for giving me the freedom to explore multiple research directions under her close guidance and for having in-depth discussions on numerous ideas over the past several years. Some of these ideas made it to a paper, many didn't, but Bianca made sure that there was learning in-between. Thanks for everything, Bianca.

I would like to thank Prof Angela Demke Brown for talking to me during the VEE 2015 conference and taking the time to review my Ph.D. application. Angela has continued to mentor me throughout my Ph.D. program in various roles - as a co-advisor, as DCS Graduate Chair, and finally as the Char of my thesis program committee. Her insightful comments and recommendations based on her expertise in storage and file-systems has significantly improved the contents of this thesis.

I would like to thank Prof Gennady Pekhimenko for sharing his wisdom from the architecture research community during all my Ph.D. checkpoint meetings. I am grateful to him for mentioning near-storage computation during one of my earlier Ph.D. checkpoints. His feedback in this area helped develop the near storage compute project into a significant part of my thesis.

I would like to extend my sincere gratitude towards my extended committee members, Prof. Raju Rangaswami and Prof. Ding Yuan for reading my thesis in detail and providing their feedback which further enhanced the quality of this thesis.

A substantial amount of learning during my Ph.D. came from fellow Ph.D.s and Postdocs in Bianca's group. I would like to thank Prof Kaveh Mahdaviani for introducing me to the wonderful world of coding theory. I consider him as a shadow co-advisor for Chapter 3 of this thesis. I would like to thank Andy Hwang for being an excellent mentor and friend and helping me debug various kernel-level crashes. I am fortunate to have Stathis Maneas as a great colleague, project partner, and a very helpful friend since the first year of my PhD. Both Stathis and Andy made me feel welcome in Bianca's group.

I would like to thank my external research collaborators, Dr Amy Tai and Dr Michael Wei for hosting me during my (virtual) internship at VMWare research. I am also grateful to Prof Gala Yadgar and Prof Moshe Gabel for their collaboration focused on characterizing workload performance on SSDs.

I would like to thank many Master's and undergraduate students who visited Bianca's group and collaborated and extended my work. Specifically, I would like to thank Brain

Chan, Tony Xiao, Jialun Lyu, Yuhan Shao and Fengjia Zhang for their contributions to the SSD workload characterization project and Charles Xu for working on the WOM code reliability project. I hope they learned from me as much as I learned from them.

I would like to thank Prof Graeme Hirst, Lynda Barnes, Marina Haloulos, Rose Marie Scott, Steve D'Silva, Pavi Chandrasegaram, Joseph Raghubar and many other members of the Department of Computer Science Graduate Program administrative staff who seamlessly run the graduate program. I am also grateful to Tom Glinos for his support in maintaining and troubleshooting the swift and mel clusters, especially on weekends and during peak pandemic times.

The University of Toronto provides numerous resources to its students for skill development and networking outside the general academic curriculum. I would like to thank The University of Toronto Graduate Student Union, The University of Toronto Karate Club, The Indian Graduate Student Association and University of Toronto Entrepreneurship Cell for providing great means to socialize, organize and participate in events within the University.

I would like to thank my current and former labmates: Mike (Dai) Qin, Kuei (Jack) Sun, Jack (Yu) Luo, Arnamoy Bhattacharrya, Ali Jokhar, Alexey Khrabrov, Shiva Ketabi, Bojian Zheng, James Gleeson and Christina Christodoulakis. Outside of the lab, I would like to thank Umang Yadav, Aamod Kore, Anand Mohan, Aditya Patel, Prabhjot Singh Sandhu, Bipul Islam, Tirthankar Mitra, Ankit Jain, Akriti Kaur, Phalguni Ghonge, Jay Patel, Maharishi Trivedi and Sandeep Sony. I am grateful to Rizwan Oskoui for motivating me to work through the final stages of my thesis.

Finally, I would like to thank my family, my sister Ayesha Ana, my brother Shafi Babar and sister-in-law Saba Sayeed, and my parents Badar Afroz Aliya and Colonel Ghazanfar Hussain for their love, encouragement and support for everything that I have done in my life so far. I am who I am today because of them – I dedicate this thesis to them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This Thesis focuses on improving reliability and performance of storage stacks on next generation of Solid State Drives (SSDs). SSDs are evolving in two different ways: First, to meet the high-capacity demands of data storage, SSDs are increasingly becoming denser. There is increased interest in storing multiple bits within a cell, the smallest unit of storage in an SSD. This has given rise to QLC and PLC drives that store 4 and 5 bits per cell respectively. The relatively higher performance and lower power consumption of such SSDs pose them as excellent candidates to replace high density Hard Disk Drives (HDDs) as the primary persistent storage media in the capacity tier.

Second, with the end of Moore's law, there is increased interest in heterogeneous computing, where computation is being done within specialized devices instead of the host. To meet this need, SSDs are being provisioned with embedded processors to perform computing within the storage device. Such SSDs are called Computational Storage Disks or CSDs.

Both advancements bring significant challenges to the reliability and performance of storage software that is run on top of the SSDs. First, with a denser SSD media, there are increased chances of device generating media errors due to wear out, retention loss and read disturbance. This reduces SSD media reliability due to increased media errors such as I/O errors and uncorrectable bit corruption errors. With higher SSD media density where more bits are stored in each cell, there is also a drastic reduction in the number of times data can be programmed and erased reliably over time. This is because the voltage range allocated to each bit reduces as we pack more bits within the same cell voltage range. Hence with increased SSD capacity, the reliability of persistent media is of heightened importance. Second, there is increased interest in identifying computation heavy applications that can be offloaded to a SSD with an on-disk CPU. Existing storage applications have varied amount of computation heavy components. Therefore, not all applications benefit from Computational Storage Devices.

We explore and improve the reliability of SSDs from two perspectives: First, we evaluate the resiliency and identify weaknesses of file systems designed and optimized for

flash. Second, we reduce erase operations done on SSDs which improves the lifetime of high-density SSD media. In the third part of the thesis, we build a framework to assess whether a storage application could benefit from using computational storage disks.

1.1 Thesis Outline

1.1.1 File System Reliability on SSDs

We study the reliability of file systems designed and optimized for flash on top of Solid State Drives. File Systems are a critical component of the storage software stack and provide an interface that the end user applications depend on for accessing correct data stored on the underlying media. We evaluate three contemporary file systems - ext4, BtrFS and F2FS that run on SSDs against a wide range of flash media errors. We answer three main questions during our study:

1. Can file systems **detect** media errors?
2. Can file systems **recover** from media errors?
3. How robust are file system checkers (**fsck**) in error recovery?

We propose a taxonomy of error detection and correction for the file systems, a framework for injecting errors into the file systems and observing their behaviour, a rich testbed for file system testing, and key results of our evaluation for each of the three file systems and their corresponding file system checkers.

1.1.2 Improving Reliability of High Density SSDs using WOM Codes

Write Once Memory (WOM) coding is an encoding scheme in which input raw data bits can be encoded into output encoded bits such that the encoded bits only increase from 0 to 1 with each subsequent device write. Although this bit-based model works on SSDs that are modelled as groups of bits (eg. SLC drives), they fail to take into account the voltage based format in which multi-bit SSDs store their data. We propose a new WOM (Write Once Memory) coding model for high density SSDs to reduce erase operations on the device and achieve higher flash reliability. In particular, we propose a novel WOM-v code that treats the underlying SSD media as a storage unit containing voltage based charge as opposed to being treated as group of bits. Next, we perform a theoretical evaluation of the trade-off between P/E cycle gains and encoding space overheads using WOM-v codes on high density SSDs. Further, we implement WOM-v code in a state of the art flash simulator (FEMU) and evaluate practical gains of WOM-v codes on real world workloads. Finally, we provide performance overheads and optimizations for using WOM-v codes.

1.1.3 A framework for evaluating Near-Storage Computation

With the end of Moore's law [175], there is an increased interest in heterogeneous computing. This has lead to creating specialized devices that have embedded processors for different components of a computing system. For storage, a new class of SSDs called Computational Storage Disks (CSDs) have recently emerged for application computation offloading. CSDs are expensive SSDs that involve low level hardware programming to realize application computation offload. Porting storage applications to computational storage is expensive, time-consuming, and cumbersome. Moreover, after a computation is offloaded to the device, it may not always result in performance gains for all applications. Hence, it would be prudent to evaluate whether application computation offload is beneficial before investing time, effort and resources in buying and offloading an application's computation to a CSD. To address this issue, we build a software-only computational storage framework that enables offloading data intensive computation tasks from the application running in the host to the underlying emulated SSD device. Our framework helps in dividing application between host and device without doing low-level hardware programming. We are able to also evaluate the processor speed, interconnect latency and interconnect bandwidth required to gain any advantages from application computation offload. This helps us determine the type of CSD that we will need to gain any type of application computation speedup without acquiring and programming real hardware. First, we describe the design and implementation of our framework. Next, we describe a theoretical model and show how it has limitations. Finally we show the usefulness of our simulator by offloading real world applications - zlib compression library, ext4 journal checksumming, LSM Tree Compaction and Spark and describe the scenarios in which speedup can be achieved with CSDs.

1.2 Thesis Contributions

In this thesis, we address both the reliability and the performance challenges that the next generation of SSDs bring to the storage stack. To study the reliability of the storage stack in the presence of SSDs that have a relatively higher number of partial errors than HDDs, we evaluate the overlying file system's capability to detect and recover from underlying SSD errors. In order to improve the SSD reliability for denser SSD drives which have lower resilience to erase operations, we propose an encoding mechanism that enables overwriting old data without performing an erase operation. Finally, to evaluate if computationally intensive tasks can leverage high-performance Computational Storage Disks, we build a framework to evaluate such applications in software without buying additional hardware resources or writing tedious, low-level hardware programs.

In the next sub-sections, we elaborate on the contributions of the thesis from the perspective of improving reliability and performance of storage stacks on next generation of SSDs.

1.2.1 Improving Reliability of the Storage Stack

A storage stack contains one or more user-space applications such as key-value stores or databases that run on top of file systems residing inside the Operating System. It is the file system that interacts with the underlying block device such as the SSD hardware. We analyse how contemporary file systems, which are the first line of defense against any underlying errors in the storage media, protect the application from the data that is being read from or written to the SSDs. In this thesis, we provide a detailed analysis and categorization of SSD-based errors that have been reported in large-scale data centers. We provide a mapping of the errors that emanate from the underlying SSDs to the form in which they would appear at the file system level. We notice that all types of SSD errors reported can be mapped to 5 categories of errors at the block layer. We create a file system error injection tool to inject these 5 errors in the 3 file systems of our study - namely btrfs, ext4 and f2fs file system. We build a rich taxonomy of error detection, error correction techniques and the effectiveness of the *fsck* tool for each file system in recovering from a particular error. We created multiple user space applications exercising multiple file system code paths and ran over 7000 experiments. Overall, we create a rich suite of 7000 experiments that inject various forms of errors in the file system. We were able to uncover severe file system errors when injecting different categories of SSD errors for 16% of the experiments. This finding demonstrates that real-world, production file systems make reliability assumptions while transitioning from HDD media to SSD media without taking the SSD reliability constraints into account. We also open source our generic error-injection framework which consists of a rich suite of tests that can be applied to other file systems. Furthermore, additional category of errors such as timing errors and power errors can be easily incorporated to our framework to do more rigorous testing of a file system under other categories of errors. Our framework enables a test-driven development of the file system that can be run on future versions of file systems with new features to check their resiliency to SSD-based errors.

We further look at the SSD hardware reliability from the perspective of the number of writes it can endure. Futuristic, higher density SSDs have very low Program-and-Erase cycles (P/E cycles). To increase the lifetime of SSDs, we propose a novel Write Once Memory coding scheme that encodes and overwrites data without erasing previous data. Our coding scheme takes the voltage-based nature of the cell within an SSD into account. This is different from previously existing Write-Once-Memory coding schemes that take a binary approach to encoding and writing data on the underlying device. We perform 3 novel optimizations for WOM-v coding scheme in the context of dense media - code word sharing, same generation transition, and programming beyond maximum generations. This enables writing up to 500% additional data, in-theory, to the underlying QLC device before an erase operation is required. This directly improves the reliability of the underlying flash drive and improves the lifetime of SSDs. We further implement and evaluate our code in the state of the art flash simulator, and demonstrate practical reliability

gains from the coding scheme. This is the first non-binary WOM code that we know of that has been implemented for high-density SSDs. Further, we have demonstrated reliability gains of $4.4x - 11.1x$ for real world traces from datacenters including Alibaba and Microsoft using WOM-v codes on QLC drives. We describe the design of our coding scheme and its implementation and evaluation in more detail in Chapter 3.

1.2.2 Improving Performance of the Storage Stack

Advanced storage systems such as file systems, volume managers and key-value stores run a number of compute intensive applications such as checksumming, compression, deduplication, encryption and snapshotting during data access to the underlying storage device. Such computation intensive tasks can often be offloaded to the underlying storage device such as Computational Storage Disks (CSDs) that has a on-disk processor near the data. Such disks are currently expensive, hard to program and and may not always lead to improvement in storage application performance.

In this thesis, we build Nescafe, a Near Storage Computation framework to evaluate if applications that might benefit from offloading such computation intensive tasks to the on-disk CPU and actually get performance gains. Our framework is written in C, does not involve writing tedious, low-level hardware code, and runs entirely in software without the need to buy expensive hardware. We build our framework on FEMU, the state of the art flash simulator and provide API's to extend block-based, stream-based and application-based storage workloads to the device. Our framework uses the NVM-e interface to send computational instructions to the underlying storage device. We build an analytical framework to evaluate simple storage tasks and show that the results obtained on Nescafe match the analytical framework results. We then port three real world computationally intensive applications - *zlib* compression, *ext4* journaling checksum computation and *LSM-Tree compaction* to demonstrate practical gains of offloading computation intensive tasks for such applications on the underlying storage device. Our framework is generic and can be easily extended to evaluate other block, stream and application based workloads. We describe our contributions to evaluate and improve the performance of storage stacks using computational storage devices in more detail in Chapter 4.

1.2.3 Research Artifacts

This thesis has resulted in three research artifacts that form a basis for further research in the area of improving reliability and performance of storage stacks on next generation of SSDs:

1. We build `dm-inject` [45], a framework to inject block-layer errors in file-systems. We have used the framework to inject various SSD errors in ext4, f2fs and btrfs file systems. Our framework can be extended to other file systems and error types.

2. We build `WOM-v coder` [181], an extension of lightNVM module that encodes and decodes data while writing to and reading from the underlying SSD device. The encoding scheme is specified using simple lookup tables that can be extended to support denser SSD devices and more sophisticated coding schemes. We also extend FEMU to support TLC and QLC devices [145] which is already part of the FEMU mainline code.

3. We build `Nescafe` [126], a framework to evaluate performance gains in applications when computation is offloaded to an on-disk drive. Our framework provides a library of NVMe directives for commonly used storage based computation tasks. The library can be extended to add customised computation for the data being stored or retrieved from the device.

Chapter 2

Evaluating File System Reliability on SSDs

It's not the prevention of bugs but the recovery, the ability to gracefully exterminate them, that counts

Victoria Livschitz

2.1 Introduction

SSD reliability depends on both the reliability of the devices as well as the ability of the file system above them to handle errors. File system reliability has been studied in the past in the context of HDD errors [143]. However, there are multiple reasons why the study should be revisited.

First, the failure characteristics of SSDs differ significantly from those of HDDs. For example, recent field studies [114, 124, 164] show that while their replacement rates (due to suspected hardware problems) are often by an order of magnitude lower than those of HDDs, the occurrence of partial drive failures that lead to errors when reading or writing a block or corrupted data can be an order of magnitude higher. Other work argues that the Flash Translation Layer (FTL) of SSDs might be more prone to bugs compared to HDD firmware. This is due to their high complexity and less maturity, and demonstrate this to be the case when drives are faced with power faults [202]. This makes it even more important than before that file systems can detect and deal with device faults effectively.

Second, file systems have evolved significantly since [143] was published 16 years ago; the ext family of file systems has undergone major changes from the ext3 version considered in [143] to the current ext4 [113]. New players with advanced file-system features have arrived. Most notably Btrfs [154], a copy-on-write file system which is more suitable for SSDs with no in-place writes, has garnered wide adoption. The design of Btrfs is particularly interesting as it has fewer total writes than ext4's journaling mechanism. Further, there are new file systems that have been designed specifically for flash, such as F2FS [97], which follow a log-structured approach to optimize performance on flash.

In this chapter, we characterize the resilience of modern file systems running on flash-based SSDs in the face of SSD faults. We also evaluate the the effectiveness of their recovery mechanisms when taking SSD failure characteristics into account. We focus on three different file systems: Btrfs, ext4, and F2FS. ext4 is the most commonly used Linux file system. Btrfs and F2FS include features particularly attractive with respect to flash, with F2FS being tailored for flash. Moreover, these three file systems cover three different points in the design spectrum, ranging from journaling to copy-on-write to log-structured approaches.

2.2 Contributions

The main contribution of this work is a detailed study, spanning three very different file systems and their ability to detect and recover from SSD faults, based on error injection targeting all key data structures. We observe huge differences across file systems and describe the vulnerabilities of each in detail. Over the course of this work we experiment with more than seven thousand fault scenarios and observe that around 16% of them result in severe failure cases (kernel panic, unmountable file system). We make a number of observations and file several bug reports, some of which have already resulted in patches. For our error injection experiments, we developed an error injection module on top of the Linux device mapper framework. The error injection module is open-sourced and is available here [62].

The remainder of this chapter is organized as follows: Section 2.4 provides a taxonomy of SSD faults. We then describe the experimental setup we use to emulate these faults and test the reaction of the three file systems. Section 2.5 presents the results from our fault emulation experiments. Section 2.6 summarizes our observations and insights. We discuss limitations and the impact of our work in Section 2.7.

2.3 Background

In this section, we describe key concepts required to understand the rest of the thesis. Figure 2.1 shows the Host and SSD interface. The `host` consists of multiple applications running as processes. the process reads and writes to a device using a *file system*.

2.3.1 Write Workflow

The writes to the device may be kept in memory in the *page cache*, or sent directly to the block layer. When the application calls *sync*, data and metadata cached in the page cache are written to the underlying block layer. The block layer receives data at the grannularity of a *block*. A typical block size is `4KB`. Each block is then sent by the device to the underlying SSD device. We describe the SSD architecture in more detail in 2.3.4.

2.3.2 Read Workflow

A read request to the disk is sent by the application to the *file system*. If the requested block is available in the *page cache*, it is returned to the application. If the requested block is not available, the read request is sent to the device driver. The device driver issues a device and interface specific read request. The block retured from the device is returned to the application.

We describe file systems in more detail in Section 2.3.3, SSDs in Section 2.3.4.

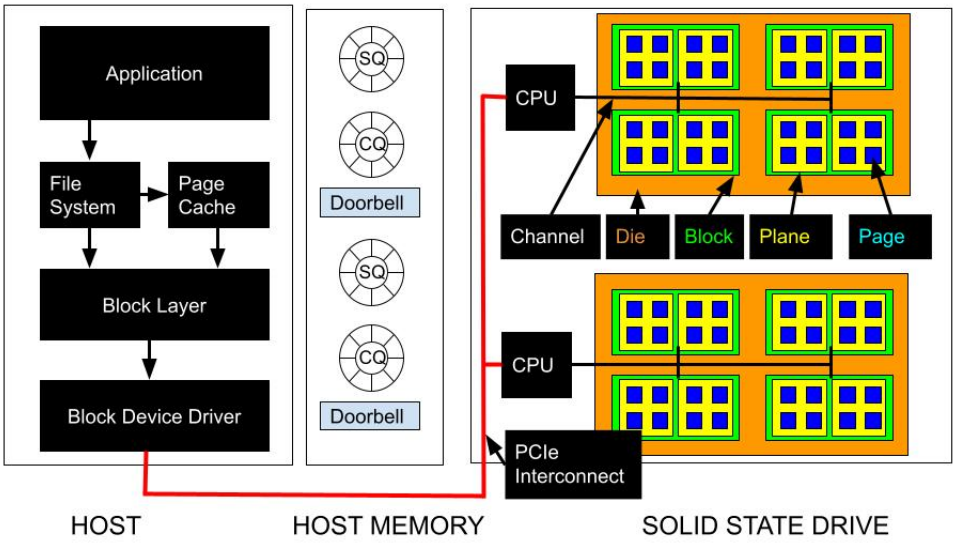


Figure 2.1: *Host and SSD Interface. The application, filesystem and device driver run inside the host. An interconnect (PCIe) connects the host to the SSD. Modern SSDs use NVMe protocol to communicate between host and SSD. Internally, the SSD contains a CPU, memory (not shown) and multiple NAND chips where data is persistently stored in the form of electronic charge. Different voltage levels correspond to different data bits.*

2.3.3 File Systems

A file system is a Operating System component that manages data stored on the underlying device. It is responsible for providing the abstraction of a file to applications. Applications can read and write files without managing the underlying data storage. Files are logically contiguous even when physically stored at non-contiguous locations. All applications using a file system use the same file-system interface. This interface is called the virtual file system (VFS) interface [92]. This helps applications to be agnostic to the type of underlying file-system. All file systems usually come with user-level tools to format the device (`mkfs`), check file system integrity (`fsck`) and inspect file system content (`fsdump`).

In this thesis, we use three file system designs. Each file-system provides different features, benefits and challenges. They also use different on-disk and in-memory data structures.

Journaling File Systems

A journaling file system maintains auxillary space in addition to the main file system. This space is called a journal [142], a circular log to store intermediate file system meta-data. The journal helps the file system run fast, delay updates to the main file system, and recover from failures on the event of a file system crash. When a file is updated by the application, both its data and metadata are updated in memory. When the application calls `sync`, the in-memory data is first written to the device. The in-memory metadata is then written to the journal. Finally, the in-memory metadata is written to the main file system. Since meta-data is written to the journal sequentially, it provides higher performance than updating main-filesystem metadata that is physically at non-contigouous locations. The main-filesystem metadata can be updated asynchronously. On the event of a file system crash before the metadata is written to the main-file system, the journal can be read on reboot, and its valid contents can be selectively written to the main file system.

Copy-On-Write File Systems

A copy-on-write [155] file system creates a new copy of file system data and metadata blocks on each overwrite. The original contents of the file remain in the file system. This helps in creating different versions of the main file system. A user may be able to naturally create different point-in-time instances of the file system, until the older data is removed. This helps the file system to provide two important features: `snapshot` and `cloning`. A user can create a file system snapshot containing all actively used blocks in the file system at that time. This creates a reference to each block, and marks them as non-overwritable for the future, unless the snapshot is deleted. `cloning` creates a identical copy of the original file system. A copy-on-write file system can also create file system clones very efficiently. the original and cloned volume of the file system share all referenced blocks until a change in any block is made.

Log-Structured File Systems

A log structured file system [157] writes all modifications to disk sequentially in a log-like structure. This increases performance during both file writing and crash recovery. The log is the only structure on disk. It contains indexing information so that files can be read back from the log efficiently. Both HDDs and SSDs perform well on sequential writes. Hence Log-Structured File systems are a natural fit on top of SSDs for high performance.

2.3.4 Solid State Drives

SSDs are persistent data storage devices that store data in the form of electronic charge. The smallest unit of storage inside a SSD is called a *cell* Multiple *cells* form a *page*. Pages are often arranged in multiple *planes*. A group of *pages* form a flash *block*. A group of flash *blocks* form a *die*. A typical SSD contains 8-10 *dies*.

2.3.5 Reliability of File Systems

File System reliability has been studied in depth by Prabhakaran et *al.* [143]. The study in [143] was focused on HDDs and covered ext3, ReiserFS and JFS. These file systems were a great fit for HDDs and were extensively used at that time. However over the past decade, a number of new file systems with enhanced features such as journaling (Section 2.3.3), copy-on-write (Section 2.3.3) and log-structure (Section 2.3.3) have been introduced. They mark a significant departure in terms of design principles compared to the systems in [143]. Moreover [143] does not evaluate the reliability of the file system checker or *fsck*. A *fsck* is usually the last line of defense for all file systems against media errors. An analysis of how resilient *fsck* is to media errors is an important study that has not been covered in [143].

Gatla et *al.* [61] study the behavior of file system checkers under faults that forcefully interrupt the repair procedure. They developed a customized fault injection tool, which was built using the Linux SCSI target framework [57], in order to systematically inject faults to the repair procedure. Their results indicate that running the file system's checker after an interrupted repair may not always return the file system to a valid state.

Gunawi et *al.* [67] make use of static analysis to explore how file systems and storage device drivers propagate error codes. Their results indicate that write errors are neglected more often than read errors. In [178], the authors conduct a performance evaluation on the transaction processing system between ext2 and NILFS2. In [121], the authors explore how existing file systems developed for different operating systems behave with respect to features such as crash resilience and performance.

Recently, two new studies presented their reliability analysis of file systems in a context other than the local file system. Ganesan et *al.* [60] present their analysis on how modern *distributed storage systems* behave in the presence of file-system faults. Their fault model includes read and write errors, along with data corruption. Their results indicate that distributed storage systems do not make use of redundancy across replicas to recover from local file system faults. User programs can experience data loss, corruption, and unavailability due to a single local file system fault, highlighting the need to carefully test these systems for all types of faults.

Cao et *al.* [31] present their study on the reliability of *high-performance parallel systems*. The authors introduce a general framework for analyzing the failure policies of parallel file systems and explore the effect of whole device failures, inconsistent global state, and network partitioning. Their results indicate that widely used parallel file systems, such as Lustre are not able to recover from I/O errors, resulting in system hangs and reboot operations. In our work, we focus on local file systems deployed on top of a single SSD and explore their resiliency characteristics against faults related to SSDs.

Prior work has explored techniques to detect inconsistencies within file systems either during runtime using invariant checking [37, 54, 94] or offline using a disk pointer corruption analysis [10]. Both approaches protect file systems against issues that can be

detected by the specified rules only. In [120, 112], the authors test the resilience of file systems against crash faults or power loss. The file systems considered in this study are more recent, modern file systems (ext4, xfs, F2FS, btrfs). This study does not consider bit level corruptions that may occur during reads or writes.

Different techniques involving hardware or modifications inside the FTL have been proposed to mitigate existing errors inside SSDs [36, 137, 26, 25, 105, 109, 107]. These techniques focus on improving SSD reliability using stronger Error Correction Codes (ECC) or employing metadata redundancy.

2.4 File System Error Injection

We emulate different types of SSD failures and check the ability of different file systems to detect and recover from them. We limit our analysis to a local file system running on top of a single drive. Although multi-drive redundancy mechanisms like RAID exist, they are not general substitutes for file system reliability mechanisms. First, RAID is not applicable to all scenarios, such as single drives on personal computers. Second, errors or data corruption can originate from higher levels in the storage stack, which RAID can neither detect nor recover.

Numerous studies published over the last few years have identified many different SSD internal error mechanisms that can result in *partial failures*. These studies have used either lab experiments or field data, and attributed failures that were due to the undelying flash media [15, 20, 21, 27, 28, 29, 30, 34, 40, 64, 65, 71, 84, 89, 98, 117, 118, 172, 177, 161, 105]. Some failures are also attributed to bugs in the FTL code when it is not hardened to handle power faults correctly [201, 202]. Moreover, a field study based on Google's data centers observes that partial failures are significantly more common for SSDs than for HDDs [164]. This issue will likely continue with existing technology trends. For example, increasing densities associated with TLC and QLC flash, and more complex FTLs that support a growing number of interfaces would only increase the number of partial failures. Hence, our work only considers *partial* drive failures where only part of a drive's operation is affected. We do not consider *fail-stop* failures, where the drive as a whole becomes permanently inaccessible.

2.4.1 SSD Errors in the Field and their Manifestation

In this section, we provide an overview of the various mechanisms that can lead to partial failures and how they manifest at the file system level. A summary of our observations is shown in Table 2.1.

Uncorrectable Bit Corruption: Previous work [15, 20, 21, 27, 28, 29, 30, 34, 64, 65, 71, 98, 105, 118] describes a large number of error mechanisms that originate at the flash level and can result in *bit corruption*, including retention errors, read and program disturb

errors, errors due to flash cell wear-out and failing blocks. Virtually all modern SSDs incorporate error correcting codes to detect and correct such bit corruption. However, recent field studies indicate that *uncorrectable bit corruption*, where more bits are corrupted than the error correcting code (ECC) can handle, occurs at a significant rate in the field. For example, a study based on Google field data observes 2-6 out of 1000 drive days with uncorrectable bit errors [164]. Uncorrectable bit corruption manifests as a read I/O error returned by the drive when an application tries to access the affected data ("Read I/O errors" in Table 2.1).

Silent Bit Corruption: This is a more insidious form of bit corruption, where the drive itself is not aware of the corruption and returns corrupted data to the application ("Corruption" in Table 2.1). While there have been field studies on the prevalence of silent data corruption for HDD based systems [11], there is to date no field data on silent bit corruption for SSD-based systems. However, work based on lab experiments shows that 3 out of 15 drive models under test experience silent data corruption in the case of power faults [202]. Note that there are other mechanisms that can lead to silent data corruption, including mechanisms that originate at higher levels in the storage stack, above the SSD device level.

FTL Metadata Corruption: A special case arises when silent bit corruption affects FTL metadata. Among other things, the FTL maintains a mapping of logical to physical (L2P) blocks as part of its metadata [7]. Metadata corruption could lead to "Read I/O errors" or "Write I/O errors", when the application attempts to read or write a page that does not have an entry in the L2P mapping due to corruption. Corruption of the L2P mapping could also result in wrong or erased data being returned on a read, manifesting as "Corruption" to the file system. Note that this is also a silent corruption - i.e. neither the device nor the FTL is aware of these corruptions.

Misdirected Writes: This refers to the situation where during an SSD-internal write operation, the correct data is being written to flash, but at the wrong location. This might be due to a bug in the FTL code or triggered by a power fault, as explained in [202]. At the file system level this might manifest as a "Corruption", where a subsequent read returns wrong data, or a "Read I/O error". This form of corruption is silent, the device does not detect and propagate errors to the storage stack above until invalid data or metadata is accessed again.

Shorn Writes: A shorn write is a write that is issued by the file system, but only partially done by the device. In [202], the authors observe such scenarios surprisingly frequently during power faults, even for enterprise class drives, while issuing properly synchronized I/O and cache flush commands to the device. A shorn write is similar to a "torn write", where only part of a multi-sector update is written to the disk, but

it applies to sector(s) which should have been fully persisted due to the use of a cache flush operation. One possible explanation is the mismatch of write granularities between layers. The default block size for file systems is larger (e.g. 4KB for ext4/F2FS, and 16KB for Btrfs) than the physical device (e.g. 512B). A block issued from the file system is mapped to multiple physical blocks inside the device. As a result, during a power fault, only some of the mappings are updated while others remain unchanged. Even if physical block sizes match that of the file system, another possible explanation is because SSDs include on-board cache memory for buffering writes, shorn writes may also be caused by alignment and timing bugs in the drive's cache management [202]. Moreover, recent SSD architectures use pre-buffering and striping across independent parallel units, which do not guarantee atomicity between them for an atomic write operation [19]. The increase in parallelism may further expose more shorn writes.

At the file system level, a shorn write is not detected until its manifestation during a later read operation, where the file system sees a 4KB block, part of which contains data from the most recent update to the block, while the remaining part contains either old or zeroed out data (if the block was recently erased). While this could be viewed as a special form of silent bit corruption, we consider this as a separate category in terms of how it manifests at the file system level (called "Shorn Write" corresponding to column (d) in Table 2.1). This form of corruption creates a particular pattern (each sequence of 512 bytes within a 4KB block is either completely corrupted or completely correct), compared to the more random corruption event referred to by column (c).

In [202], the authors observe shorn writes manifesting in two patterns, where only the first 3/8th or the first 7/8th of a block gets written and the rest is not. Similarly, in our experiments, we keep only the first 3/8th of a 4KB block. We assume the block has been successfully erased, so the rest of the block remains zeroed out. Our module can be configured to test other shorn write sizes and patterns as well.

Dropped writes: The authors in [202] observe cases where an SSD internal write operation gets dropped even after an explicit cache flush (e.g. in the case of a power fault when the update was in the SSD's cache, but not persisted to flash). If the dropped write relates to FTL metadata, in particular to the L2P mapping, this could manifest as a "Read I/O error", "Write I/O error" or "Corruption" on a subsequent read or write of the data. If the dropped write relates to a file system write, the result is the same as if the file system had never issued the corresponding write. We create a separate category for this manifestation which we refer to as "Lost Write" (column (e) in Table 2.1).

Incomplete Program operation: This refers to the situation where a flash program operation does not fully complete (without the FTL noticing), so only part of a flash page gets written. Such scenarios were observed, for example, under power faults [202]. At the file system level, this manifests as a "Corruption" during a subsequent read of the data.

SSD/Flash Errors	(a)	(b)	(c)	(d)	(e)
Uncorrectable Bit Corruption	✓				
Silent Bit Corruption			✓		
FTL Metadata Corruption	✓	✓	✓		
Misdirected Writes	✓		✓		
Shorn Writes				✓	
Dropped Write	✓	✓	✓		✓
Incomplete Program Operation			✓	✓	
Incomplete Erase Operation			✓		

Table 2.1: *Different types of flash errors and their manifestation in the file system. (a) Read I/O error (b) Write I/O error (c) Corruption (d) Shorn Write (e) Lost Write.*

Incomplete Erase operation: This refers to the situation where a flash erase operation does not completely erase a flash erase block (without the FTL detecting and correcting this problem). Incomplete erase operations have been observed under power faults [202]. They could also occur when flash erase blocks wear-out and the FTL does not handle a failed erase operation properly. Subsequent program operations to the affected erase block can result in incorrectly written data and consequently "Corruption", when this data is later read by the file system.

2.4.2 Comparison with HDD faults

We note that there are also HDD-specific faults that would manifest in a similar way at the file system level. However, the mechanisms that cause faults within each medium are different and can for example affect the frequency of observed errors. One such case are uncorrectable read errors which have been observed at a much higher frequency in production systems using SSDs than HDDs [164], a trend that will likely only get worse with QLC. There are faults though whose manifestation does actually differ from HDDs to SSDs, due to inherent differences in their overall design and operation. For instance, a part affected by a shorn write may contain previously written data in the case of an HDD block, but would contain zeroed out data if that area within the SSD has been correctly erased. In addition, the large degree of parallelism inside SSDs makes correctness under power faults significantly more challenging than for HDDs (for example, ensuring atomic writes across parallel units). Finally, file systems might modify their behavior and apply different fault recovery mechanisms for SSDs and HDDs. For example, Btrfs turns off metadata duplication by default when deployed on top of an SSD.

Programs
`mount`, `umount`, `open`, `creat`, `access`, `stat`, `lstat`, `chmod`, `chown`, `utime`, `rename`, `read`, `write`, `truncate`, `readlink`, `symlink`, `unlink`, `chdir`, `rmdir`, `mkdir`, `getdirentries`, `chroot`

Table 2.2: *The programs used in our study. Each one stresses a single system call and is invoked several times under different file system images to increase coverage.*

2.4.3 Device Mapper Tool for Error Emulation

The key observation from Section 2.4 is that all SSD faults we consider manifest in one of five ways, corresponding to the five columns (a) to (e) in Table 2.1. This section describes a device mapper tool we created to emulate all five scenarios.

In order to emulate SSD error modes and observe each individual file system's response, we need to intercept the block I/O requests between the file system and the block device. We leverage the Linux *device mapper* framework to create a virtual block device that intercepts requests between the file system and the underlying physical device. This allows us to operate on block I/O requests and simulate faults as if they originate from a physical device. It also helps us observe the file system's reaction without modifying its source code. In this way, we can perform tracing, parse file system metadata, and alter block contents online, for both read and write requests, while the file system is mounted. For this study, we use the Linux kernel version 4.17.

Our module can intercept read and write requests for selected blocks as they pass through the block layer and return an error code to the file system, emulating categories (a) "Read Error" and (b) "Write Error" in Table 2.1. Possible parameters include the request's type (read/write), block number, and data structure type. In the case of multiple accesses to the same block, one particular access can be targeted. We also support corruption of specific data structures, fields and bytes within blocks, allowing us to emulate category (c) "Corruption". The module can selectively shear multiple sectors of a block before sending it to the file system or writing it on disk, emulating category (d) "Shorn Write". Our module can further drop one or more blocks while writing the blocks corresponding to a file system operation, emulating the last category (e) "Lost Write". The module's API is generic across file systems and can be expanded to different file systems. Our module can be found at [62].

2.4.4 Test Programs

We perform injection experiments while executing test programs chosen to exercise different parts of the POSIX API, similar to the "singlets" used by Prabhakaran et *al.* [143]. Each individual program focuses on one system call, such as `mkdir` or `write`. Table 2.2 lists all the test programs that we used in our study. For each test program, we populate the disk with different files and directory structures to increase code coverage. For exam-

ple, we generate small files that are stored inline within an inode, as well as large files that use indirect blocks. All our programs pedantically follow POSIX semantics; they call `fsync(2)` and `close(2)`, and check the return values to ensure that data and metadata has successfully persisted to the underlying storage device.

2.4.5 Targeted Error Injection

Our goal is to understand the effect of block I/O errors and corruption in detail depending on which part of a file system is affected. That means our error injection testbed requires the ability to target specific data structures and specific fields within a data structure for error injection, rather than randomly injecting errors. We therefore need to identify for each program which data structures are involved and how the parts of the data structure map to the sequence of block accesses generated by the program. We rely on a combination of approaches to derive the relationship between sequence of block accesses and data structures within each file system.

First, we initialize the file system to a clean state with representative data. We then run a specific test program (Table 2.2) on the file system image, capturing traces from `blktrace` and the kernel to learn the program's actual accessed blocks. Reading the file system source code also enables us to put logic inside our module to interpret blocks as requests pass through it. Lastly, we use offline tools such as `dumpe2fs`, `btrfs-inspect`, and `dump.f2fs` to inspect changes to disk contents. Through these multiple techniques, we can identify block types and specific data structures within the blocks. Table 2.3 summarizes our approach to identify different data structures in each of the file systems.

After identifying all the relevant data structures for each program, we re-initialize the disk image and repeat test program execution for error injection experiments. We use the same tools, along with our module, to inject errors to specific targets. A single block I/O error or data corruption is injected into a block or data structure during each execution. This allows us to achieve better isolation and characterization of the file system's reaction to the injected error.

Our error injection experiments allow us to measure both immediate and longer-term effects of device faults. We can observe immediate effects on program execution for some cases, such as user space errors or kernel panics (e.g. from write I/O errors). At the end of each test program execution, we unmount the file system and perform several offline tests to verify the consistency of the disk image, regardless of whether the corruption was silent or not (e.g. persisting lost/shorn writes): we invoke the file system's integrity checker (*fsck*), check if the file system is mountable, and check whether the program's operations have been successfully persisted by comparing the resultant disk image against the expected one. We also explore longer-term effects of faults where the test programs access data that were previously persisted with errors (read I/O, reading corrupted or shorn write data). In this study, we use *btrfs-progs v4.4*, *e2fsprogs v1.42.13*, and *f2fs-tools v1.10.0* for our error injection experiments.

ext4	
Data Structure	**Approach**
super block, *group descriptor*, *inode blocks*, *block bmap*, *inode bmap*	`dumpe2fs`
dir_entry	`debugfs`, get block inode, stat on inode number, check file type
extent	`debugfs`, check for extent of a file or directory path
data	`debugfs`, get block inode, stat on inode number, check file type
journal	`debugfs`, check if parent inode number is 8
Btrfs	
Data Structure	**Approach**
`DIR_ITEM`, `DIR_INDEX`, `INODE_REF`, `INODE_DATA`, `EXTENT_DATA`	`btrfs-debug-tree`
F2FS	
Data Structure	**Approach**
superblock, *checkpoint*, *SIT*, *NAT*, *inode*, *d/ind node*, *dir. block*, *data*	device mapper module

Table 2.3: *The approach to type blocks collected using either* `blktrace` *or our own device mapper module.*

2.4.6 Detection and Recovery Taxonomy

We report the *detection* and *recovery* policies of all three file systems with respect to the data structures involved. We characterize each file system's reaction via all observable interfaces: system call return values, changes to the disk image, log messages, and any side-effects to the system (such as kernel panics). We classify the file system's detection and recovery based on a taxonomy that was inspired by previous work [143], but with some new extensions: unlike [143], we also experiment with file system integrity checkers and their ability to detect and recover from errors that the file system might not be able to deal with and as such, we add a few additional categories within the taxonomy that pertain to file system checkers. Also, we create a separate category for the case where the file system is left in its previous consistent state prior to the execution of the program ($R_{Previous}$). In particular, if the program involves updates on the system's metadata, none of it is reflected to the file system. Table 2.4 presents our taxonomy in detail.

A file system can detect the injected errors online by checking the return value of the block I/O request ($D_{ErrorCode}$), inspecting the incoming data and performing some sanity checks (D_{Sanity}), or using redundancies, e.g. in the form of checksums ($D_{Redundancy}$). A

Symbol	Level	Description
◯	$\mathbf{D_{Zero}}$	No detection.
–	$\mathbf{D_{ErrorCode}}$	Check the error code returned from the lower levels.
\	$\mathbf{D_{Sanity}}$	Check for invalid values within the contents of a block.
/	$\mathbf{D_{Redundancy}}$	Checksums, replicas, or any other form of redundancy.
|	$\mathbf{D_{Fsck}}$	Detect error using the system checker.
◯	$\mathbf{R_{Zero}}$	No attempt to recover.
/	$\mathbf{R_{Retry}}$	Retry the operation first before returning an error.
|	$\mathbf{R_{Propagate}}$	Error code propagated to the user space.
\	$\mathbf{R_{Previous}}$	File system resumes operation from the state exactly before the operation occurred.
–	$\mathbf{R_{Stop}}$	The operation is terminated (either gracefully or abruptly); the file system may be mounted as read-only.
■	$\mathbf{R_{Fsck\_Fail}}$	Recovery failed, the file system cannot be mounted.
■	$\mathbf{R_{Fsck\_Partial}}$	The file system is mountable, but it has experienced data loss in addition to operation failure.
■	$\mathbf{R_{Fsck\_Orig}}$	Current operation fails, file system restored to pre-operation state.
■	$\mathbf{R_{Fsck\_Full}}$	The file system is fully repaired and its state is the same with the one generated by the execution where the operation succeeded without any errors.

Table 2.4: *The levels of our detection and recovery taxonomy.*

successful detection should alert the user via system call return values or log messages. To recover from errors, the file system can take several actions. The most basic action is simply passing along the error code from the block layer ($R_{Propagate}$). The file system can also decide to terminate the execution of the system call, either gracefully via transaction abort, or abruptly such as crashing the kernel (R_{Stop}). Lastly, the file system can perform retries (R_{Retry}) in case the error is transient, or use its redundancy data structures to recover the data.

It is important to note that for block I/O errors, the actual data stored in the block is not passed to the disk or the file system. Hence, no sanity check can be performed and D_{Sanity} is not applicable. Similarly, for silent data corruption experiments, our module does not return an error code, so $D_{ErrorCode}$ is not relevant.

We also run each file system's *fsck* utility and report on its ability to detect and recover file systems errors offline. It may employ different detection and recovery strategies than the online file system. The different categories for *fsck* recovery are shown in Table 2.4.

2.4.7 I/O Latency Injection

I/O latency is another characteristic that can potentially impact the reliability of the file system. Flash devices are well known to suffer from latency spikes due to various reasons, from internal ECC calculations to FTL garbage collection [199, 193]. Some of these have been categorized as *fail-slow* failures in the literature. While the increased latency may not

affect the correctness of the individual device I/O, it can potentially have an impact on higher-level software [68, 136]. To investigate whether increased I/O delays would have adverse effects on the file system, we inject artificial delays to targeted block requests using our module. We can specify delays from millisecond granularities to many seconds, and can delay any individual block request during a sequence of accesses.

2.5 Results

Tables 2.5 and 2.6 provide a high-level summary of the results from our error injection experiments following the detection and recovery taxonomy from Table 2.4. Our results are organized into six columns corresponding to the fault modes we emulate. The six tables in each column represent the fault detection and recovery results for each file system under a particular fault. The columns (a-w) in each table correspond to the programs listed in Table 2.2, which specify the operation during which the fault mode was encountered, and rows correspond to the file system specific data structure, that was affected by the fault.

Note that the columns in Tables 2.5 and 2.6 have a one-to-one correspondence to the fault modes described in Section 2.4 (Table 2.1), with the exception of *shorn writes*. After a shorn write is injected during test program execution and persisted to the flash device, we examine two scenarios where the persisted partial data is accessed again: during fsck invocation (*Shorn Write + Fsck* column) and test program execution (*Shorn Write + Program Read* column).

2.5.1 Btrfs

Btrfs [154] is a copy-on-write file system that uses a B-tree as its main on-disk data structure. An overwrite of any data structure or block leads to the creation of a new block with the updated contents.

B-Tree Semantics.

All B-trees in the file system contain multiple tree nodes. Each tree node contains a reference count. The reference count of a child node is incremented each time a new or updated parent node points to the child node. Similarly, a child node's reference count is decremented upon the deletion of any of its parent nodes. When the reference count becomes equal to zero, then the corresponding child node is reclaimed by the file system.

We observe in Table 2.5 that Btrfs is the only file system that consistently detects all I/O errors as well as corruption events, including those affecting data (rather than only metadata). It achieves this through the extensive use of checksums.

However, we find that Btrfs is much less successful in recovering from any issues than the other two file systems. It is the only file system where four of the six error

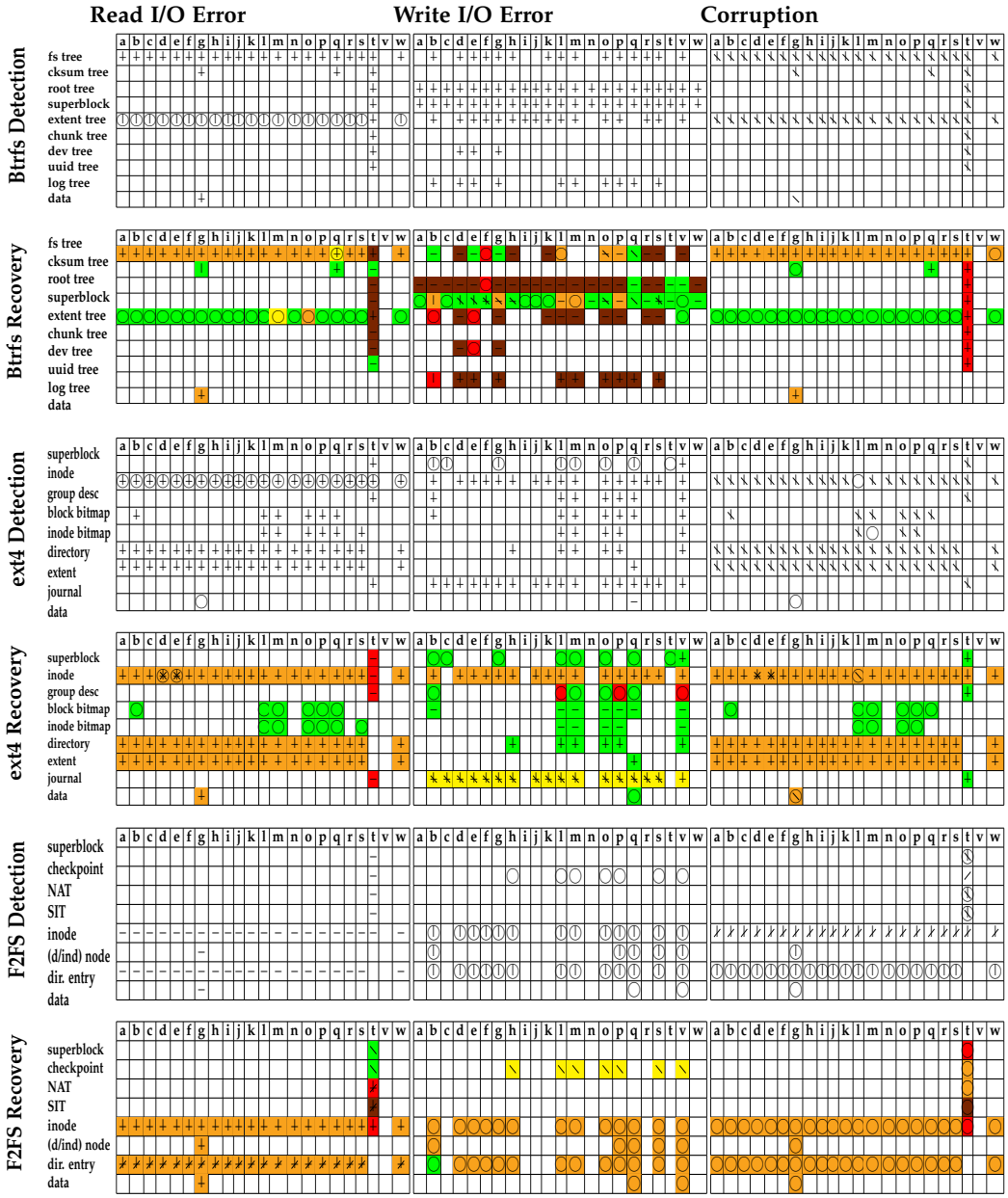


Table 2.5: The results of our analysis on the detection and recovery policies of Btrfs, ext4, and F2FS for different read, write, and corruption experiments. The programs that were used are: **a:** access **b:** truncate **c:** open **d:** chmod **e:** chown **f:** utimes **g:** read **h:** rename **i:** stat **j:** lstat **k:** readlink **l:** symlink **m:** unlink **n:** chdir **o:** rmdir **p:** mkdir **q:** write **r:** getdirentries **s:** creat **t:** mount **v:** umount **w:** chroot. An empty box indicates that the block type is not applicable to the program in execution. Superimposed symbols indicate that multiple mechanisms were used.

○ D_{Zero}	– $D_{ErrorCode}$	\ D_{Sanity}	/ $D_{Redundancy}$	| D_{Fsck}
○ R_{Zero}	/ R_{Retry}	| $R_{Propagate}$	\ $R_{Previous}$	– R_{Stop}
■ R_{Fsck_Full}	■ R_{Fsck_Orig}	■ $R_{Fsck\_Partial}$	■ R_{Fsck_Fail}	■ Crash/Panic+R_{Fsck_fail}

Table 2.6: The results of our analysis on the detection and recovery policies of Btrfs, ext4, and F2FS for different shorn write + program read, shorn write + fsck, and lost write experiments. The programs that were used are: **a:** access **b:** truncate **c:** open **d:** chmod **e:** chown **f:** utimes **g:** read **h:** rename **i:** stat **j:** lstat **k:** readlink **l:** symlink **m:** unlink **n:** chdir **o:** rmdir **p:** mkdir **q:** write **r:** getdirentries **s:** creat **t:** mount **v:** umount **w:** chroot. An empty box indicates that the block type is not applicable to the program in execution. Superimposed symbols indicate that multiple mechanisms were used.

◯ D_{Zero}	– $D_{ErrorCode}$	\ D_{Sanity}	/ $D_{Redundancy}$	| D_{Fsck}
◯ R_{Zero}	/ R_{Retry}	| $R_{Propagate}$	\ $R_{Previous}$	– R_{Stop}
■ R_{Fsck_Full}	■ R_{Fsck_Orig}	■ $R_{Fsck\_Partial}$	■ R_{Fsck_Fail}	■ Crash/Panic+R_{Fsck_fail}

modes can lead to a kernel crash or panic and subsequently a file system that cannot be mounted even after running `btrfsck`. It also has the largest number of scenarios that result in an unmountable file system after `btrfsck` (even if not preceded by a kernel crash). Furthermore, we find that node level checksums, although good for detecting block corruption, they remove an entire node even if a single byte becomes corrupted. As a result, large chunks of data are removed, causing data loss.

The copy-on-write nature of Btrfs makes it efficient for creating snapshots and subvolumes. Each instance of the Btrfs file system contains one or more subvolumes. Each subvolume may contain one or more files and directories. Subvolumes may also share file-extents. For every subvolume, a snapshot can be created as a read-only or a read-write point-in-time image of the file system. Any changes made to a newly created subvolume do not effect the files and directories of the main subvolume. Btrfs can be mounted using the default subvolume at root directory, or any other subvolume snapshot which appear as directories within the root directory of the file system.

Before we describe the results in more detail below, we provide a brief summary of Btrfs data structures. The Btrfs file system arranges data in the form of a forest of trees, each serving a specific purpose (e.g. file system tree (*fstree*) stores file system metadata, checksum tree (*csumtree*) stores file/directory checksums). Btrfs maintains checksums for all metadata within tree nodes. Checksum for data is computed and stored separately in a checksum tree. A *root tree* stores the location of the root of all other trees in the file system. Since Btrfs is a copy-on-write file system, all changes made on a tree node are first written to a different location on disk. The location of the new tree nodes are then propagated across the internal nodes up to the root of the file system trees. Finally, the *root tree* needs to be updated with the location of the other changed file system trees.

Read errors.

All errors get detected ($D_{ErrorCode}$) and registered in the operating system's message *log*, and the current operation is terminated (R_{Stop}). `btrfsck` is able to run, detect and correct the file system in most cases, with two exceptions. When the *fstree* structure is affected, `btrfsck` removes blocks that are not readable and returns an I/O error. Another exception is when a read I/O error is encountered while accessing key tree structures during a `mount` procedure: `mount` fails and `btrfsck` is unable to repair the file system.

Corruption.

Corruption of any B-tree node. Checksums inside each tree node enable reliable detection of corruption; however, Btrfs employs a different recovery protocol based on the type of the underlying device. When Btrfs is deployed on top of a hard disk, it provides recovery from metadata corruption using metadata replication. Specifically, reading a corrupted block leads to `btrfs-scrub` being invoked, which replaces the corrupted primary

metadata block with its replica. Note that `btrfs-scrub` does not have to scan the entire file system; only the replica is read to restore the corrupted block. However, in case the underlying device is an SSD, Btrfs turns off metadata replication by default for two reasons [23]. First, an SSD can remap a primary block and its replica internally to a single physical location, thus deduplicating them. Second, SSD controllers may put data written together in a short time span into the same physical storage unit (i.e. cell, erase block, etc.), which is also a unit of SSD failures. Therefore, `btrfs-scrub` is never invoked in the case of SSDs, as there is no metadata duplication. This design choice causes a single bit flip of *fstree* to wipe out files and entire directories within a *B-Tree* node. If a corrupted tree node is encountered while `mount` reads in all metadata trees into memory, the consequences are even more severe: the operation fails and the disk is left in an inconsistent and irreparable state, even after running `btrfsck`.

Directory corruption. We observe that when a node corruption affects a directory, the corruption could actually be recovered, but Btrfs fails to do so. For performance reasons, Btrfs maintains two independent data structures for a directory (`DIR_ITEM` and `DIR_INDEX`). If one of these two becomes corrupted, the other data structure is not used to restore the directory. This is surprising considering that the existing redundancy could easily be leveraged for increased reliability.

Write errors.

Superblock & Write I/O errors. Btrfs has multiple copies of its superblock, but interestingly, the recovery policy upon a write error is not identical for all copies. The superblocks are located at fixed locations on the disk and are updated at every write operation. The replicas are kept consistent, which differs from ext4's behavior. We observe that a write I/O error while updating the primary superblock is considered severe; the operation is aborted and the file system remounts as *read-only*. On the other hand, write I/O errors for the secondary copies of the superblock are detected, but the overall operation completes successfully, and the secondary copy is not updated. While this allows the file system to continue writing, this is a violation of the implicit consistency guarantee between all superblocks, which may lead to problems in the future, as the system operates with a reduced number of superblock copies.

Tree Node & Write I/O errors. A write I/O error on a tree node is registered in the operating system's message *log*, but due to the file system's asynchronous nature, errors cannot be directly propagated back to the user application that wrote the data. In almost all cases, the file system is forced to mount as *read-only* (R_{Stop}). A subsequent `unmount` and `btrfsck` run in repair mode makes the device unreadable for *extentTree*, *logTree*, *rootTree* and the root node of the *fstree*.

Shorn Write + Program Read.

We observe that the behavior of Btrfs during a read of a shorn block is similar to the one we observed earlier for corruption. The only exception is the superblock, as its size is smaller than the 3/8th of the block and it does not get affected.

Shorn Write + Fsck.

Shorn writes on *root tree* cause the file system to become unmountable and unrecoverable even after a `btrfsck` operation. We also find kernel panics during shorn writes as described in Section 2.5.1.

Lost Writes.

Errors get detected only during `btrfsck`. They do not get detected or propagated to user space during normal operation. `btrfsck` is unable to recover the file system, which is rendered unmountable due to corruption. The only recoverable case is a lost write to the superblock; for the remaining data structures, the file system eventually becomes unmountable.

Subvolume creation.

```
create_pending_snapshots()
  create_pending_snapshot() // BUG() on non-zero value
    btrfs_set_inode_index()
      btrfs_set_inode_index_count()
        btrfs_search_slot()
          read_block_for_search()
            btrfs_buffer_uptodate() // returns -EIO
            btrfs_verify_level_key() // returns -EUCLEAN
```

Listing 2.1: Error propagation during Btrfs snapshot creation

Table 2.7 summarizes our error injection results regarding subvolume creation. The data structures updated are the *root tree*, *superblock* and the file system *extent tree*. We note that a write error on the *root tree* leads to an unmountable and unrecoverable file system - similar to our test programs in Table 2.5 and Table 2.6. Although `btrfsck` is able to detect the errors, it fails to recover the file system which remains unmountable. Write I/O errors related to the *extent tree* are detected by the file system and the subvolume creation procedure is terminated. However, shorn writes and lost writes on the *extent tree* are not detected and not repaired by the file system checker. An error on the *superblock* during subvolume creation results in an unmountable file system only during a shorn write. Write I/O errors are detected by the file system; in this case, the subvolume creation

operation is terminated but the *superblock* is recovered using the secondary superblock replica. A lost write on the primary *superblock* is recovered from the backup replica and the requested subvolume is successfully created.

Snapshot creation.

Table 2.8 summarizes error injection results for snapshot creation. A snapshot is also a subvolume that has the same initial contents as the original subvolume. To ensure that the snapshot creation workflow accesses a range of data structures, we first create a snapshot of the root subvolume after running the initialization workload. We then create a snapshot of the initialization workload, which leads to a range of block-type accesses. In addition to trees that get accessed by test programs in 2.5 and 2.6, snapshot creation also updates the *uuid tree* to create a persistent universally unique identifier (uuid) to subvolume mapping for fast access.

For all read related errors, we find that errors related to an *fstree* block lead to a kernel panic that has the same root cause as the one shown in Listing 2.1. During a *fstree* read corruption or a shorn write, `btrfs_verify_level_key()` fails and an `EIO` is returned. During a read I/O error, `btrfs_buffer_uptodate()` checks the state of the read buffer, which is set to `EXTENT_BUFFER_READ_ERR` during the read I/O request. Both errors are propagated upwards to `btrfs_create_pending_snapshot()` that crashes with a generic `BUG()` statement[1]. This shows that there are still loose ends to tie in file system workflows when it comes to handling media corruptions, and file system developers should gracefully terminate upon receiving media errors from the underlying storage medium. Finally, a read error on an *extent tree* does not lead to a file system inconsistency, is not detected by the file system and leads to successful snapshot creation.

While analyzing write errors, we find that all write errors on the *root tree* remain undetected and lead to an unmountable file system, similar to our observations for the test programs in Tables 2.5, 2.6 and subvolumes in Table 2.7. A write I/O error on the *superblock* and the *uuid tree* is detected and stopped by the file system. A shorn write and lost write on the *root tree*, *uuid tree* and *superblock* renders the file system unmountable, except a lost write on the *superblock*, which can be recovered using the backup superblock replica.

Bugs found/reported.

We submitted 2 bug reports for Btrfs. The first bug report is related to the corruption of a `DIR_INDEX` key. The file system was able to detect the corruption but deadlocks while listing the directory entries. This bug was fixed in a later version [22]. The second bug

[1] BUG() is a statement used as a debugging help and indicates that something went wrong inside the kernel. In this case, the kernel prints out the contents of the registers and the stack trace associated with the current process and then immediately kills the process.

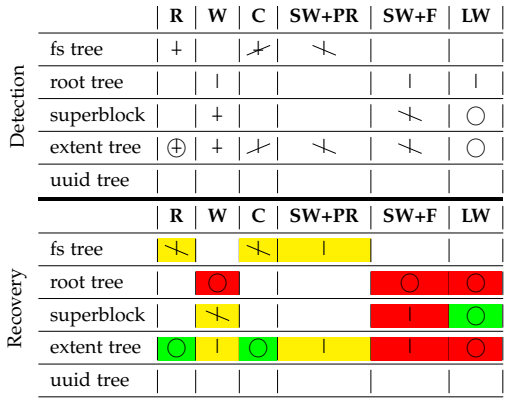


Table 2.7: Btrfs subvolume creation

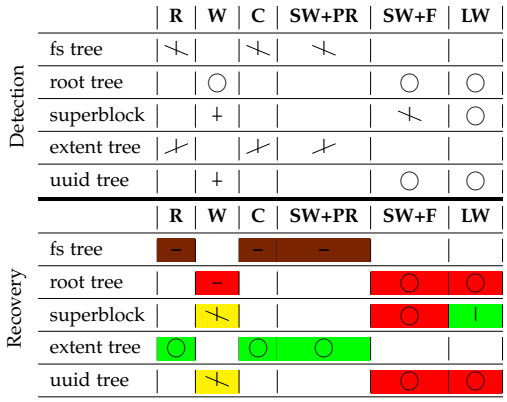


Table 2.8: Btrfs snapshot creation

Table 2.9: *

The results of our analysis on the detection and recovery policies of Btrfs during subvolume and snapshot creation. The error modes that were used are: ***R:*** *Read I/O,* ***W:*** *Write I/O,* ***C:*** *Corruption,* ***SW+PR:*** *Shorn Write + Program Read,* ***SW+F:*** *Shorn Write + Fsck, and* ***LW:*** *Lost Write. An empty box indicates that the block type is not applicable to the program in execution. Superimposed symbols indicate that multiple mechanisms were used.*

◯ D_{Zero}	– $D_{ErrorCode}$	\ D_{Sanity}	/ $D_{Redundancy}$	| D_{Fsck}
◯ R_{Zero}	/ R_{Retry}	| $R_{Propagate}$	\ $R_{Previous}$	– R_{Stop}
■ R_{Fsck_Full}	■ R_{Fsck_Orig}	■ $R_{Fsck\_Partial}$	■ R_{Fsck_Fail}	■ *Crash/Panic*+R_{Fsck_fail}

is related to read I/O errors specifically on the root directory, which can cause a kernel panic for certain programs. We encountered 2 additional bugs during a shorn write that result in a kernel panic, both having the same root cause. The first case involves a shorn write to the root of the *fstree*, while the second case involves a shorn write to the root of the *extent tree*. In both cases, there is a mismatch in the leaf node size, which forces Btrfs to print the entire tree in the operating system's message *log*. While printing the leaf block, another kernel panic occurs where the size of a Btrfs *item* does not match the Btrfs *header*. Rebooting the kernel and running `btrfsck` fails to recover the file system.

2.5.2 ext4

ext4 is the default file system for many widely used Linux distributions and Android devices. The entire file system is divided into multiple block groups. Each block group contains a group descriptor table, an inode table, a block and inode bitmap, and free blocks that can be used for creating either a file or a directory. Like Btrfs and F2FS, ext4 also has a superblock. ext4 is an in-place or overwrite file system, i.e. any changes made to any data structure in the file system overwrite its previous version.

ext4 makes use of a journal to provide recovery against crash faults. The *journal* is a circular log where all metadata updates are sequentially written before the main file system is updated. First, data corresponding to a file system operation is written to the in-place location of the file system. Next, a transaction log is written on the *journal*. Once

the transaction log is written on the *journal*, the transaction is said to be *committed*. When the *journal* is full or sufficient time has elapsed, a *checkpoint* operation takes place that writes the in-memory metadata buffers to the in-place metadata location on the disk. On the event of a crash before the transaction is committed, the file system transaction is discarded. If the commit has taken place successfully on the *journal* but the transaction has not been checkpointed, the file system replays the *journal* during remount, where all metadata updates that were committed on the *journal* are recovered from the journal and written to the main file system.

The ext4 file system employs physical journaling, where all metadata blocks that have to be written to the main file system are logged as-is on the journal. Each transaction written on the ext4 journal consists of three types of blocks - the transaction descriptor block, the metadata blocks, and the transaction commit block. The transaction descriptor block contains important fields like the *transaction id* and *metadata tags*, which denote the metadata block numbers that are part of the current transaction, while the transaction commit block signifies the end of the transaction.

ext4 is able to recover from an impressively large range of fault scenarios. Unlike Btrfs, it makes little use of checksums unless the *metadata_csum* feature is enabled explicitly during file system creation. Furthermore, it deploys a very rich set of sanity checks when reading data structures such as directories, inodes and extents[2], which helps it deal with corruptions.

It is also the only one of the three file systems that is able to recover lost writes of multiple data structures, due to its in-place nature of writes and a robust file system checker. However, there are a few exceptions where the corresponding issue remains uncorrectable (see the red cells in Table 2.6 associated with ext4's recovery).

Furthermore, we observe instances of data loss caused by shorn and lost writes involving write programs, such as `create` and `rmdir`. For shorn writes, ext4 may incur silent errors, and not notify the user about the errors.

Before describing some specific issues below, we point out that our ext4 results are very different from those reported for ext3 in [143], where a large number of corruption events and several read and write I/O errors were not detected or handled properly. Clearly, in the 13 years that have passed since then, ext developers have made improvements in reliability a priority, potentially motivated by the findings in [143].

I/O errors, corruption and shorn writes of inodes.

The most common scenario leading to data loss (but still a consistent file system) is a fault, in particular read I/O error, corruption or shorn write, that affects an inode, which results in the data of all files having their inode structure stored inside the affected inode block becoming inaccessible.

[2] We report failure results for both directory and file extents together. Since our pre-workload generation creates a number of files and directories in the root directory, at least 1 extent block corresponding to the root directory gets accessed by all programs.

		R	W	C	SW	LW
Detection	Superblock	–	+	+	◯	◯
	Descriptor Block	+	+	◐	+	◯
	Metadata	+	+	+	+	◯
	Commit Block	+	+	◯	◯	◯
		R	**W**	**C**	**SW**	**LW**
Recovery	Superblock	+	+	+	◯	◯
	Descriptor Block	+	–	◯	+	◯
	Metadata	+	–	+	+	◯
	Commit Block	+	–	◯	◯	◯

Table 2.10: *The results of our analysis on the detection and recovery policies of ext4 during journal replay. The error modes that were used are:* ***R:*** *Read I/O,* ***W:*** *Write I/O,* ***C:*** *Corruption,* ***SW:*** *Shorn Write, and* ***LW:*** *Lost Write. An empty box indicates that the block type is not applicable to the program in execution. Superimposed symbols indicate that multiple mechanisms were used.*

◯ D_{Zero} – $D_{ErrorCode}$ \ D_{Sanity} / $D_{Redundancy}$ | D_{Fsck}

◯ R_{Zero} / R_{Retry} | $R_{Propagate}$ \ $R_{Previous}$ – R_{Stop}
■ R_{Fsck_Full} ■ R_{Fsck_Orig} ■ $R_{Fsck\_Partial}$ ■ R_{Fsck_Fail} ■ Crash/Panic+R_{Fsck_fail}

Read I/O errors, corruption and shorn writes of directory blocks.

A shorn write involving a directory block is detected by the file system and eventually, the corresponding files and directories are removed. Empty files are completely removed by `e2fsck` by default, while non-empty files are placed into the *lost+found* directory. However, the parent-child relationship is lost, which we encode as $R_{Fsck\_Partial}$.

Write I/O errors and group descriptors.

There is only one scenario where `e2fsck` does not achieve at least partial success: When `e2fsck` is invoked after a write error on a group descriptor, it tries to rebuild the group descriptor and write it to the same on-disk location. However, as it is the same on-disk location that generated the initial error, `e2fsck` encounters the same write error and keeps restarting, running into an infinite loop for 3 cases (see R_{Fsck_Fail}).

Read I/O error during mount.

ext4 fails to complete the `mount` operation if a read I/O error occurs while reading a critical metadata structure. In this case, the file system cannot be mounted even after invoking `e2fsck`. We observe similar behavior for the other two file systems as well.

Journal corruption.

In all our previous experiments, listed in Table 2.5 and Table 2.6, we safely unmount the file system before checking for error detection and recovery. In order to test the resiliency of ext4's journal against corruptions, we use an alternative approach to ensure that the data structures logged on the journal are accessed during remount. Since each transaction written on the journal is accessed only if an unclean shutdown occurs, we simulate improper shutdown by forcefully rebooting the system. Once the system reboots, ext4 reads and replays journal contents in three passes. We analyze the effect of error injection on different journal data structures while rebooting and describe our observations. Note that unlike our previous experiments, we show both *Shorn Read + Program Write* and *Shorn Write + Fsck* in the same table as our test program in both cases is the `mount` operation. A repeated I/O error on accessing the *journal superblock* leads to an unmountable file system. Although `e2fsck` detects the error, it is unable to recover the file system. A corruption of the *journal superblock* is fixed by the file system checker. Both shorn writes and lost writes of the *journal superblock* remain undetected; however, a subsequent file system `mount` operation is successful. We observe two cases where silent data loss might occur while the ext4 file system is being mounted; a corruption or a lost write of the *journal descriptor block* or the *journal commit block* aborts the journal replay procedure. Internally, journal replay consists of three passes - the *scan pass*, the *revoke pass*, and the *replay pass*. During the scan pass, the file system checks the integrity of different transactions logged on the journal. To identify any transaction, a *journal header* is added at the beginning of all *journal metadata blocks*. When a corruption or a lost write occurs, the scan pass terminates as it cannot find the relevant *journal header magic number* at the beginning of the expected block; as a result, all subsequent transactions are discarded. A subsequent `mount` operation takes place without any errors, but a recursive directory traversal reveals metadata inconsistency, that can be fixed by a subsequent `e2fsck` invocation.

A write I/O error on both the *journal descriptor block* and the *journal commit block* is detected and the file system stops all subsequent write operations by remounting as *read-only*. A `e2fsck` scan fixes any inconsistencies and the file system becomes mountable once again. A lost write of both the *journal descriptor block* and the *journal commit block* is not detected during file system operation. A subsequent remount and recursive scan of the file system reveals metadata inconsistency which is successfully repaired by a subsequent `e2fsck` invocation. Any error to the *journal metadata* block is not detected during a subsequent `mount` operation. A recursive directory scan of the file system reveals metadata inconsistency which is subsequently fixed using `e2fsck`.

It is important to note that the role of journaling is to ensure that the time-consuming *fsck* operation does not run each time a crash occurs. However, when a journal corruption takes place, we observe that `e2fsck` still needs to be invoked in order to ensure file system consistency. `e2fsck` replays the journal and subsequently performs a full file system check to ensure the integrity of the file system, which defeats the purpose of journaling.

2.5.3 F2FS

F2FS is a log-structured file system designed for devices that use an FTL. The file system is composed of fixed-size *segments*, which are groups of blocks. The default size of a block is equal to $4K$, while that of a *segment* is $2MB$, which corresponds to 512 blocks in each *segment*. At any given time, data and metadata are separately written into 6 (default configuration) active *segments*, called *log segments*, which are grouped by data/metadata, files/directories, and other heuristics. This *multi-head logging* allows similar blocks to be placed together and increases the number of sequential write operations.

F2FS divides its logical address space into the *metadata region* and the *main area*. The metadata region is stored at a fixed location and includes the Checkpoint (CP), the Segment Information Table (SIT), and the Node Address Table (NAT). The checkpoint stores information about the state of the file system and is used to recover from system crashes. SIT maintains information on the *segments* (the unit at which F2FS allocates storage blocks) in the main area, while NAT contains the block addresses of the file system's *nodes*, which comprise file/directory inodes, direct, and indirect nodes. F2FS uses a two-location approach for these three structures. In particular, one of the two copies is "active" and used to initialize the file system's state during `mount`, while the other is a shadow copy that gets updated during the file system's execution. Finally, each copy of the checkpoint points to its corresponding copy of SIT and NAT.

F2FS's behavior when encountering read/write errors or corruption differs significantly from that of ext4 and Btrfs. While read failures are detected and appropriately propagated in nearly all scenarios, we observe that F2FS consistently fails to detect and report any write errors, independently of the operation that encounters them and the affected data structure. Furthermore, our results indicate that F2FS is not able to deal with lost and shorn writes effectively and eventually suffers from data loss. In some cases, a run of the file system's checker (called `fsck.f2fs`) can bring the system to a consistent state, but in other cases, the consequences are severe. We describe some of these cases in more detail below.

Read errors.

Checkpoint / NAT / SIT blocks & read errors. During its `mount` operation, if F2FS encounters a read I/O error while trying to fetch any of the checkpoint, NAT, or SIT blocks, then it mounts as *read-only*. Specifically for NAT and SIT blocks, the file system retries the operation before eventually switching to a *read-only* mode. Additionally, F2FS cannot be mounted if the inode associated with the root directory cannot be accessed. In general, `fsck.f2fs` cannot help the file system recover from the error since it terminates with an assertion error every time it cannot read a block from the disk.

Write errors & Lost Writes.

We observe that F2FS does not detect write errors (as injected by our framework), leading to different issues, such as corruption, reading garbage values, and potentially data loss. As a result, during our experiments, newly created or previously existing entries have been completely discarded from the system, applications have received garbage values, and finally, the file system has experienced data loss due to an incomplete recovery protocol (i.e. when `fsck.f2fs` is invoked). We describe several cases in detail below.

Inodes & write errors. We observe that when the write operation for an updated inode returns an error, the file system does not detect this and eventually experiences data loss. This happens because the updated inode is never persisted on disk, but the file system's metadata has been updated to point to the new location. When called `fsck.f2fs` is invoked, it detects that an invalid inode is written on disk and discards the entry from the system, leading to loss of the corresponding file.

Direct/indirect nodes & write errors. We observe that errors when writing direct or indirect nodes are not detected and recovered. As a result, the file system might later access these blocks and read garbage data that is interpreted to locate blocks. If this garbage data can be interpreted as a valid file system location, then a read by an application might receive garbage. If it is not a valid file system location, then an error is returned to the application.

Directory entries & write errors. When the structure of a directory is modified by inserting or removing entries, then the file system takes a new checkpoint. However, when our error injecting framework prevents the write operation of the block containing the directory's entries from occurring, we observe that eventually, the file system experiences data loss. Once again, the file system's metadata is updated during the checkpoint procedure and stores a new on-disk location for the block containing the directory's entries. Since the write operation for that particular block never occurred, all the entries in that block are marked as *unreachable* during the invocation of `fsck.f2fs`. Individual subdirectories are never created inside the *lost_found* directory; only regular files are moved into that directory. As all files are stored together inside the *lost_found* directory, if there are different files with the same name, only one is stored in the end.

Data & write errors. We observe that errors during a write operation of a file's data (either involving a data block or an indirect node) go undetected and are not recovered. In particular, if the update to the corresponding inode succeeds, but the write of the data fails, an application later reading the data will receive garbage, instead of an error.

Lost writes.

The difference between a lost write and a write I/O error is that a lost write is silent, i.e. the operation does not return with an error. Since F2FS does not detect write I/O errors, lost writes have the same effect (as described above).

Corruption.

Data corruption is reliably detected only for inodes and checkpoints, which are the only data structures protected by checksums, but even for those two data structures, recovery can be incomplete, resulting in the loss of files (and the data stored in them). The corruption of other data structures can lead to even more severe consequences. For example, the corruption of the superblock can go undetected and lead to an unmountable file system, even if the second copy of the superblock is intact. We have filed two bug reports related to the issues we have identified and one has already resulted in a fix. Below we describe some of the issues around corruption in more detail.

Inode block corruption. Inodes are one of only two F2FS data structures that are protected by checksums, yet their corruption can still create severe problems. One such scenario arises when the information stored in the *footer* section of an inode block is corrupted. In this case, `fsck.f2fs` will discard the entry without even attempting to create an entry in the *lost_found* directory, resulting in data loss.

Another scenario is when an inode associated with a directory is corrupted. Then all the regular files stored inside that directory and its sub-directories are recursively marked as *unreachable* by `fsck.f2fs` and are eventually moved to the *lost_found* directory (provided that their inode is valid). However, we observe that `fsck.f2fs` does not attempt to recreate the structure of sub-directories. It simply creates an entry in the *lost_found* directory for regular files in the sub-directory tree, not sub-directories. As a result, if there are different files with the same name (stored in different paths of the original hierarchy), then only one is maintained at the end of the recovery procedure.

Checkpoint corruption. Checkpoints are the other data structure, besides inodes, that is protected by checksums. We observe that issues only arise if both copies of a checkpoint become corrupted, in which case the file system cannot be mounted. Otherwise, the uncorrupted copy will be used during the system's `mount` operation.

Superblock corruption. While there are two copies of the superblock, the detection of corruption to the superblock relies completely on a set of sanity checks performed on (most) of its fields, rather than checksums or comparison of the two copies. If sanity checks identify an invalid value, then the backup copy is used for recovery. However, our results show that the sanity checks are not capable of detecting all invalid values and thus, depending on the corrupted field, the reliability of the file system can suffer.

One particularly dangerous situation is a corruption of the *offset* field, which is used to calculate a checkpoint's starting address inside the corresponding *segment*, as it causes the file system to boot from an invalid checkpoint location during a `mount` operation and to eventually hang during its `unmount` operation. We filed a bug report which has resulted in a new patch that fixes this problem during the operation of `fsck.f2fs`; specifically, the patch uses the (checksum-protected) checkpoint of the system to restore the correct value. Future releases of F2FS will likely include a patch that enables checksums for the superblock.

Another problem with superblock corruption, albeit less severe, arises when the field containing the counter of supported file extensions, which F2FS uses to identify *cold data*, is corrupted. The corruption goes undetected and as a result, the corresponding file extensions are not treated as expected. This might lead to file system performance problems, but should not affect reliability or consistency.

SIT corruption. SIT blocks are not protected against corruption through any form of redundancy. We find cases where the corruption of these blocks severely compromises the consistency of the file system. For instance, we were able to corrupt a SIT block's bitmap (which keeps track of the allocated blocks inside a *segment*) in such a way that the file system hit a bug during its `mount` operation and eventually, became unmountable.

Recently, a series of patches has been introduced to increase the sanity checks performed every time a SIT block is read from the underlying storage device. The developers have implemented more thorough and strict check statements, but still, these sanity checks are not capable of capturing every single invalid value.

NAT corruption. This data structure is not protected against corruption and we observe several problems this can create. First, the node ID of an entry can be corrupted and thus, point to another entry inside the file system, to an invalid entry or a non-existing one. Second, the block address of an entry can be corrupted and thus, point to another entry in the system or an invalid location. In both cases, the original entry is eventually marked as *unreachable* by `fsck.f2fs`, since the reference to it is no longer available inside the NAT copy, and placed in the *lost_found* directory. As already mentioned, files with identical names overwrite each other and eventually only one is stored inside the *lost_found* directory.

Recently, the developers have implemented a few patches to increase the level of check operations every time a NAT block is read from the underlying storage device. The new check operations verify that both the node ID and the block address of a NAT entry actually point to a valid entry and a valid location inside the file system respectively. Nonetheless, these patch files are not able to capture the case where a NAT entry becomes corrupted, but still points to a valid location inside the file system (i.e., the boundaries of the systems are not violated).

Direct/Indirect Node corruption. These blocks are used to access the data of large files and also, the entries of large directories (with multiple entries). Direct nodes contain entries that point to on-disk blocks, while indirect nodes contain entries that point to direct nodes. Neither single nor double indirect nodes are protected against corruption. We observe that corruption of these nodes is not detected by the file system. Even when an invocation of `fsck.f2fs` detects the corruption, problems can arise. For example, we find a case where after the invocation of `fsck.f2fs` the system kept reporting the wrong (corrupted) size for a file. As a result, when we tried to create a copy of the file, we received a different content.

Directory entry corruption. Directory entries are stored and organized into blocks. Currently, there is no mechanism to detect corruption of such a block and we observe

numerous problems this can create. For example, when the field in a directory entry that contains the length of the entry's name is corrupted the file system returns garbage information when we try to get the file's status. Moreover, the field containing the node ID of the corresponding inode can be corrupted and as a result point to any node currently stored in the system. Finally, an entry can "disappear" by storing a zero value into the corresponding index inside the directory's bitmap.

In the last two cases, any affected entry is eventually marked as *unreachable* by `fsck.f2fs`, since their parent directory does no longer point to it. As already mentioned, files with identical names overwrite each other and eventually only one is stored inside the *lost_found* directory.

Shorn Write + Program Read.

The results when a program reads a block previously affected by a shorn write are similar to those for corruption, since shorn writes can be viewed as a special type of corruption. The only exception is the superblock, as it is a small data structure that happened not to be affected by our experiments.

Shorn Write + Fsck.

Directory entries and shorn writes. Blocks that contain directory entries are not protected against corruption. Therefore, a shorn write goes undetected and can cause several problems. First, valid entries of the system "disappear" after invoking `fsck.f2fs`, including the special entries that point to the current directory and its parent. Second, in some cases, we additionally observed that after re-mounting the file system, an attempt to list the contents of a directory resulted in an infinite loop. In both cases, the affected entries were eventually marked as *unreachable* by `fsck.f2fs` and were dumped into the *lost_found* directory. As we have already mentioned, files with identical names in different parts of the directory tree conflict with each other and eventually, only one makes it to the *lost_found* directory. In some cases, `fsck.f2fs` is not capable of detecting the entire damage a shorn write has caused; we ran into a case where after remounting the file system, all the entries inside a directory ended up having the same name, eventually becoming completely inaccessible.

Bugs found/reported.

We have filed two bug reports related to the issues we have identified around handling corrupted data and one has already resulted in a fix [48, 50]. Moreover, we have reported F2FS's failure to handle write I/O errors [49].

2.5.4 I/O Latency Results

We injected varying delays to selected blocks for typical accesses. The delays range from milliseconds to several seconds. We found that the file systems tested were largely able to cope with the increased I/O delay, even for extremely long times (seconds) as long as the request eventually returns successfully. The only side effect is longer wait times observed by the test programs. Note that since we inject the delays at the block layer, it bypasses any error handling mechanisms that may be present in the device driver level (e.g. SCSI). From inspecting the source code, it is possible that the I/O latencies are handled at the device or driver levels and are masked from the upper layers. One example is the error handling mechanisms built into the device firmware and the SCSI driver. These layers could automatically retry failed requests or abort if they timed out, and ultimately present the upper layer with meaningful return values [165]. The file system and user programs react base on the return code, without any information regarding access latencies.

2.6 Implications

• ext4 has significantly improved over ext3 in both detection and recovery from data corruption and I/O injection errors. Our extensive test suite generates only minor errors or data losses in the file system, in stark contrast with [143], where ext3 was reported to silently discard write errors.
• On the other hand, Btrfs, which is a production grade file system with advanced features like snapshot and cloning, has good failure detection mechanisms, but is unable to recover from errors that affect its key data structures, partially due to disabling metadata replication when deployed on SSDs.
• F2FS has the weakest detection against the various errors our framework emulates. We observe that F2FS consistently fails to detect and report any write errors, regardless of the operation that encounters them and the affected data structure. It also does not detect many corruption scenarios. The result can be as severe as data loss or even an unmountable file system. We have filed 3 bug reports; 1 has already been fixed and the other 2 are currently under development.
• File systems do not always make use of the existing redundancy. For example, Btrfs maintains two independent data structures for each directory entry for enhanced performance, but upon failure of one, does not use the other for recovery.
• We notice potentially fatal omissions in error detection and recovery for all file systems except for ext4. This is concerning since technology trends, such as continually growing SSD drive capacities and increasing densities as QLC drives which are coming on the market, all seem to point towards increasing rather than decreasing SSD error rates in the future. In particular for flash-focused file systems, such as F2FS, where for a long time focus has been on performance optimization, an emphasis on reliability is needed if they want to be a serious contender for ext4.

• File systems should make every effort to verify the correctness of metadata through sanity checks, especially when the metadata is not protected by a mechanism, such as checksums. The most mature file system in our study, ext4, does a significantly more thorough job at sanity checks than for example F2FS, which has room for improvement. There have also been recent efforts towards this direction in the context of a popular enterprise file system [94].
• Checksums can be a double-edged sword. While they help increase error detection, coarse granularity checksums can lead to severe data loss. For instance, manipulation of even 1 byte of the checksummed file system unit leads to discard of the entire file system unit in the case for Btrfs. Ideally having a directory or a file system level checksum that discards only 1 entity instead of all co-located files/directories should be implemented. A step in this direction is File-Level Integrity proposed for Android [47, 55]. The tradeoff of adding fine-grained checksums is the space and performance overhead, since a checksum protects a single inode instead of a block of inodes. Finally, note that checksums can only help with detecting corruption, but not with recovery (ideally a file system can both detect corruption and recover from it). These points have to be considered together when implementing checksums inside the file system.
• One might wonder whether added redundancy as described in the Iron file systems paper [143] might resolve many of the issues we observe. We hypothesize that for flash-based systems, redundancy can be less effective in those (less likely) cases where both the primary and replica blocks land in the same fault domain (same erase block or same flash chip), after being written together within a short time interval. Even though modern flash devices keep multiple erase blocks open and stripe incoming writes among them for throughput, this does not preclude the scenario where both the primary and replica blocks land in the same fault domain.
• Maybe not surprisingly, a few key data structures (e.g. the journal's *superblock* in ext4, the root directory *inode* in ext4 and F2FS, the root node of *fstree* in Btrfs) are responsible for the most severe failures, usually when affected by a silent fault (e.g. silent corruption or silently dropped write). It might be worthwhile to perform a series of sanity checks for such key data structures before persisting them to the SSD e.g. during an `unmount` operation.

2.7 Limitations and Future Work

In this section, we discuss the limitations of our work, along with suggestions for future work.

Some of the fault types we explore in our study are based on SSD models that are several years old by now, whose internal behavior could have changed since then. However, we observe that some issues are inherent to flash and therefore likely to persist in new generations of drives, such as retention and disturb errors, which will manifest as read errors at the file system level. The manifestation of other faults, e.g. those related

to firmware bugs or changes in page and block size, might vary for future drive models. Our tool is configurable and can be extended to test new error patterns.

The device mapper module operates at the block layer. Therefore, it is oblivious to any errors and mitigation that occur at lower levels, such as device drivers or the firmware. However, since this is the layer immediately below the file systems, we believe it is the appropriate place to manipulate the I/O requests and inject errors that are based on real SSD faults. This approach is also file system agnostic, allowing us to study multiple file systems simultaneously without modifying their code. It may be useful to inject errors into other parts of the I/O stack to test error propagation and interaction between layers. This will also allow observing each layer's response to errors in isolation.

Additionally, this work can be expanded to include additional file systems, such as XFS, NTFS and ZFS. Another extension to this work could be exploring how file systems respond to timing anomalies as those described in [64], where I/Os related to some blocks can become slower, or the whole drive is slow.

Our framework runs a rich suite of 7000+ test cases. One additional work would be to eliminate test cases between different file system versions using static analysis, and only run tests corresponding to the new on-disk data structure changes.

2.8 Impact

The work presented in this chapter has resulted in reliability discussions across different file systems in the form of Linux mailing list threads and bugs on file systems bug report tool. We have published the work presented in this thesis at the USENIX Annual Technical Conference [78] 2019 and in ACM Student Research competition [73] 2019. Our paper was also one of the 2 papers in the conference (of 27 accepted storage based papers) that was fast-tracked to appear in the ACM Transactions of Storage [79] 2020.

Our work has also resulted in active research in the area of file system reliability and has been referred by other publications. For instance, in [150], Rebello et. *al.* describe how file systems and user space storage applications recover from *fsync()* failures. Their work is complementary to the work presented in this thesis and extends to storage systems beyond file systems. In [46], Domingo et. *al.* create a parallel file system checker to accelerate the file system recovery against errors. Their work is based on our findings of how crucial file system recovery is in the face of underlying media errors. This work has also inspired the design of low overhead file-systems for embedded devices [196] and for reliability characterization in 3D TLC SSDs for large scale storage systems [103]. We believe that with time, the work presented in this chapter will shape other file system reliability studies on a range of next generation of storage stacks.

Chapter 3

Improving Reliability of High Density SSDs using WOM-v Codes

Although there are no textbooks on simplicity, simple systems work and complex don't.

James Nicholas "Jim" Gray

In this chapter, we focus on improving the reliability of SSD hardware in the context of high-density SSDs. In particular, new generations of SSDs offer increased storage density with higher bits per cell storage, but an order of magnitude lower Program and Erase (P/E) cycles. This decreases the number of times one can rewrite on the SSD, and hence, the overall lifetime of the drive.

One way of improving drive lifetime is by applying Write-Once Memory (WOM) codes which can rewrite on top of pre-existing data without erasing previous data. This increases the total logical data that can be written on the physical medium before an erase operation is required.

Traditional WOM codes are not scalable for high density SSDs. This is primarily because traditional WOM codes were designed for media that stored data in the form of bits instead of voltages. Imposing bit-level constraints to media reduce WOM code scalability. Further, such treatment of underlying media is acceptable for media such as punch-cards and tapes, but not for voltage-based media such as SSDs.

In this work we present a novel, **simple** and highly efficient family of WOM codes called WOM-v codes. Our design is generic and could be applied to any N-Level cell drive. In this work, we focus on QLC drives to demonstrate and evaluate the proposed scheme. We first present the theoretical gains that can be achieved on QLC flash. Next, we implement WOM-v codes in Linux LightNVM and FEMU. We then describe 2 novel opti-

mizations - GC_OPT (Optimized Garbage Collection) and NR (No-Reads Configuration) to further improve reliability and performance of the underlying flash media. A careful evaluation, including micro-benchmarks and trace-driven evaluation, demonstrates that WOM-v codes can reduce the erase cycles for QLC drive by 4.4x-11.1x for real world workloads with minimal performance overheads resulting in improved QLC SSD lifetime.

3.1 Introduction

Flash-based Solid State Drives (SSDs) offer a faster alternative to Hard Disk drives (HDDs), but have a major limitation: unlike HDDs, where previously written data is over-writable, a flash cell needs to be erased before it can be programmed, and each erase operation causes wear-out that reduces a cell's lifetime. Older generations of flash were based on single-level cells (SLC), which store only a single-bit in a cell and can typically tolerate around one hundred thousand program and erase cycles before wearing out. However, to keep up with the increasing demand for storage capacity, newer SSD generations enabled storing more bits in each flash cell, and at the same time reduced the size of each cell. Such SSDs are called multi-bit cell SSDs. Recent work [173] shows that with each additional bit stored in one SSD cell, the number of erase cycles that the SSD can endure reduces by 3 times with each new SSD generation. Figure 3.1 illustrates the problem

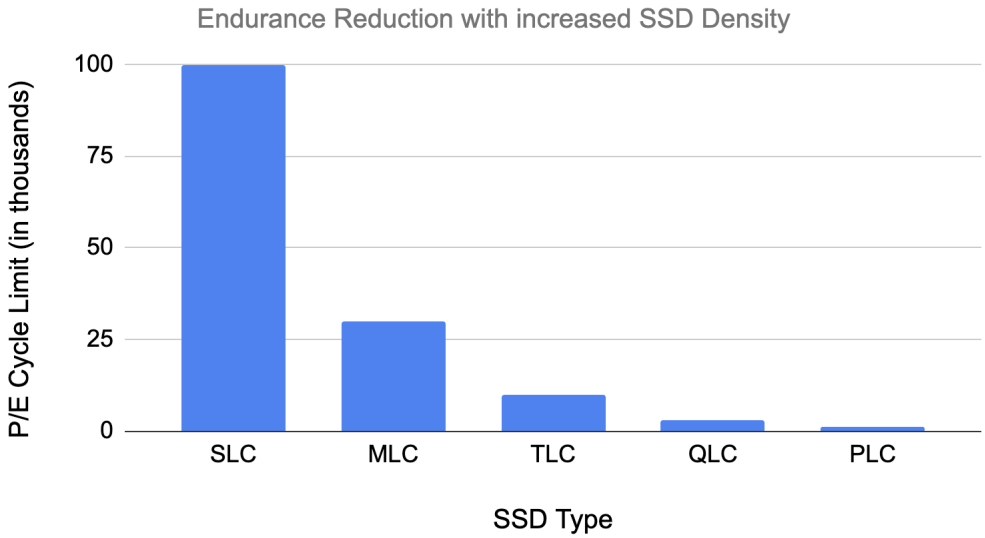


Figure 3.1: *SSD Endurance with increased density. As the density of SSD increases, the number of times the device can be erased reduces by a factor of 3X. PLC drives are in experimental stage [180] and their P/E cycle limit is based on projected values.*

based on recent projections. Flash based on Multi-level Cells (MLC) and Triple Level Cells (TLC), which are common nowadays, can tolerate a significantly smaller number of

P/E cycles. Recently QLC drives have started being deployed in datacenters. Even more worrisome is a look into the future with PLC drives, which might see P/E cycle limits drop to tens or a few hundred. To make high-density SSD drives usable beyond archival applications, it is paramount to reduce the number of times the storage media needs to be erased. In this thesis, first we introduce the idea f voltage-based WOM-v codes [77] to improve the endurance of multi-bit cell SSDs by enabling overwrites between erases. WOM-v codes present a new way of modeling flash-cells with multiple voltage levels that allows WOM-v codes to scale better for denser flash compared to traditional binary WOM codes. Moreover, WOM-v codes involve just a single table look-up for read and write operations resulting in much smaller performance overheads compared to other codes that allow additional overwrites before erase and are more space optimal, but involve multiple iterations before encoding and decoding is completed and are therefore performance inefficient. Further, WOM-v codes provide a family of codes that can be adapted to the amount of space overheads the underlying storage media can tolerate. In this thesis, we first present some simple back-of-the-envelope calculations based on a simplistic model of an individual cell and show that up to 500% additional writes might be achievable on QLC drives using WOM-v codes. While these back-of-the envelope calculations show great promise for improving SSD endurance, it is not clear how much of these improvements can be achieved in practice. First, in the real world, SSDs have several restrictions while overwriting data. SSD contains multiple erase-blocks containing multiple pages. Writes to erase-blocks are done at page granularity, where pages can be written to in sequential order from the first to the last page. Only a limited number of erase-blocks can be programmed at a time. Once the pages are programmed their voltage level can only be increased further, and hence, in general it is not possible to overwrite new data on them before they are first erased. In order to erase pages in an erase-block while avoiding data loss, the SSD must first relocate all valid pages from target erase-block to another erase-block. Once the valid pages are relocated, the entire erase-block can be erased and is available to be reprogrammed. This "copy before overwrite" step, also called garbage collection, considerably increases the overall writes done to the device, which counters the gains we get from WOM-v codes. The simple estimates provided in our theoretical model ignore the garbage collection and its effect on endurance. Moreover, modern SSDs employ parallelism for higher performance. Multiple erase-blocks are arranged in groups called Erase Units (EUs). Only one EU is active at a time. Incoming data is first buffered and subsequently sharded across all erase-blocks in the active EU. Any performance based metric computation may not capture such nuances involved in writing data to a shared buffer that requires the use of locks to prevent race conditions, nor the gains due to striping the data across different parallel units. The work in [59] does not provide any performance evaluation. Finally, real world workloads vary in their storage access patterns. The access pattern determines the amount of garbage collection in the device. A thorough analysis of WOM-v codes over multiple real world workloads is required to assess the practical gains of WOM-v codes. The contribution of our work is to demonstrate that WOM-v codes have

real gains in practice. Toward this end, we present the first work to our knowledge that provides a detailed design, implementation and evaluation of Non-Binary WOM codes on next generation dense (QLC) SSD drives to improve SSD lifetimes. Our implementation includes two novel optimizations targeted at high write amplification workloads - GC-OPT and NR-Mode - which significantly reduce erase operations while retaining high performance. Our simulator is open-sourced and can be used as a test-bed to evaluate future WOM code designs on next generation SSDs. We then use the simulator for a detailed evaluation of WOM-v codes using micro-benchmarks and traces and show that even when taking all real world factors into account WOM-v codes still offer significant gains in endurance: we find that WOM-v codes improve the lifetime of QLC flash by 4.4x-11.1x. Moreover, these gains do not come with any significant impact on performance. Counter intuitively, we also show that higher-order WOM-v codes do not provide Erase Cycle reductions if the workload generates high write amplification. Finally, we extend an existing analytical framework to WOM-v codes to derive theoretical bounds on the number of erase operations. We find that when comparing against the WOM-v coding implementation with the optimal scheme of garbage collection, all the existing analytical models, including the one introduced in this work, will only provide upper-bounds on the number of required erase operations, and in many cases these bounds are not tight. Lastly, we show how WOM-v codes can be easily extended to higher density, future generation of SSDs such as PLC SSDs and more sophisticated coding schemes.

3.2 Limitations of traditional WOM codes

data	Gen1	Gen2
00	000	111
01	100	011
10	010	101
11	001	110

Table 3.1: *WOM(2,3)*

data	Gen1	Gen2
00	0000	0111
01	0100	0011
10	0010	0101
11	0001	0110

Table 3.2: *WOM(2,4)*

Historically, Write Once Memory (WOM) Codes were motivated by media, such as punch cards and optical disk, where data is written at the granularity of bits, and once written a bit can only be changed in one direction, e.g. from 0 to 1. Seminal work by Rivest and Shamir [153] presents a way to accommodate multiple writes on such media by encoding x bits of (*data bit sequences*) into a *code-word* of y bits such that a certain number of overwrites in place are possible. This is referred to as a WOM(x,y) code.

We call each time data is transferred to the media for writing as one *Write Cycle* (WC). For example, writing the 4 data bit sequences 01, 01, 10, 10 requires only two writes to the medium (100 and 101), but count as 4 Write Cycles.

The assumption of being able to change a bit in only one direction (from 0 to 1) matches the characteristics of a (single-level) flash cell and inspired prior work [191, 186, 184, 110]

to use WOM codes to increase the lifetime of flash. However, these prior applications of WOM codes have several limitations:

3.2.1 Modelling cells as group of bits

WOM codes, as described before in the context of flash, work at bit-level and assume bits can only be changed from 0 to 1. We call such codes *bit based codes*. Prior work has explored non-binary WOM codes [58] but not in the context of QLC flash. The bit-based model is not a good fit for modern flash, which stores multiple bits in a cell. For example, a QLC cell distinguishes 16 different voltage levels, where each voltage level represents a 4 bit binary code. Any transition between different states of a cell is governed by the current voltage level: reducing the voltage level is not possible without first erasing, but increasing is possible. It is not dependent on the individual bit values of the 4 bit value stored in a cell (represented by the current voltage level).

3.2.2 Scalability

Non-Binary WOM codes have been well studied and optimized for various metrics [58, 17, 53, 56, 24, 35, 95, 70, 185, 167] but we limit extending to a code-word scheme with 4 bits per code-word and 4 code-words per generation each representing a unique data bit sequence to model a QLC cell. The *bit based model* creates unnecessary constraints that limit the usability for QLC flash. In order to show these restrictions, we introduce a naive extension of WOM(2, 3) to a WOM(2, 4) code which encodes any 2 bits of data into 4 *coded bits* to be written in a cell for a QLC drive where all the bit-level constraints are still satisfied. To this end we try and extend WOM(2, 3) to WOM(2, 4) only to find that we are unable to arrange codewords in a way such that bit based constraint can be satisfied. Although WOM(2,4) allows us to write 2 generations before erasure, only 8 out of 16 possible four-bit code-words that could be written in a QLC drive's cell are used in this scheme as we try to maintain the bit-level constraint. This limits the total number of potential generations to 2 instead of 4 generations. The artificial reduction in code-word usability due to bit constraints will only worsen with denser media.

3.3 Solution

SSD cells that store multiple bits have different characteristics than how WOM codes in the past have modeled them. In [110] Margaglia et al. examined possible reprogramming operations between different intermediate states in MLC SSDs. Their observation confirms that some transitions are possible which are not compatible with the bit-level constraints. Moreover, there are cases where the bit-level constraints assume the transition is possible but in practice that transition is not possible due to hardware constraints. In order to design WOM codes that are capable of utilizing the full potential of QLC

drives, we consider a different model for WOM code constraints. Although we evaluate our approach on the most recent and dense drive commercially available, our approach is generic and can be extended to any N-Level Cell drive, and also matches better with the actual underlying constraints of the flash cell hardware to the best of our knowledge.

3.3.1 Voltage Based QLC WOM Code

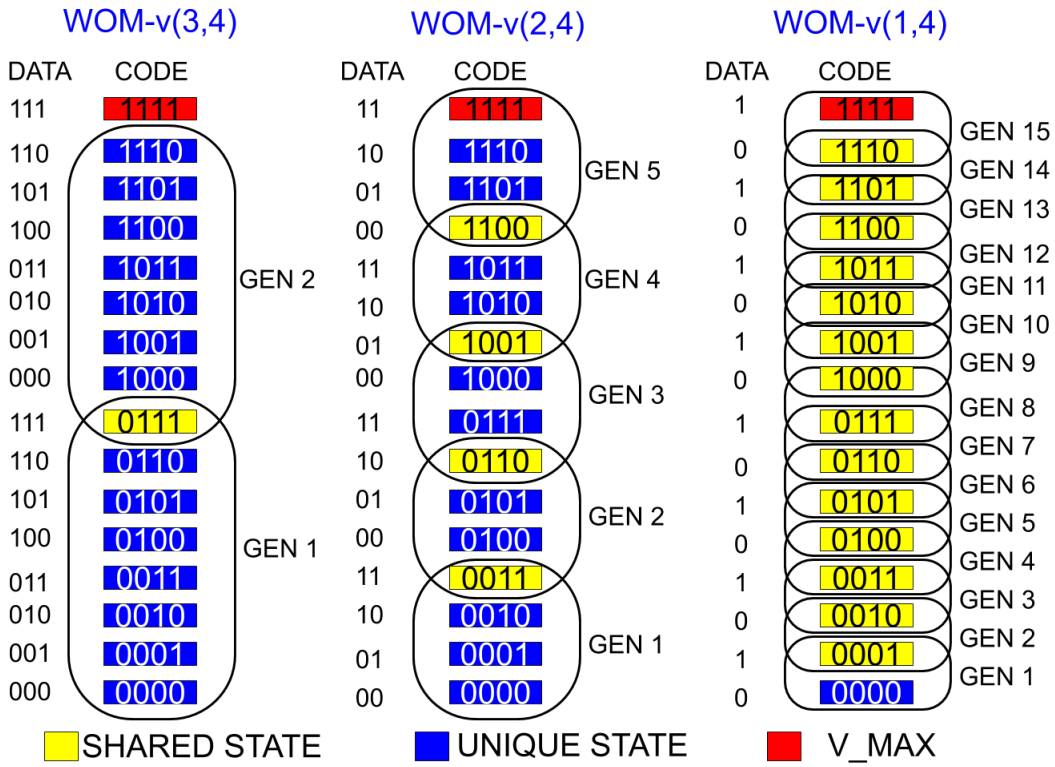


Figure 3.2: *Voltage based codes for encoding 3 2 and 1 data bit(s) into 4 coded bits respectively. Each oval represents a generation. Each generation contains a set of code-words mapped to a set of all possible data bit sequence for the code.*

In this section we change the model by imposing the invariant constraints for WOM coding in terms of voltage levels stored in flash cells rather than encoded bits. In this model the only invariant constraint that should be satisfied in WOM code reprogram operations is that the voltage level in each flash cell can only be increased or kept unchanged before an erasure. We refer to WOM codes designed based on this constraint model as *voltage based WOM codes*, and denote them by WOM-v(k, n) when the coding scheme maps any k data bits to one of 2^n voltage levels stored in the flash cell.

In a QLC drive, each voltage level could be considered as a sequence of 4 coded bits. In Figure 3.2, we present 3 instances of a family of WOM-v codes, WOM-v(3,4), WOM-v(2, 4) and WOM-v(1,4). In summary, a WOM-v(k, n) code is a unique mapping between any

sequence of k data bits and a set of voltage levels (or equivalently their coded bit sequence representations), referred to as *code-words*. A set of consecutive *code-words* that cover all possible *data bit sequence* make one *generation*. Each coding scheme can have GEN_MAX number of generations, which are 2, 5 and 15 in WOM-v(3, 4),WOM-v(2, 4) and WOM-v(1, 4) respectively. All coding schemes have a maximum voltage V_MAX (shown in red) after which the cell needs to be erased. Consider WOM-v(2, 4) code (Figure 3.2 center) where 2 data bits are encoded into 4 code bits. Therefore, we have four possible data bit sequences, indicated by 00, 01, 10 and 11. Each generation has 4 *code-words* mapped to a unique *data bit sequence* within a generation. With *voltage based coding*, the following three optimizations help us accommodate 5 generations for WOM-v(2,4) efficiently:

Code-word Sharing

One feature used in the design of this WOM-v code family is that each pair of consecutive generations are *sharing* one common code-word. A *voltage based model* has a linear structure where we are able to overlap *code-words* between successive generations. This compresses all generations leaving space for additional generations. For example, in WOM-v(2, 4) code, without sharing code-words we would only be able to accommodate 4 generations consisting of voltage levels 0-3, 4-7, 8-11 and 12-15 respectively (not shown). With shared *code-words* between generations, we now can have 5 generations from 0-3, 3-6, 6-9, 9-12, 12-15, (Fig 3.2) with shared *code-words* 3,6,9 and 12 between consecutive generations (shown in yellow). The data bit sequence corresponding to each shared *code-word* is the same in both generations. For example, *code-word* 0011 maps to data bit sequence 11 in both Gen 1 and Gen 2. Note that having larger GEN_MAX translates into more logical data write cycles before erasure. The gain of code-word sharing are much higher in WOM-v(1, 4), where GEN_MAX increases from 8 disjoint generations, to 15 generations with overlapping states, Fig 3.2).

Same Generation Transition

Another feature we suggest in the design of this WOM-v code family is to perform reprogram operations by transition to a higher voltage level within the same generation whenever possible. In other words, in each write cycle, we suggest to simply increase the voltage level in each cell to the lowest voltage level corresponding to the data bit sequence we need to write in that cell which is not below the current voltage level in that cell. For example, consider WOM-v(2,4), if data bit sequences 01, 01, 10 and 00 are written in 4 successive write cycles, we would encode and change the voltage level to 0001, 0001, 0010 and 0100 for the write cycles one through four respectively. The second write does not change the first code-word as the corresponding data bit sequences are the same. While writing the third code-word on 0001, instead of transitioning to 0110 in the next higher generation, we increase the voltage level from 0001 to 0010. Finally, the fourth write cy-

cle involves writing data bits 00 so we overwrite 0010 to 0100. Note that without self generation transition, the *code-word* would have the following order 0001, 0001, 0110 and 1000.

As demonstrated in this example, with the same generation transitions we could potentially achieve a significant saving in voltage level increase during write cycles, and therefore, it enables us to perform more write cycles before we need an erasure. However, this comes with a drawback, which is addressed in the next subsection. The drawback here is that the voltage level stored in different cells in a page would end up being in different generations after a few write cycles. That is due to the fact that the pattern of voltage level increment in each cell depends on the pattern of data bits written in successive write cycles. This is a problem as some cells may no longer be programmable as they have reached the highest voltage level, while others remain underutilized since they still have a few additional write cycles they can accomodate before an erase operation is required. It is important to note that this does not affect correctness of reading the stored data, since the decoding mapping from voltage levels to data bits continues to remain unique.

Reusing underutilized Cells

In this section we present another optimization that further increases the number of writes before an erase becomes unavoidable. Our idea is based on two key observations: a) A page spans many flash cells and as soon as only one of the cells in a page reaches a state where it needs to be erased the entire page becomes unusable for rewriting. b) Flash drives incorporate ECC for each page and this ECC is designed conservatively for the high bit error rates that are expected at the end of a drive's life, i.e. for much of the drive's life (when bit error rates are still low) the ECC could handle significantly larger error rates than what it actually experiences.

Our idea is to allow further rewrites of pages where only some limited number of cells have reached the V-MAX by marking those cells as invalid and let ECC recover the value in those cells. Considering random data bits, it is easy to see the variance in stored voltage levels across different cells in a page increases with the number of generations designed in the WOM-v code. While each WOM-v code certainly guarantee at least GEN_MAX number of possible write cycles in each cell, one could see that for a WOM-v code with larger GEN_MAX, there is a good chance to have majority of the cells still being capable of accommodating more reprograms after write cycle GEN_MAX. In order to enable writing beyond the write cycle GEN_MAX, we suggest to identify the cells not writable anymore by increasing the voltage level in them to V-MAX which would be interpreted as *invalid* for write cycles beyond GEN_MAX. As long as the percentage of such cells remains low in each page, a slightly stronger error correcting code (ECC) could handle these cells as noise, and enable reading stored data from the page.

Current literature on QLC flash drive reliability [173] provides limited insight to the ex-

isting ECC within a QLC drive. Given the increasing trend in error rates with denser media, we hypothesize QLC drive to have provisions for strong ECC mechanisms to correct uncorrectable bit errors. We suggest to use an enhanced ECC mechanism, rather than the already existing ECC embedded in the flash drives to correct raw bit errors in the SSDs. Since invalid cells in our proposed scheme would be identified as cells reaching V-MAX for write cycles beyond GEN_MAX, correcting this type of noise is easier than other types such as *rotten bits* caused by retention or program interference errors. This is because unlike traditional errors or corruptions that require reading parity or checksums for identifying the location of corrupted or errored cell, our approach simply checks for the value in a cell reaching the V-MAX voltage level.

Note that the maximum voltage level is marked as invalidated or EINVAL *after* the number of write cycles become more than GEN_MAX. This gives us a guaranteed GEN_MAX number of write cycles even without using ECC. Since all pages in a block undergo the same number of write cycles before erase, keeping count of write cycles has minimal storage overhead. Using pre-existing flash ECC to correctly reconstruct cells in V-MAX after write cycles reach GEN_MAX helps reprogram under-utilized cells which are still at lower Generations. By correcting cells that cannot be reprogrammed further, we increase the number of write cycles. For example, in WOM-v(1,4), increasing the number of writes helps increase logical space from 15X to 20X with a 4X physical space overhead. This raises the total logical writable bytes from 375% to 500% of the original physical storage capacity of the drive.

Each of the three optimizations - code-word sharing, same generation transition and reusing underutilized cells, may be done for TLC drives as well, but the resultant gains will not be significant. However, as the disks become denser (eg. PLC) which is an upcoming trend for SSDs, our optimizations will reap great benefits while using WOM codes.

3.4 Theoretical Evaluation

Our WOM coding scheme is based on simple table lookups and does not involve large firmware changes or complicated computation. In this section, we provide an analysis of the tradeoffs involved in implementing our scheme for QLC drives.

3.4.1 P/E Cycle versus Physical Space Tradeoff

Each WOM coding scheme offers a trade-off between number of write cycles possible before erase and the physical space overhead. For instance, in WOM(2,3) (Table 3.1), if a user has 1 unit of physical storage, we may write at least 2X logical data before erase is required. However, for each 2-bit data sequence, a 3-bit *code-word* is written, increasing storage overhead by 50%. While this might seem like a large space overhead, let's compare it with a solution that does not use codes, and therefore needs to perform an erase after

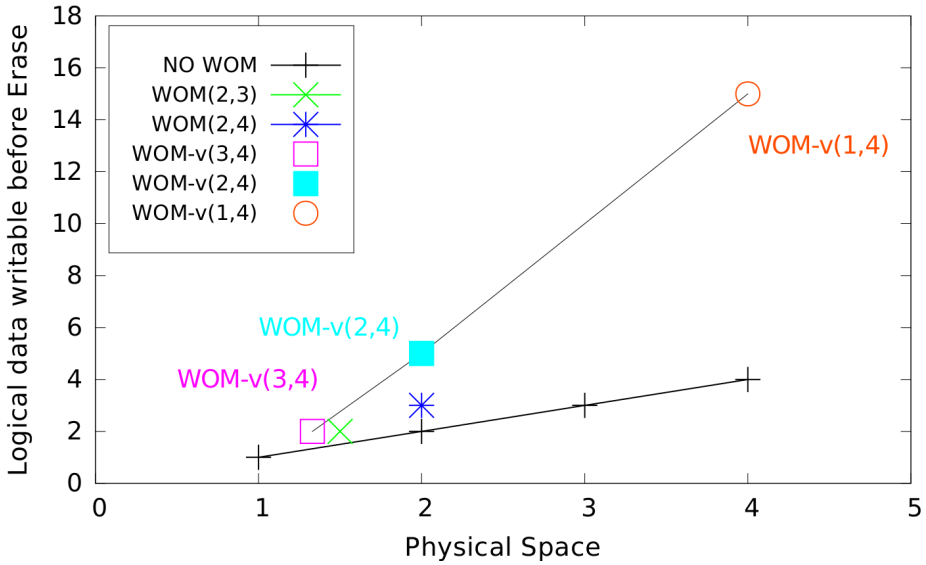


Figure 3.3: *Each point represents physical space required for 1 unit of user data, and corresponding logical writes possible before erase.*

every write. The only way to double the number of writes that is feasible for such a device is to double the physical capacity and spread all writes and erases over twice as many cells. This would be an additional 100% physical storage overhead.

Figure 3.3 shows the increased logical data before an erase versus the physical space overhead. With no coding NO WOM(+), the total amount of logical space written equals the total amount of physical space available. For more logical writes with a constant number of P/E cycles, the physical space required needs to be increased proportionately. For a 2X increase in logical writes using WOM(2,3)(X), we only need 1.5X physical space. Similarly, using WOM(2,4)($\star$), one can write 3X more logical data, with a space overhead of 2X.

For WOM-v codes, although we need 1.33X, 2X and 4X physical space, we are able to write 2X, 5X and 15X amount of logical data for WOM-v(3,4), WOM-v(2,4) and WOM-v(1,4) respectively, before any cell reaches V-MAX. This is a huge gain over existing WOM(2,4) coding scheme that gives us a 50% additional logical space per physical unit. Using WOM-v(1,4) scheme, we get 375% additional logical space per physical unit. The lines joining WOM-v points show that underlying firmware may choose to take a middle-ground and use a combination of multiple codes based on available physical storage.

For example, if a firmware encodes half of the user data using WOM-v(2,4) and the other half using WOM-v(1,4), the overall physical space overhead per 1 unit of user facing physical storage equals $0.5 \times 2 + 0.5 \times 4 = 3$ units. The corresponding logical data that can be written before we need an erase operation would be $0.5 \times 5 + 0.5 \times 15 = 10$ units increasing overall logical space usage of physical media before erase by 333%.

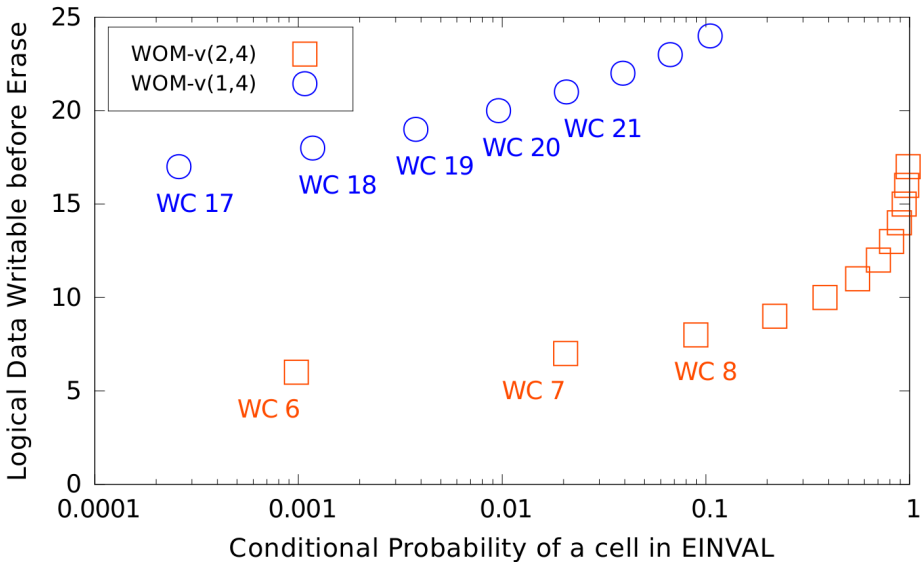


Figure 3.4: *Number of cells that reach EINVAL versus Logical Data Writable before erase. Each point shows the conditional probability of a cell being in V-MAX and the Write cycle (WC). Each additional WC increases Logical data writable before erase by 1. The probability of the cell in EINVAL is 5 and 16 for WOM-v(2,4) and WOM-v(1,4), not shown in figure due to log scale on the X-Axis. The probability is conditional based on a cell having a lower voltage than EINVAL.*

3.4.2 Reprogramming beyond GEN_MAX

Once a cell in a page reaches V-MAX it can no longer be reprogrammed. The number of cells that have reached V-MAX can be easily computed in practice by reading cell values. In order to determine the probability of a cell reaching V-MAX, we use a Markov model considering each *data bit sequence* that is written to be an Independent and Identically Distributed (IID) random variable. At each voltage level, a transition to the same *code-word* or one of the other valid *code-words* takes place with equal probability. Therefore, in the worst case a cell will become unwritable only if each write cycle (WC) writes different data GEN_MAX times. Recall that after WC equals GEN_MAX, we treat V-MAX as EINVAL and treat all cells having EINVAL value unwritable. A coding scheme with larger GEN_MAX has less unwritable cells when WC equals GEN_MAX.

Figure 3.4 shows the amount of logical data that can be written for a given fraction of cells in EINVAL state after each WC for two coding schemes. We observe that for the same percentage of cells (0.02) in EINVAL, we are able to write more logical data beyond each code's respective GEN_MAX with WOM-v(1,4) as compared to WOM-v(2,4). We propose that if the fraction of cells reaching EINVAL beyond GEN_MAX is low (Eg. 0.02) and correctable using ECC, we can enhance the number of logical bytes written from 5X to 7X and from 15X to 20X in WOM-v(1,4) and WOM-v(2,4) respectively. To analyze the total

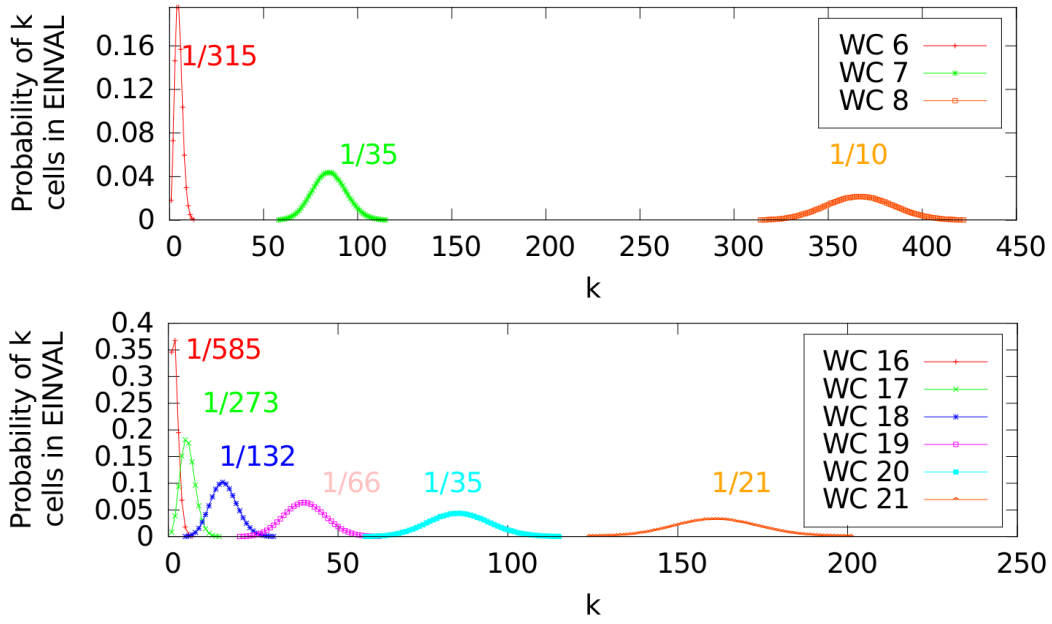


Figure 3.5: *Binomial distribution of the number of cells reaching EINVAL in a 4096 cell page for each write cycle for WOM-v(2,4) and WOM-v(1,4). The X axis denotes the number of cells (k) in a flash page. The Y axis denotes the probability of k cells having EINVAL value. 1/p denotes 1 additional ECC correction bit that needs to be maintained for every p data bits to continue reprogramming underutilized cells WC times.*

number of cells in a flash page that reaches EINVAL after GEN_MAX write cycles, we create a binomial distribution for each WC as shown in Figure 3.5. We consider the flash page size to be 4096. The X axis denotes the number of cells (k) in a flash page (between 0 and 4096). The Y axis denotes the probability of k cells having EINVAL value. For each WC, we provide the minimum correctable error rate (superscript on each curve) that needs to be maintained in flash ECC to correct cells in EINVAL. As the number of WC increase, the total number of cells in EINVAL in a page increase, which requires a stronger ECC.

We leave determining the exact value of ECC to the system designer based on the underlying media being used, as the uncorrectable bit error rate varies widely across flash media [173]. Rather, we provide an estimate of the ECC required to recover data in cells that have reached EINVAL after specific number of write cycles. For example, an ECC with error correction capability of 1/35 would help us do WC 7 and WC 20 increasing the logical bytes writable before erase from earlier 250% to 350% for WOM-v(2,4) and from earlier 375% to 500% for WOM-v(1,4) respectively.

3.5 Limitations of a Theoretical Model

Although the theoretical gains show significant improvement, it is not clear how the theoretical gains can be translated in practice. In the real-world, SSDs have several restrictions while overwriting data.

First, SSD contains multiple erase blocks containing multiple pages. Writes to erase blocks is done at page granularity, where pages can be written to in sequential order from the first to the last page. Only a limited number of erase blocks can be programmed at a time. Unless all the pages within the target erase block are invalidated, the SSD must first relocate all valid pages from target erase block to another erase block. Once the valid pages are relocated, the entire erase block is available to be reprogrammed. This "copy before overwrite" step, also called garbage collection, considerably increases the overall writes done to the device, which counters the gains we get from WOM-v codes. The evaluation in Section 3.4 falls short as it completely ignores the garbage collection workflow in SSDs.

Second, modern SSDs employ parallelism for higher performance. Multiple erase blocks are arranged in groups called *Erase Units*. Only one Erase Unit is active at a time. Incoming data is first buffered and subsequently sharded across all erase blocks in the active erase unit. Any performance based metric computation may not capture such nuances involved in writing data to a shared buffer that requires preventing race conditions using locks, nor the gains due to striping the data across different parallel units. The evaluation in Section 3.4 fails to take performance evaluation into account, and hence needs a real-world flash emulator to evaluate any impact of WOM code on application performance.

Finally, real-world workloads have varying patterns of accessing the SSD device. The block access pattern determines the amount of garbage collection in the device. In practice, real-world workloads from different production datacenters have varying access patterns. A thorough analysis of WOM-v codes over multiple real-world workloads is required to assess the practical gains of WOM-v codes introduced by Section 3.4.

The contribution of the remaining sections is to demonstrate that the codes have real gains in practice. In Section 3.6 we present the first work to our knowledge that provides a detailed design, implementation and evaluation of applying Non-Binary WOM codes on next generation of dense SSD (QLC) drives to improve SSD lifetime. Second, we show that there is a difference in theory and practical reduction in SSD Erase Cycles obtained using WOM-v codes. Moreover, counter intuitively, higher order WOM-v codes do not provide Erase Cycle reduction if the workload generates high write amplification. Third, our simulator can be used as a test-bed to evaluate future WOM code designs on next generation SSDs. Finally, we show how WOM-v codes improve the lifetime of QLC flash by $4.4x - 11.1x$ with negligible performance overheads.

3.6 System Implementation

The theoretical model of WOM-v code provides us with promising results of reducing Erase Cycles and improving SSD lifetime.

With a systems implementation of the WOM-v code, we are able to measure the practical reduction in erase cycles in SSDs. We are also able to measure the impact of input data contents, workload patterns and performance overheads of WOM-v codes that cannot be accurately modeled in theory. This is because SSDs are programmed at the granularity of pages and not cells. Further, SSDs are erased at the granularity of erase units before additional data can be written on them. Ideally, we would want to implement WOM-v code on a real world hardware by manipulating the device Flash Translation Layer (FTL). However, we have the following challenges: 1) SSD FTL is a proprietary closed-source software that is not available for change. 2) An evaluation done on one hardware device configuration may not be conclusive and applicable to future SSD generations.

To address these issues, we implement WOM-v code in the Linux LightNVM Open-Channel SSD Subsystem module [19] which allows making changes in the device FTL. We add 445 LOC in LightNVM module and 220 LOC in FEMU. In order to emulate a QLC device, we extend FEMU, a widely used Flash Emulator [101] for MLC devices to emulate a QLC device. Our extension for QLC support is already merged to mainline FEMU repository [74]. Our WOM-v implementation requires no changes in the application or file system running on top of the device. Although our coding scheme is built within the lightNVM module, in reality, some of the key functionality might be implemented using special purpose hardware. We leave more efficient hardware designs and high performance hardware implementations of encode and decode operations as part of future work.

3.6.1 LightNVM Architecture

LightNVM is a Linux module that exposes the underlying architecture of a real or an emulated NVMe SSD to the host. This helps us to make modifications in the way we write to and read from the device. LightNVM also enables us to control and manipulate the garbage collection scheme and control when an erase operation should be performed on the underlying device. The internal architecture of the device is as shown in Figure 3.6. The two main data structures of a LightNVM module are 1) the shared ring buffer and 2) the Parallel Units.

Ring Buffer

The ring buffer is a circular buffer where data is placed before being written to the underlying device. The device may be accessed either by the application or through the file system as shown in Figure 3.6. Once the ring buffer is full or the user requests a sync operation, the data copied to the ring buffer is striped across different Parallel Units in

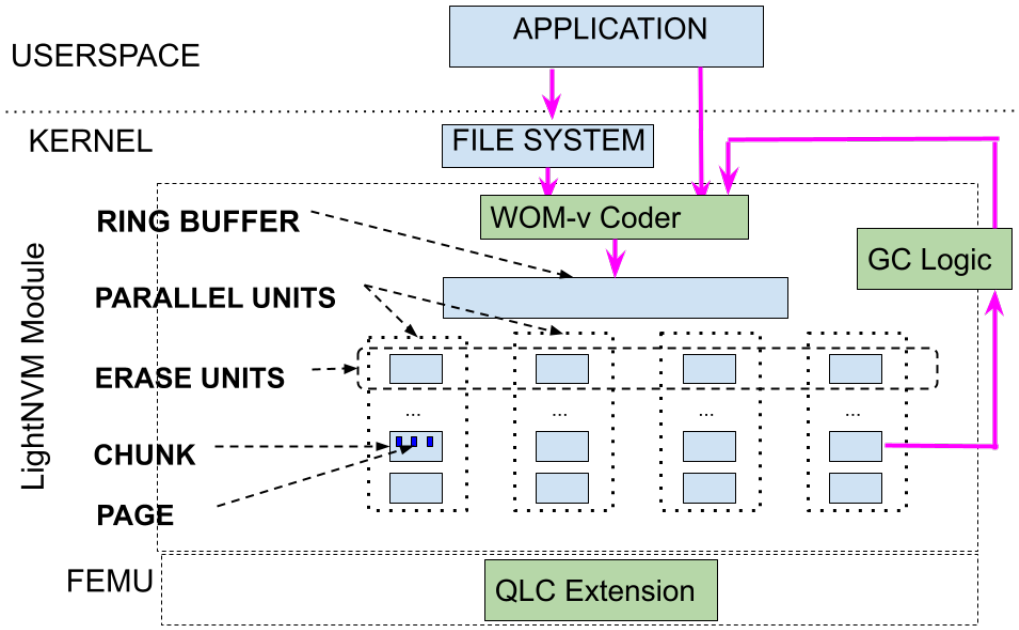


Figure 3.6: *LightNVM Architecture with our contributions in green. The LightNVM and FEMU modules together emulate a host-managed-SSD device. Our WOM-v encoder logic is inside the Lightnvm module. We change the garbage collection workflow to perform erase operations based on WOM-v configuration type. We add QLC-flash support to FEMU.*

a round robin order. The ring buffer is a shared resource between two threads, the *user-write thread* that copies the incoming application/file-system data to the ring-buffer and the *gc thread* that copies valid pages from the underlying device to the ring-buffer during garbage collection.

Parallel Units

Figure 3.6 shows a device with 4 Parallel Units. A Parallel Unit (PU) is an independent unit of storage on the device. Each Parallel Unit is divided into multiple erase blocks or *chunks*. Each *chunk* contains a linear array of *pages* that can be sequentially programmed from the first page to the last page of the chunk. A group of same-sized chunks, one chunk from each Parallel Unit, forms an *erase unit*. Pages within a chunk are sequentially programmed. Pages across chunks within an erase unit are programmed in parallel. As a result, all chunks in an erase unit get filled at the same time. Furthermore, all chunks in an erase unit can only been erased together and hence are garbage collected at the same time in the default setup. At any time, a single erase unit is opened for application writes. The erase unit is closed once all pages in the erase-unit have been programmed.

We can issue 3 types of operations to each parallel unit - 1) page read 2) page write and 3) chunk erase which is issued in parallel to all chunks within an erase unit. Reads on

SSD are 10 times faster than writes, and erase operations are 10 times slower than writes.

All operations are performed sequentially within a Parallel Unit. Two operations on different Parallel Units can be performed in parallel. The number of parallel units on an emulated lightNVM module is configurable. We use the default 4 Parallel Units for all our experiments.

Write and Read Operation

All page writes are staged on the ring buffer. If the data available in the ring buffer is small and a *sync* command is issued by the user, the data to be written is appropriately padded for alignment and striped across Parallel Units. A single erase unit is always kept open for writes. The opened erase unit is called an *active erase unit*. Equal number of pages are simultaneously written to all chunks of an active erase unit, until the last page of all chunks have been programmed. Once the *active erase unit* has been filled, the erase unit is closed and a new erase unit is made *active* and opened for future writes.

All read operations are sent to the device as a block I/O (*bio*) request. The lightNVM module first translates the logical block address (LBA) of the requested page from the *bio* structure to the device Physical Page Address (PPA) using the Logical-to-Physical (L2P) Map. The page contents are then copied from the device PPA to the *bio* request and returned back to the user.

Garbage Collection

A logical page overwrite results in invalidation of the physical page containing the older version of the logical page. A page may also get invalidated when the overlying file system issues a *TRIM* command. All invalidated pages need to be reclaimed from the SSD to make space for additional writes. To free SSD pages occupied by such invalidated pages, a closed erase unit is opened in the *gc-mode* by the *gc-thread* for garbage collection. In the garbage collection phase, all valid pages from a *gc-erase-unit* are copied to the ring buffer. The entire erase unit is then reset to 0, closed and marked erased. This erase unit is returned back to the free pool of erase-units available for the *user-write* thread to be opened for future writes.

LightNVM follows a greedy approach to select an erase unit to be garbage collected. An erase unit with the maximum number of invalidated pages is chosen first. This results in the minimum number of valid pages being copied to the ring buffer, and hence causes the least amount of write amplification for the device. LightNVM reserves an over provisioned space of 11% in order to not run out of space while performing garbage collection.

3.6.2 WOM-v Implementation

To incorporate WOM-v codes in the lightNVM code, first, all writes to the device need to be encoded. Second, all reads issued to the device need to decode previously written data.

Third, the default garbage collection logic needs to be modified. Instead of erasing all the erase units during garbage collection, an erase should now be selectively done based on the state of the pages within an erase unit.

Moreover, for our experiments, the underlying device emulator needs to support next generation SSD devices with QLC or denser flash medium.

Finally, we implement two optimizations, which although do not change the design of WOM codes, help to improve the performance and reduce the overheads of WOM coding. We first present the baseline implementation without these improvements in the next subsection and then present these optimizations in subsection 3.6.4

3.6.3 Baseline Implementation

We add the following components to the lightNVM module (highlighted in green in Figure 3.6): 1) encode and decode logic 2) WOM-v aware garbage collection logic and 3) QLC support for FEMU. Our framework is extensible to emulate future SSDs and future coding schemes.

Write Operation

An application or a file system can submit a write request to lightNVM. All writes are encoded before being written to the drive. By default, a WOM coding scheme first reads the previously written data on the media. This data is encoded and overwritten on the physical page, maintaining the voltage based constraint of the underlying media. Since this default methodology causes increased read amplification, we present a simple mechanism to avoid such reads altogether for WOM-v codes using No-Reads configuration as discussed in 3.6.4.

During a write operation, the ring buffer creates a mapping between the logical block address (LBA) of pages staged in the ring buffer to the destination physical page address (PPA) of the pages on the device before writing the pages to the device. We intercept all writes at this stage and apply the following transformation: First, we read preexisting encoded data in the PPA of all pages being written. Next we encode incoming pages using the preexisting data. The encoding scheme is straightforward and involves a simple lookup in the static WOM-v(k,N) encode table shown in Figure 3.2. Finally, we write the new encoded pages to the device PPA on the drive.

For a WOM-v(1,4) coding scheme, each 4KB incoming write is encoded and stored in 4 x 4KB physical of a QLC page on the device. Similarly, for a WOM-v(2,4) coding scheme, each 4KB incoming write is encoded and stored in 2 x 4KB physical pages on the device. To reduce the performance overheads of additional page writes, we increase logical page size at which application page is sent to the device after an encode operation to 16KB and 8KB for WOM-v(1,4) and WOM-v(2,4) configuration respectively. We maintain logical page locality among all encoded pages. i.e. all pages belonging to the original logical

page are encoded into consecutive logical LBAs. We describe the importance of logical page locality during reads in the next sub-section.

Read Operation

In the read workflow, the original read block I/O (*bio*) request from the application is first translated into consecutive encoded LBA address *bio* requests. The consecutive pages read correspond to a single page due to logical page locality. Next, all encoded pages that were read are simply decoded in the read return path. The decoded data is copied to the originally submitted *bio* request structure and can be read by the application with no modifications. The decoding scheme is straightforward and involves a simple lookup in the static WOM-v(k,N) encode table shown in Figure 3.2.

Garbage Collection

With a WOM-v(k,N) coding scheme, we want to reduce the number of times each erase unit is erased. The default garbage collection scheme in the lightNVM module erases an erase unit during each garbage collection cycle. Hence a new strategy is required to decide whether an erase unit needs to be erased during garbage collection.

GC-Logic

When an erase unit is opened for garbage collection by the *gc-thread*, all valid pages are first read from the *gc-erase-unit* and written back to the ring buffer. Once valid pages have been moved out of the gc-erase-unit, WOM-v(k,N) garbage collection erases that erase unit only if any page in the erase-unit has cells set to maximum voltage level which needs to be corrected using page-ECC as discussed in Section 3.2.

Using Page ECC to extend WOM-v(k,N) gains

The existing page ECC in the Out of Band (OOB) region [83] of each page is capable of correcting a limited number of erroneous bits in each page. We use a fraction of this existing page error correction capability, referred to hereafter as *ECC_threshold*, to recover the bits we can not store in a limited number of cells in a erase unit as they reach their maximum voltage level. As a result we determine whether an erase-unit should be erased or not during garbage collection based on the number of cells reaching the maximum voltage level in each page in that erase-unit. In the WOM-v(k,N) implementation, the *gc-thread* first checks if any page in the erase unit has more than *ECC_threshold* cells in the highest generation (*GEN_MAX*) for that WOM coding scheme. If the number of cells in *GEN_MAX* in each page of the erase unit is less than the *ECC_correction* threshold, we do not erase the erase unit, but set all cells in *GEN_MAX* to have a *V_MAX* voltage to mark

them as cells to be corrected using ECC_threshold in a subsequent write operation. When this erase unit is reopened by a *user-thread* all pages in the erase unit are overwritten.

On the other hand, if the number of cells in *GEN_MAX* in any page of the erase-unit are more than the ECC correction threshold, we erase the erase-unit and reset all pages in the erase-unit to 0.

ECC Correction without Error detection

The correction of cells at *V_MAX* in a page during WOM-v(k,N) is easier than conventional error correction of cells that have incurred random bit errors. This is because we already know the position of the cell that needs to be corrected by scanning for cells that have *V_MAX* value. In other words, to correct the missing values from these cells the ECC needs to perform erasure correction rather than error detection and correction used to fix raw bit errors in flash media. Since correcting x erased bits requires half the redundancy compared to detecting and correcting x erroneous bits [122], the ECC needs less redundancy to correct unprogrammable cells in *V_MAX* and it is less likely for the ECC to fail the job.

ECC_Threshold to correct unprogramable Cells

We set the *ECC_threshold* to 3% i.e. we erase an erase-unit that has any page with more than 3% cells in *GEN_MAX*. This threshold is chosen based on theoretical evaluation in 3.4 where 7 and 20 generations could be written with 1% *ECC_threshold* for WOM-v(2,4) and WOM-v(1,4) configurations respectively. Recent TLC flash devices have reported 7% ECC in each page [135]. We predict higher ECC being reserved for QLC and future generation drives.

It is also important to note that using ECC is not a necessity for using WOM-v(k,N) codes. As the flash device gets older and more ECC is needed for correcting Raw Bit Error Rates (RBER) in flash medium, the device may switch to a Non-ECC supported WOM-v(k,N) configuration with slightly reduced gains proposed in 3.4.

3.6.4 WOM-v Optimizations

We identify two optimizations to the baseline implementation of WOM-v(k,N) codes. First, we present `GC_OPT Mode`, a novel methodology for garbage collection in WOM-v(k,N) configuration that improves SSD endurance considerably by delaying valid page rewrites during garbage collection. Second, we present `NR Mode` WOM coding scheme, a technique to perform encode operations without reading the previous state of the flash drive that eliminates read amplification during writes.

GC_OPT Mode

The garbage collection thread relocates all valid pages in a previously active erase-unit to the current active erase-unit. This is followed by erasing the previous erase-unit in the NO_WOM mode. With WOM-v(k,N) codes using the conventional garbage collection scheme, relocation may or may not be followed by erase, based on the state of the cells in the page. We argue that using WOM-v(k,N) codes it is possible to delay relocating valid pages to another erase-unit. This delays relocating valid pages until all pages in the flash block contain either valid data or are no longer programmable. Instead, we skip writing to pages that have valid data or cannot be programmed since their cells are in `GEN_MAX`. Since pages can be overwritten without erase with WOM codes, one can continue reprogramming invalidated pages of an erase-unit as long as the cells in those pages are yet in one of the generations bellow `GEN_MAX`, and hence are programmable with WOM codes. This helps first to eliminate extra writes during garbage collection, and second to fully utilize the reprogramming capacity of all pages in the erase-unit before erasing it. However, as the number of programmable pages in the erase-unit decrease (by accumulating more valid or unprogrammable pages over multiple cycles) the performance overhead of recycling such erase-units without erasing them increase and hence we can consider a minimum threshold for the number of programmable pages in the erase-unit for reusing it without erasure. As we will demonstrate in Section 3.7.6, the gains achieved using this technique is promising.

NR Mode

As described in Section 3.6.3, the baseline implementation of WOM-v(k,N) codes by default read the previous content of a page to encode and create next cell values for reprogramming the page. This read-before write, which adds performance overhead, is necessary in WOM codes. However, in WOM-v(k,N) codes, this is only necessary for performing the same generation transition optimization, discussed in Section 3.2, and can be skipped if we always move to the next generation in each write cycle. This is because the mapping of the voltage levels to information content in each generation is unique and independent of the codeword of the previous write cycle. We refer to such implementation as NR (No-Read) mode.

In NR mode, ideally we can keep track of the generation of the last write cycle for each page in the active erase-unit as the metadata, and simply encode and write the new content for all cells in each page based on the user data using the appropriate generation. This enforces all cells in the page to reach the `GEN_MAX` together. The meta data for all pages of the active erase-unit can be kept in the memory and once we close the active erase-unit this metadata can be written in the over provisioned space. As we open a new active erase-unit, the metadata describing current generation of all pages in that erase-unit will be loaded in the memory. Note that this still allows each page in the erase-unit to be reprogrammed independent of the other pages in that erase-unit, and will not

impose any further restrictions on the programability of pages in the active erase-unit. However, it adds some extra storage and performance overhead footprint to keep track of the generations of all pages in the drive adding to already high space requirements of WOM-v(k,N) codes.

In our current implementation of NR mode, we optimize keeping pages in a single generation by reducing the metadata tracking to a single generation number for the entire erase-unit. This needs negligible space and minimal metadata tracking for active erase-unit. One side-effect of this optimization is that it enforces all pages in each erase-unit to use the same generation for each subsequent write cycle. Therefore, we might end up having pages in generations below the current generation of the erase-unit if they were not programmed in one or more previous write cycles since they contained valid page in that write cycle. In the subsequent write cycle, we program such pages to the generation dictated by the generation number of the erase-unit possibly skipping one or more intermediate generations.

We observe that this optimization significantly improves the performance overheads of write operation of WOM-v(k,N) codes. Note that this performance gain is achieved at the cost of reducing the total number of possible write cycles before erasure as we will not be able to perform same generation transition. We discuss the tradeoffs between improved performance gains and reduced endurance gains using NR mode optimization in detail in Section 3.7.4.

Adding QLC Support to FEMU

We use FEMU [101] to emulate the underlying SSD media. FEMU emulates the SSD in main-memory and adds predictable I/O latency to each I/O request to mimic a real Open Channel SSD device. In order to read or write a page, a specific amount of reference voltage needs to be used to access the page. For high-density flash such as QLC drives, the number of reference voltage applied is higher than lower density drives. [90].

The main challenge in using FEMU for new generation SSD device emulation is that the existing FEMU emulator only supports an MLC SSD with 2 page levels. A page can either be an Upper or a Lower page with write latency of 850μs and 2300μs and the corresponding read latency of 48μs and 64μs respectively. FEMU also adds a constant NAND read, write and erase latency of 40μs, 200μs and 2ms to each read, write and erase I/O request respectively.

We modify the default MLC configuration of FEMUs page layout in each chunk. Instead of having alternating upper and lower pages with varying latency in each chunk, we extend FEMU for QLC emulation: Each chunk in a QLC device has alternating Lower(L), Center-Lower(CL), Centre-Upper(CU), Upper(U) pages. A write latency of 850μs, 2300μs, 3750μs and 5200μs, and read latency of 48μs, 64μs, 80μs and 96μs is applied to L,CL,CU and U pages respectively based on the number of reference voltages [90] required to read a specific page type.

Testbed for Future SSDs and Coding Schemes

Our WOM-v simulator is generic and can be used as a testbed for denser SSD or higher order coding schemes. In order to add a new WOM-v(k,N) coding scheme, first, the user has to provide a simple lookup table mapping each data word to a code word similar to WOM-v tables shown in Figure 3.2. Second, the user optionally sets an *ECC_threshold* value. Finally, The user specifies the latency of additional page levels for next generation SSDs in FEMU.

Our emulator can also be used standalone without any coding scheme. We open-source our generic N-LC Simulator for more advanced coding and next generation SSD research [74].

3.7 Evaluation

In order to correctly estimate the gains of WOM-v(k,N) codes on QLC drives, it is important to evaluate the endurance gains and performance tradeoffs associated with using WOM-v(k,N) codes. In particular, WOM-v(k,N) code reduces the number of erase cycles (EC) the drive endures over its lifetime and therefore leads to improved flash endurance. However, WOM-v(k,N) also introduces space amplification during writes. Further, both read and write workflows in WOM-v(k,N) codes introduce read amplification for the device.

The goal of this section is to use our implementation to evaluate the EC reduction and the impact on performance for both micro-benchmarks and real world workload traces.

Since WOM-v(k,N) requires more space than NO_WOM configuration, there are two ways to compare them. The first option is to keep the size of the user facing logical address space constant, and increase the physical space allocated to WOM-v(k,N) configuration by a factor of N/k. For a fair comparison of EC reduction per hardware chip, we have to divide the resultant number of ECs encountered by NO_WOM code by N/k to account for additional space provided to the WOM-v(k,N) scheme.

A second more practical approach to evaluate SSD hardware is to keep the physical capacity of the hardware constant, and reduce the logical block address by a factor of N/k for WOM-v(k,N) coding scheme. For example, we use the same 4GB physical space in NO_WOM and WOM-v(k,N) configurations. For NO_WOM, the entire 4GB logical address space is exposed to the application, while for all modes of WOM-v(2,4) codes (i.e, WOM-v(2,4), WOM-v(2,4)-GC-OPT and WOM-v(2,4)-NR) the range of logical address space on the same 4GB drive is reduced to a 2GB logical space, and similarly for all modes of WOM-v(1,4) codes, the same drive is exposed as a 1GB logical space. We assume the workload running on such a drive to have a logical space of 1GB.

Under the same workload this setup clearly leads to have more empty space for NO_WOM configuration which acts as extra over provisioned space to reduce garbage collection overheads. However, we show that for the same physical device - WOM-v(k,N)

is able to successfully reduce the number of ECs in all our experiments with minor performance overheads.

In the rest of this section, we compare the default SSD configuration (NO_WOM) with baseline implementation of WOM-v(k,N) codes, garbage collection optimized (WOM-v(k,N)-GC_OPT) and performance optimized no-reads mode (WOM-v(k,N)-NR) implementations.

3.7.1 Micro Benchmarks

We use micro-benchmarks to study the impact of specific access patterns and data-block contents, on WOM-v(k,N) codes. To evaluate this impact, we use micro-benchmarks. Each micro-benchmark writes 25GB data to 1GB physical emulated drive containing 4 Parallel Units and 160 erase units.

Effect of change in data buffer contents

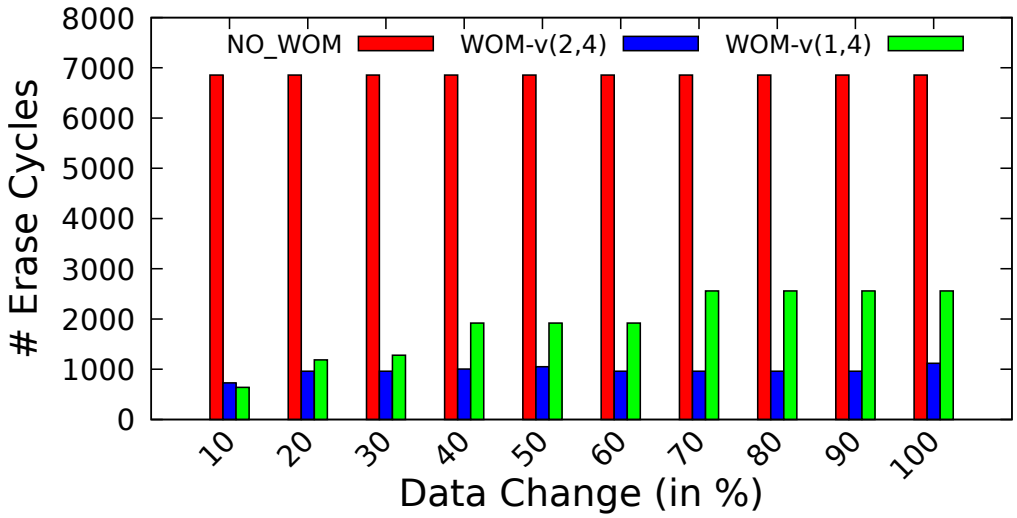
The performance of baseline WOM-v(k,N) codes depend on the data contents of the block that is overwritten to the existing data. If blocks typically get overwritten with similar content, same generation transitions are more likely to take place as compared to overwrites with drastically different data.

In this micro-benchmark, we fill the drive sequentially so that no garbage collection is invoked. Once the drive is full, we flip a fraction of all bits in the data buffer, and fill the entire drive again with this partially modified page. We continue this operation multiple times modifying a fixed amount of data buffer contents and measure the number of erase units erased as the rate of data buffer change increases.

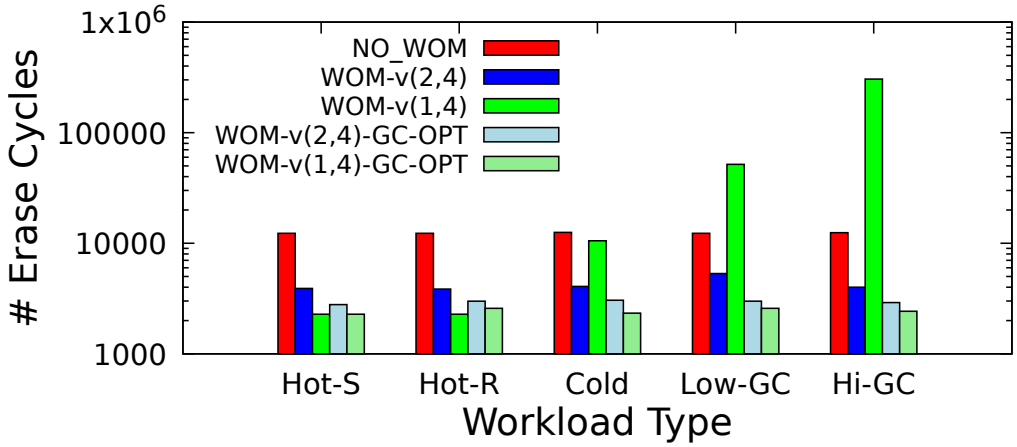
Figure 3.7a shows the results of varying data buffer contents during writes. We note that for all types of data buffer contents, WOM-v codes reduce the number of erase units erased in the default NO_WOM configuration. However, the rate at which a cell reaches the maximum voltage level will be slower when there is smaller rate of data change between subsequent writes. For workloads that have a higher amount of data change, the maximum voltage level will be reached faster, and so the EC gains will be much lower. Unlike WOM-v codes, the NO_WOM configuration Erase Cycles remain constantly at the higher end irrespective of the data buffer contents.

Effect of access Pattern

In order to measure the impact of access patterns, we create custom micro-benchmarks that invalidate previously written pages in a specific order which causes varying amount of device write amplification: *Hot-S* keeps updating the same data sequentially, *Hot-R* updates same data in a random order, *Cold* only updates a fraction of pages, *Low-GC* and *High-GC* generate medium and high amount of Garbage collection respectively.



(a) *Effect of rate of data change*



(b) *Effect of access pattern*

Figure 3.7b shows the number of erase units erased and the corresponding to different workload patterns. Across all benchmarks, we observe that WOM-v(1,2) codes significantly reduce erase units. However, with higher number of overwrites due to GC, we see diminishing gains for WOM-v(1,4) code. Finally, GC_OPT mode is able to alleviate the problems incurred and maintain low write amplification overhead.

Hot-S keeps all pages "hot" i.e. uniformly accessed over the course of the benchmark. During garbage collection no pages from the previously written erase unit are garbage collected and hence we do not have any write amplification. We observe that *Hot-R* also does not create any garbage collection. Hence the pattern of write between two workloads does not impact the gains by WOM-v codes if the garbage collection thread

writes minimal or no additional pages to the drive.

For *Cold* benchmark, we observe a slight increase in the total number of erase units erased for the WOM-v(2,4) configuration and an order of magnitude increase in erase unit erases for the WOM-v(1,4) configuration as compared to *Hot-S* and *Hot-R* configurations. This is due to localized writes on only a subset of the device. We also observe that WOM-v(1,4)-GC-OPT and WOM-v(2,4)-GC-OPT continue to maintain lower write amplification for this benchmark as no valid page from hot erase units are relocated and there is almost always a candidate erase unit with at-least a single programmable page.

Low-GC creates a moderate amount of write amplification in NO_WOM configuration. WOM-v(2,4) configuration continues to outperform NO_WOM. But WOM-v(1,4) configuration starts performing poorly as compared to NO_WOM configuration. This is because additional writes generated space amplification in WOM-v(1,4) configuration.Even with Low-GC, since there are continuous page invalidations, both WOM-v(2,4)-GC-OPT and WOM-v(1,4)-GC-OPT continue to reprogram an erase unit without relocating any pages. This keeps the relative write amplification in check.

High-GC benchmarks causes severe write amplification due to high influx of valid pages recycled during garbage collection. This causes WOM-v(1,4) to perform two orders of magnitude worse than NO_WOM. However, Even in High-GC mode, WOM-v(1,4) GC_OPT continues to find erase units that have intermediate pages available for reprogramming, and hence the write amplification remains consistent over the course of the workload run.

Conclusion

We conclude that across all data write patterns, WOM-v codes are highly effective in reducing the number of erase cycles. For workloads where similar or incremental data is overwritten on the device, huge gains are possible.

WOM-v(2,4) codes are highly robust to different kinds of workload patterns. For workloads that exhibit increased garbage collection, WOM-v(k,N)-GC-OPT codes continue to maintain near constant erase cycles even for artificial, extremely-high garbage collection workload.

3.7.2 Real World Workload Traces

Trace Selection

Table 3.3 shows a summary of 844 real world traces from 5 different sources. We shortlist and present 10 write-based traces and 6-read-based traces representing each source. The write-based workloads have a higher number of writes than reads and help us to measure the endurance improvement (Section 3.7.3) and write performance tradeoffs (Section 3.7.4). The read-based workloads help us better understand the impact of WOM-v codes on read performance (Section 3.7.4).

Source	# Traces	Medium	Year
Alibaba [102]	814	SSD	2020
RocksDB/YCSB Trace [171]	1	SSD	2020
Microsoft Cambridge [123]	11	HDD	2008
Microsoft Production [91]	9	HDD	2008
FIU[93]	7	HDD	2010

Table 3.3: *Historical HDD and recent SSD-based block traces*

We selected traces that have at-least 1 million and at most 50 million page writes. We also make sure to select traces that cover a varying number of unique block accesses and varying number of updates per unique block. Most of the traces we chose are from production systems with the exception of the RocksDB and YCSB traces [188]. The reason we included those two benchmark-generated traces is that they were actually collected on SSD-based systems, while the publicly available block traces from production systems all come from HDD based systems (with the exception of the Alibaba trace, which is from a system comprising both HDDs and SSDs, but the trace does not allow us to distinguish requests to SSDs from requests to HDDs).

Trace Reduction

The real-world traces are captured on different types of disks having different storage capacities. FEMU emulates the underlying SSD storage in memory, so the disk size for our simulation is much smaller than the original HDD or SSD on which the trace was collected. To standardize the real-world workload traces for our simulation, we need to pre-process the real-world traces such that the logical block address (LBA) space of the trace can fit into our fixed-sized 16 GB FEMU-based LightNVM emulator. While reducing the traces, we attempt to preserve the following key characteristics:

1. The distribution of page update frequencies in the reduced trace remains the same.
2. The relative order of page accesses in the reduced trace remains the same as in the original trace.
3. The number of pages requested in each block I/O captured by the original trace remains the same in the reduced trace.

One key observation we make is that all logical pages written to an SSD are internally buffered in the SSD's ring buffer and subsequently mapped to physical pages across different parallel units. The actual numbering of logical pages will not affect how they are treated by the device. For example, if we renamed logical page 1 to page 1,000 and the original page 1,000 to page 1 the resulting trace would merit the same drive behavior. (This is different from a hard drive which will, for example, try collocate pages that are

close in LBA space to areas that are physically close on the disk.) We use this property to select a subset of logical block addresses (LBAs) from the original trace maintaining the relative order of page accesses and map it to a reduced trace that fits inside our underlying SSD emulator configuration.

We propose two types of reduction techniques based on the way in which hot and cold pages are distributed across the entire trace, segment-based reduction and subset-based selection:

Segment Based Reduction: A key property we need to maintain in the reduction process is the distribution of the update frequencies of pages (hot pages versus cold pages), as it will significantly impact SSD operations, in particular garbage collection and therefore write amplification. We note that if the distribution of the page update frequency is similar for different segments of a trace then we can reduce the trace by choosing just one segment of the trace (as this will be representative of the larger trace).

We analyze the traces and observe that for many traces the page update frequency distribution is stable across the trace. This includes for example the Microsoft and FIU traces. We therefore propose to reduce these traces by choosing a segment of the original trace, where we choose the length of the segment such that the number of unique logical pages accessed in the segment is equal to the total number of logical pages that can be stored in the emulated device. Figure 3.7 illustrates this process, where the greyscale of a page indicates its hotness. As we show later in the section, the reduced trace preserves the page update frequency distribution of the original trace.

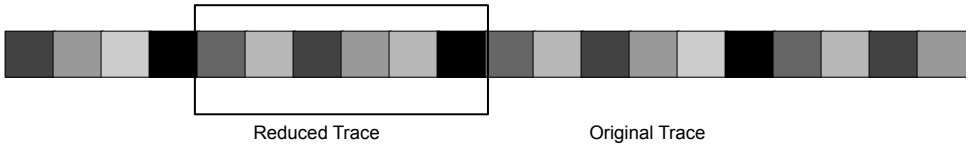
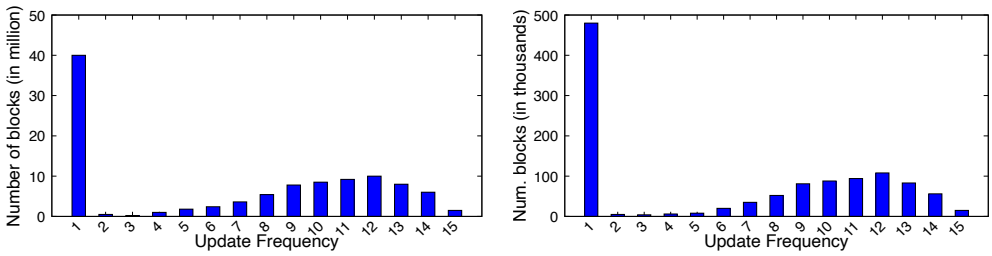


Figure 3.7: *Segment-based reduction: This method is suitable where the update frequency distribution is stable across different areas of a trace and it simply selects a contiguous subset of the original trace. (In the illustration, the darker a page the hotter it is.) The reduced trace is representative of the temperature distribution of the original trace.*

Subset-based Selection: Some real-world traces do not have an update frequency distribution that is stable across the trace. For these traces, the trace's page access frequency distribution cannot be properly captured if we limit ourselves to a continuous segment of the trace.

To reduce such traces while ensuring the same update frequency distribution as the original trace we specifically pick a subset of the LBAs accessed in the trace, such that the update frequency distribution of this subset of LBAs is the same as the update frequency distribution of the entire LBA space. We then include in the reduced trace only requests



(a) original trace update frequency distribution **(b)** reduced trace update frequency distribution

Figure 3.8: Alibaba-4 trace update frequency distribution for the original and the reduced trace. Visually, both traces appear similar.

for LBAs that are part of this subset. We refer to this as *subset-based selection* and implement it as follows:

First, we generate a mapping between all update frequencies **U** and the unique page numbers that are present in the real-world trace. We create a list of these pages and sort them by their update frequency.

Second, we calculate *reduction factor*, which is the ratio of the number of unique logical pages in the original trace to the number of logical pages that can be accommodated in the emulated disk. For example, a reduction factor of ten would mean the original disk has a logical block address range ten times that of the emulated disk.

Third, we filter the blocks from the original update frequency mapping we constructed in the first step and select every other *reduction factor* page from the sorted list we created in the first stage. These list of filtered logical block addresses from the original trace form our candidate logical block address for the reduced trace. Finally, we scan through the original trace and only translate a page to the reduced trace if it lies in the candidate list of pages generated in step three.

Reduced Trace Validation To ensure that our assumption above is correct and to further verify that our range based selection and segmented selection captures a fair sample from the original trace, we create a distribution of the Update Frequency with the Number of Blocks for both the original and reduced traces.

We define p_i as the percentage of the pages in the trace with the update frequency i. We define e_i to be the error between p_i in the original trace and $p_i^{'}$ in the reduced subtrace.

$$p_i = \frac{Number\ of\ pages\ with\ Update\ Frequency\ i}{Number\ of\ pages\ in\ the\ entire\ trace}$$

$$e_i = p_i - p_i^{'}$$

As an example, Figure 3.8 shows the reduction of Alibaba's Device 4 trace. From the reduced trace, we find that the variance is $r^2 = 0.006$. We observe that despite having

only one tenth of the original disk's storage space, the reduced trace still captures the trace collected on the original drive very well. We accept all traces that can be reduced using either range based selection and segment selection technique with the observed variance of $r^2 >= 0.05$..

Page Invalidations through the TRIM Command

In addition to the differences in device capacity discussed above, another issue is that there are no publicly available block traces where TRIM information is available. We therefore only consider a block to be invalidated if it is overwritten. As a result our estimate of gains using WOM-v codes are rather conservative. When considering the addition of pages invalidated with the TRIM command, there will be more opportunity for reprogramming physical pages that contain TRIM'ed logical pages in WOM-v codes. Moreover, we will have lower write amplification induced by garbage collection that will favour WOM-v codes.

Populating Block Data Contents

Finally, block-traces do not have any information about the buffer contents that are to be written to the specified LBAs for a write request. We fill each block with random data. This does not impact NO_WOM configuration, but impacts the WOM-v configurations and higher gains could be possible if the data was more uniform causing less state change.

FEMU+LightNVM Drive Setup for Trace-driven Experiments.

For each workload, we compute the number of unique blocks accessed. We assume the drive is half full, which is a characteristic of real-world drives reported in the field [6]. Hence, for each workload, we create drives that are twice the size of the total number of unique blocks accessed. We use the same sized physical drive for NO_WOM and all variants WOM-v(k,N) configurations. The logical address space for WOM-v(2,4), WOM-v(2,4)-GC-OPT and WOM-v(2,4)-NR is half the logical address space of NO_WOM. The logical address space for WOM-v(1,4), WOM-v(1,4)-GC-OPT and WOm-v(1,4)-NR is one-fourth the logical address space of NO_WOM.

3.7.3 Reduction in Erase Cycles

Figure 3.9 compares the number of erase units erased in default NO_WOM configuration with WOM-v(2,4), WOM-v(1,4), WOM-v(2,4)-GC_OPT and WOM-v(1,4)-GC_OPT configuration. In all cases, WOM-v(2,4) code reduces the number of erase units erased on the underlying device by 29-32%. In most cases, WOM-v(1,4) also reduces the erase units erased on the underlying device with two exceptions - for Microsoft-Production trace, we observe that WOM-v(1,4) code erases more erase units than NO_WOM configuration.

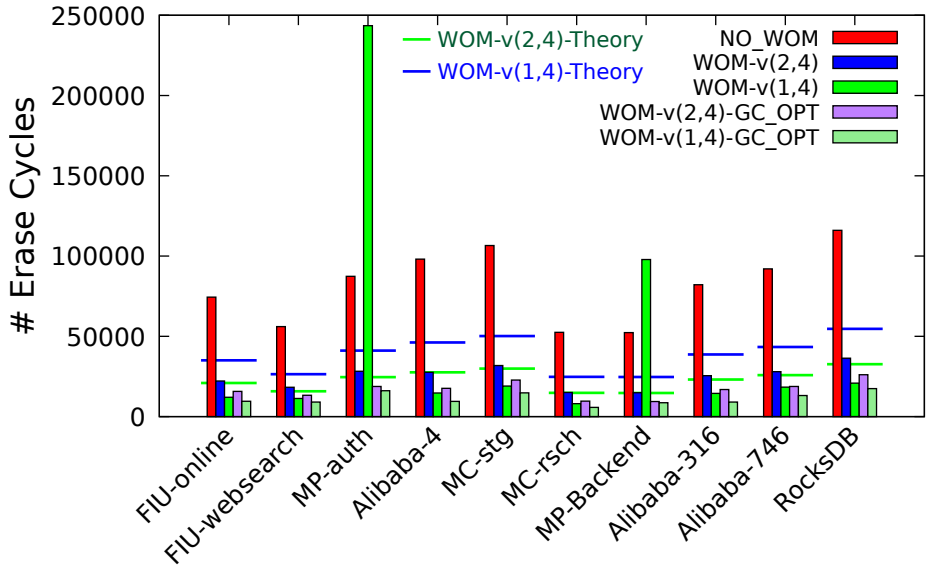


Figure 3.9: *Reduction in Erases using WOM codes.*

This is due to higher write amplification contributed by two factors. First, using higher order WOM-v(1,4) codes with same physical drive leads to logical space reduction by 4x. Second, there is 4 times additional data being encoded and written to the device.

For WOM-v(2,4)-GC_OPT and WOM-v(1,4)-GC_OPT, we observe WOM-v codes are able to reduce the number of erase cycles for all workloads, including the MP-Workloads. The gains are promising - the erase cycles are reduced by 77-83% and to 82-91% of NO_WOM configuration for WOM-v(2,4)-GC-OPT and WOM-v(1,4)-GC-OPT respectively. This is because unlike WOM-v(2,4) and WOM-v(1,4) codes, which relocate valid pages on every garbage collection cycle, GC_OPT mode continues reusing a partially programmable erase block for writes until all pages in the block are entirely unusable or contain 0 invalid pages to overwrite.

We observe that for both MP workloads, the amount of garbage collected pages are significantly higher as compared to other workloads, the limitation can be successfully alleviated using WOM-v(1,4)-GC-OPT mode by incurring almost 0 garbage collection penalty.

We also provide the theoretical estimate (WOM-v(k,N)-Theory) of WOM codes discussed in Section 3.7.7. We note that the theoretical assumption of uniform invalidations may give significantly different results as compared to the ones achieved on a full system emulated WOM-v(k,N) drive.

In order to understand the effectiveness of the GC_OPT configuration, we introduce a tunable threshold - *minimum programmable page count* to our workload setup. When a erase unit is analysed during the garbage collection phase, we calculate the number of programmable pages in the erase unit. If the number of programmable pages in the erase unit is lower than the *minimum programmable page count*, we relocate all valid pages from

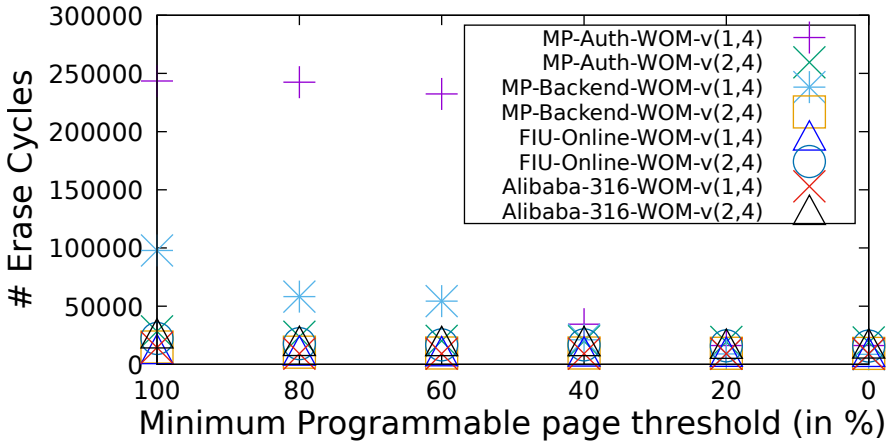


Figure 3.10: *Reduction in Erase Cycles with change in minimum programmable page count.*

the erase unit and then reuse the erase unit. On the other hand, if the programmable pages is higher than the *minimum programmable page count*, we continue to program all pages in the erase unit that are invalidated and still have cells below the maximum generation. For GC_OPT, the *minimum programmable page count* is 0, i.e. we continue reusing a erase unit until all pages in the erase unit are exhausted. For the default WOM-v mode, the *minimum programmable page count* is 100% and defaults to relocating all pages before reprogramming the block.

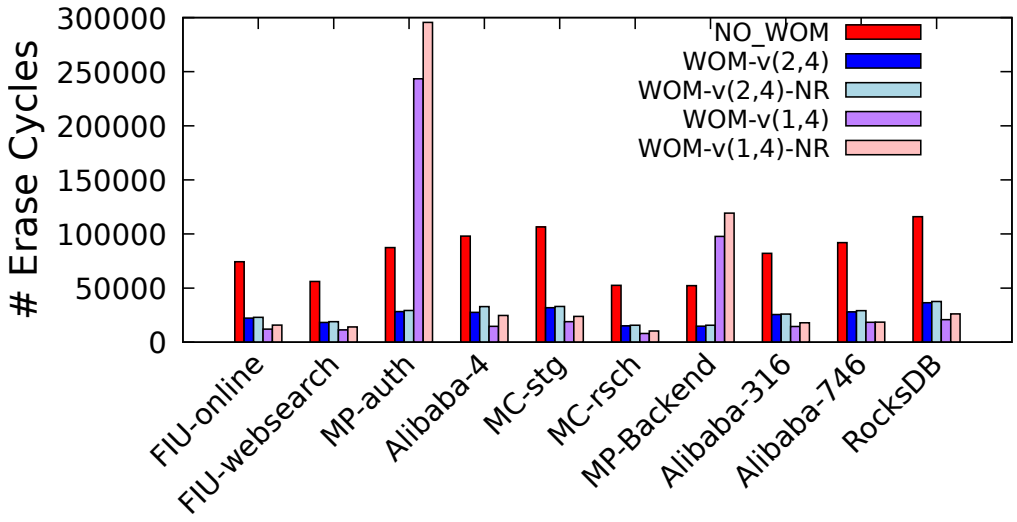
Figure 3.10 shows the resultant erase units erased for the different minimum programmable page count values. We observe that as we reduce the *minimum programmable page count* parameter, the number of erase units erased significantly reduce as an erase block keeps getting reused until all pages in the block are either valid or no longer programmable. There is significantly reduced amount of rewrites of valid pages during garbage collection which improves overall endurance of the drive.

In conclusion, we demonstrate that higher order WOM-v codes such as WOM-v(1,4) codes are not impossible to use in high write amplification scenarios. Their endurance gains can still be leveraged by using GC_OPT mode.

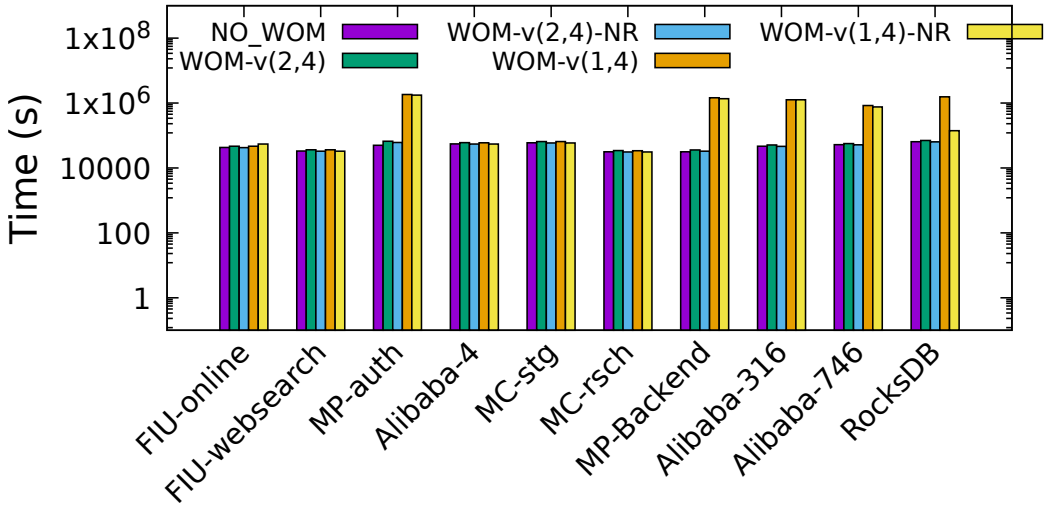
3.7.4 Performance Optimizations

WOM-v(k,N) coding scheme has two performance overheads. First, each page write adds additional (N/k) data to the device causing significant write amplification. To solve this problem, we propose increasing the logical page size to 2x and 4x the size of original logical page size for WOM-v(2,4) and WOM-v(1,4) configurations respectively. We assume the time to program 4K, 8K and 16K pages will be the same.

Second, each page write causes read amplification because previously encoded data needs to be read to know previous state before new encoded data can be created. To solve



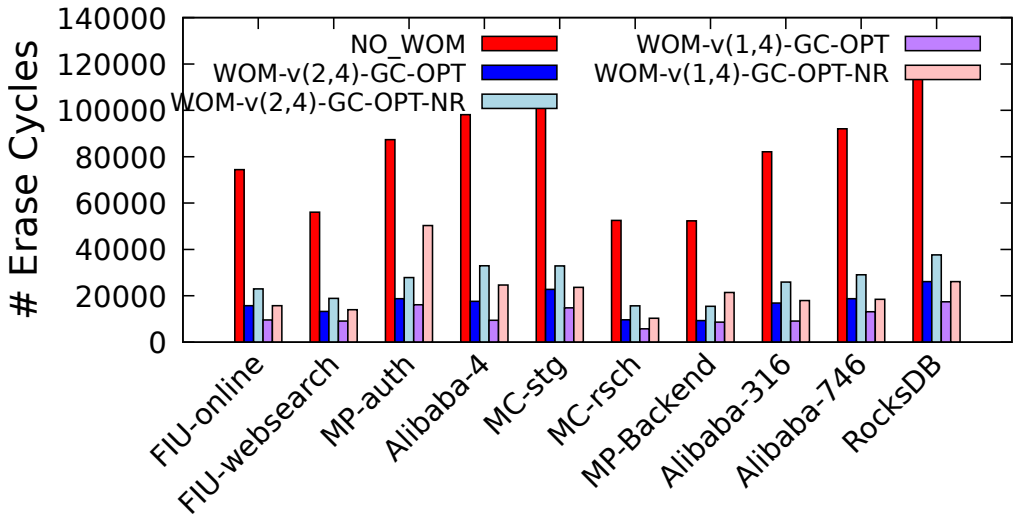
(a) WOM-v(k,N) Baseline Endurance



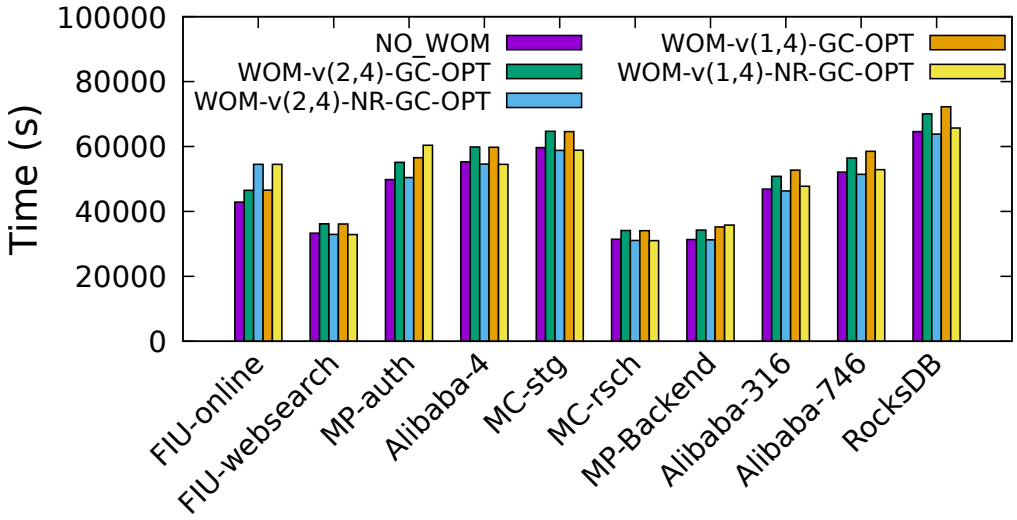
(b) WOM-v(k,N) Baseline Performance

Figure 3.11: WOM-v(k,N) Baseline Endurance and Performance

this problem, we ensure all cells in the entire erase unit are always in the same generation, and remove same generation transition optimization. WOM-v(k,N) also has advantages, it reduce the number of erase operations issued to the device. That counters the first two performance overheads and a combination of these factors result in overall performance gains.



(a) WOM-v(k,N) GC_OPT Endurance



(b) WOM-v(k,N) GC_OPT Performance

Figure 3.12: WOM-v(k,N) GC_OPT Endurance and Performance.

Write Performance

Figure 3.11b and 3.12b show the cumulative time to run write intensive workloads for WOM-v(k,N) baseline and WOM-v(k,N) GC_OPT modes. For WOM-v(2,4) and WOM-v(2,4)-GC-OPT configuration, the performance overheads are only 0-2% of the NO_WOM configuration, with EC gains between 67-69% over NO_WOM configuration as shown in Figures 3.11a and 3.12a.

WOM-v(1,4) baseline coding scheme takes significantly high amount of time to run as compared to default mode for high GC workloads as shown in Figure 3.11b. In such

scenarios, using only NR mode is not sufficient. Instead, a combination of GC_OPT and NR modes help reduce the performance to 0-2% of NO_WOM configuration while keeping the endurance gains from 42%-80% of original configuration as shown in Figure 3.12b and 3.12a.

Read Performance

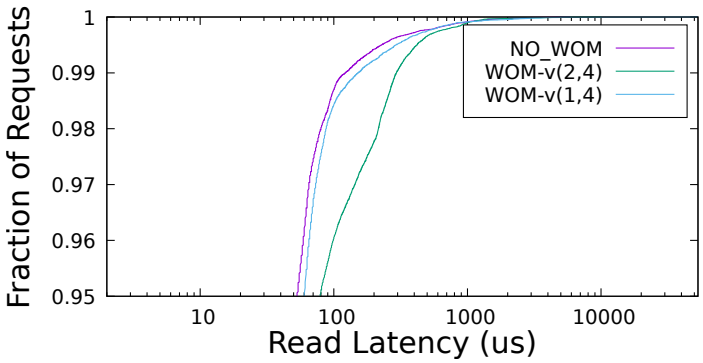
Since all workloads analyzed until now were write heavy and used to study write endurance or performance, we curated read heavy workloads that have reasonable amount of intermixed reads and writes to perform an analysis of the impact of WOM codes on read performance. Specifically, we are interested to see whether the tail latency of reads is impacted, as tail latency is a particular concern in practice. In this section, we present results for the baseline implementation of WOM-v(k,N) coding schemes, without the optimizations described in 3.6.4. The results are therefore a worst case upper bound on latency, as the WOM-v(k,N) optimization codes would only further reduce any impact on latency. As we will see there is no significant impact on latency for WOM-v(k,N) baseline coding schemes, and we therefore expect no impact on WOM-v(k,N)-GC-OPT and WOM-v(k,N)-NR codes either.

Figure 3.13a, 3.13b and 3.13c show our performance results for traces from 3 different repositories. For all three cases, we observe that the 95'th percentile tail latency is 0.6-7% for NO_WOM and WOM-v(k,N) baseline encoding schemes, and that there is no large tail latency introduced due to the use of WOM-v(k,N) code.

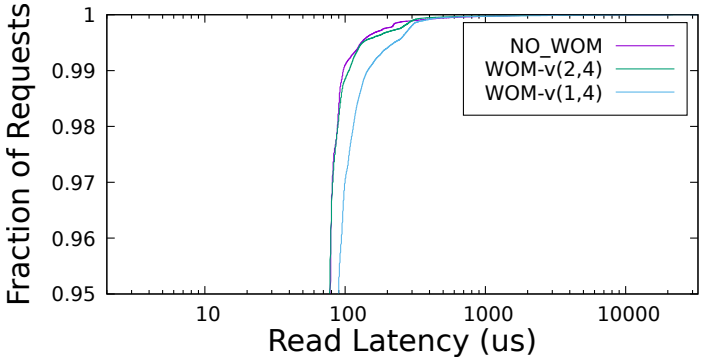
3.7.5 Comparison with MLC Drives

Given that increases in drive endurance from WOM codes come with space overheads, an interesting question is how the WOM-v endurance/space tradeoff compares to simply using MLC technology. MLC cells have higher PE cycle limits than QLC drives, but are less space efficient. In particular, if we reduce a QLC drive to an MLC drive with the same number of physical cells, we lose 50% of logical capacity (2 bits per cell instead of 4), which is identical to the logical capacity loss when applying WOM-v(2,4) codes to a QLC drive. In exchange for the 50% capacity loss, the MLC drive's endurance will increase because the PE cycle limit of a cell increases from 3K for a QLC drive to 10K for an MLC drive [1, 144, 176, 169], and the WOM-v QLC drive's endurance will increase compared to QLC due to overwrites between erases.

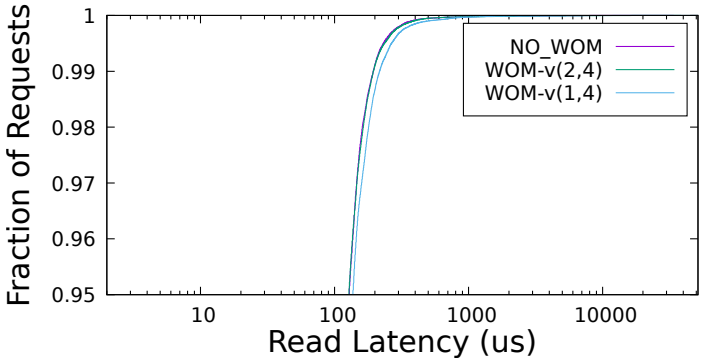
The goal of this section is to compare the endurance of an MLC drive to that of a WOM-v(2,4) QLC drive with the same logical capacity and the same number of physical cells. Endurance is the amount of user data that can be written before the PE cycle limit is reached. We can use our results from Section 3.7.3 to estimate endurance for a WOM-v(2,4) QLC drive for a given workload, based on the number of erases observed for the experiment with the corresponding trace and the amount of user data written in the



(a) *MSR-Cambridge Web1*



(b) *MSR-Production DAPL*



(c) *Alibaba 3*

Figure 3.13: *Figure 3.13a, 3.13b and 3.13c show read tail latency of MSR-Cambridge Web1, MSR-Production Display Ads Payload and Alibaba-Server 3 workload with read:write ratios of 11:9, 4:3 and 5:6 respectively.*

trace. We also ran experiments for all workloads on an MLC drive with the same physical capacity and recorded those numbers.

Figure 3.14 compares the ratio of the endurance of an WOM-QLC drive and the endurance of an MLC drive for different workloads based on the methodology described

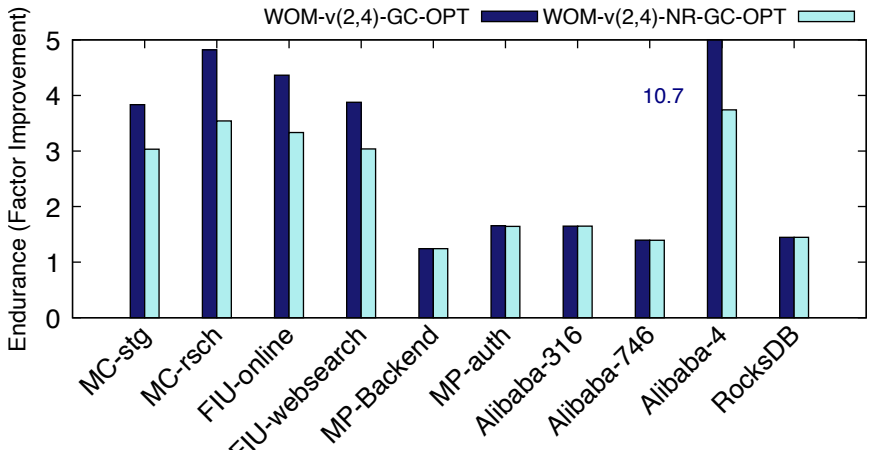


Figure 3.14: *The ratio of write endurance of WOM-v based QLC drives in GC_OPT and NR-GC_OPT mode and the write endurance of an MLC drive with the same logical capacity and the same number of physical cells. For all workloads, WOM-v(2,4) enabled QLC drives provide better endurance than MLC drives.*

above. We observe that for the same write pattern, the endurance of the WOM-QLC drive exceeds the endurance of the MLC drive in all scenarios (ratio larger than 1). This is the case even for the workloads with higher garbage collection where improvements from WOM codes were lower than for other workloads. On average the improvement in endurance is a factor of 3.5x for GC_OPT mode and 2.4x for GC_OPT-NR Mode, with negligible performance overheads as compared to a NO-WOM QLC drive using the NR Optimization. In summary, we conclude that WOM-v codes can significantly improve drive endurance compared to standard MLC as well as QLC drives.

3.7.6 Theoretical Analysis

The goal of this section is to explore analytical approaches to obtain estimates of the endurance gains under WOM-v. The most realistic model with greedy GC and skewed popularity distribution is not easily analytically tractable. Instead we analyze two simpler models. One with greedy GC and uniform page popularity distribution based on prior theoretical results. One with LRW GC and skewed popularity distribution based on a new theoretical framework we present. We find that both overestimate number of require erases, but that the model based on greedy GC and uniform popularity distribution provides tighter bounds.

The theoretical framework we explore in this section evaluates the expected gain of a WOM coding scheme in terms of reducing the number of required erase operations for writing a given amount of data in an SSD, compared to the case where no WOM coding is applied, by considering the write amplification resulted by coding overhead and the garbage collection operation into the picture. However it makes a simplifying assumption

that all segments of the stored data have the same rate of expiration. Later, we evaluate the theoretical gain of the presented WOM-v scheme based on this theoretical framework to see how well it can predict the gain of this coding scheme for new SSD generations. Finally, we extend a different version of this theoretical framework by considering a different garbage collection scheme, and also present a more elaborate theoretical framework for evaluating the expected gain of WOM codes even when different segments of logical data have different expiration rates, which more accurately resembles the real world applications.

All of the theoretical models considered in this section are based on the optimal scheme of garbage collection, namely GC-Opt, introduced in the previous section, in which the valid pages remaining in the erase blocks which are used for writing more data to them without erasing (i.e., reprogramming with WOM codes) will remain in place. In the final subsections we present a comparison between our analytical predictions of the reduction in the number erase operations with the results of simulating real traces using the implementation of WOM-v codes in a flash simulator in Linux LightNVM Open-Channel SSD Subsystem module that we developed and was first presented in [76]. This comparison better demonstrates the power and limitations of the analytical frameworks. Before we begin the presentation of theoretical framework, we need to first introduce the notations through a general description of the WOM coding schemes.

3.7.7 Erasure Factor

As discussed above, the main goal of using WOM codes is to reduce the number of erase cycles required for writing same amount of information to the drive. On the other hand, they introduce a certain amount of storage overhead and therefore, increase the total amount of writes. In order to measure the overall impact of WOM codes on the drives endurance, Yaakobi et al [187] introduce the *erasure factor* metric as the ratio between the number of block erasures and the number of logical block writes. Building on top of the analysis of write amplification for the greedy garbage collection provided in [41] Yaakobi et al [187] then derive the closed-form formula for erasure factor for a WOM coding scheme with t generations and rate R as follows,

$$E(\alpha) = \frac{1}{t\left(1 + \frac{\alpha}{R}\mathrm{W}\left(-\frac{R}{\alpha}e^{-\frac{R}{\alpha}}\right)\right)}, \tag{3.1}$$

where α represents the ratio between the total logical and physical space in the drive. In other words, for a drive with x% over provisioning, $\alpha = 1/(1+x)$. Finally, $\mathrm{W}(\cdot)$ is the Lambert W function. Note that this formula is derived based on several over-simplified assumptions for the greedy garbage collection scheme, which makes the analysis tractable. For instance, page invalidation is uniformly random in the sense that all valid pages in the drive have the same probability of getting invalidated at any time. Also, the drive is

considered to be in a steady state, which means the workload does not change over time.

Although such simplifying assumptions are crucial to make the analysis tractable, the real-world traces rarely satisfy them. Moreover, the erasure factor only evaluates the impact of WOM codes on the endurance, but does not capture any aspect of performance. Such analysis through theoretical models easily becomes intractable considering the large number of parameters involved. Hence it is necessary to have a different method to evaluate the implications of incorporating WOM codes in SSD drives.

3.7.8 Generalizing the WOM-v Coding Scheme

While our presentation of WOM-v coding has so far focused on QLC SSD drives, we now show that it is easy to extend the WOM-v code design to any other flash cell configuration with N distinguishable voltage levels.

Let $n = \log_2(N)$, then each of the N distinguishable voltage levels for any cell can be labeled as an n-bit codeword for the general WOM-v(k, n) coding scheme, whit some $k \leq n$. Such coding scheme then maps each sequence of k data bits into an n-bit codeword that can be stored in a single cell as its corresponding voltage level. The *rate* of the coding scheme, denoted by R for a WOM-v code can be computed as,

$$R = \frac{k}{n}$$

Each generation in the WOM-v(k, n) coding scheme consists of a set of 2^k consecutive voltage levels, It is then clear that if we allocate disjoint sets of voltage levels for every generation, the total number of generations, denoted by g for a WOM-v(k, n) code can be computed as $g = 2^{n-k}$. However, the codeword sharing technique introduced in Section 3.3.1, allows the last voltage level of every generation, excepting the last generation, to be shared with the next generation. Hence, the number of generations with codeword sharing can be expressed as:

$$g = \frac{2^n - 1}{2^k - 1}$$

3.7.9 Flash Friendliness of WOM-v(k,N) codes

Besides reducing the number of required erase operations, WOM-v codes also have the added benefit that their writes are more flash friendly as they only involve voltage level increases within a short range of voltage levels. For instance, in a standard QLC drive with 16 distinct voltage levels, the voltage increment for a non-coded configuration could be anywhere between V0 to V15. However using a WOM-v scheme, the increase in voltage will be bound by the voltage range spanned by one generation, since in each write we will move up by at most one generation (e.g., the next 4 voltage levels in WOM-v(2, 4) coding scheme). In general, for a WOM-v(k, n) coding scheme, the next write will only increase the voltage level from V0 to the next $2^k - 1$ levels as compared to non-coded configuration

where the voltage may increase anywhere between 0 to 2^n levels. The implications of this lower rate of increment in voltage level during a write operation for WOM-v(k, n) code is that less amount of charge will be injected to each cell as compared to non-coded configuration. This mode of programming a cell in WOM-v encoding scheme is more flash friendly as it induces less program disturb errors in both programmed cell and neighbouring cells as compared to non-coded configuration. Further, gradual voltage increment may also simplify SSD circuitry as the number of possible transitions on each state is significantly reduced in WOM-v configuration as compared to the NO-WOM configuration.

In this work, we focus our attention on reducing the number of erase operations in exchange for more number of writes to improve SSD endurance. We acknowledge that there are other factors, including write operations, and temperature that would impact the endurance of SSD drives. However, such factors have much less impact on the endurance of SSD drives as compared to the erase operation [42].

3.7.10 Uniform Page Invalidation with LRW Garbage Collection

LRW garbage collection is the simplest and easiest scheme of garbage collection for implementation in SSDs. In this scheme, erase-blocks are chosen for garbage collection based on the same order in which they are written to, so there is no need for any background processing of the number of valid pages and optimizing the choice of erase-blocks selection. When no WOM coding is applied, this scheme simply treats the erase-blocks as a single first-in-first-out queue. In a WOM coding scheme with t write cycles, however, the blocks could be written to t times before they need to be erased. Hence, we would end up having t consecutive queues of erase blocks, such that the fresh erased blocks are written to at the beginning of the first queue and rewritten to without erasing at the beginning of each of the other $t-1$ queues, and finally getting garbage collected and erased at the end of the last one.

Consider an SSD with a physical storage capacity of T erase blocks. Assume the SSD can store U user erase blocks of data. In other words, the over provisioning is $(T-U)/U \times 100\%$. Assume each erase block consists of N_p pages. Figure 3.15 depicts the steady state of LRW garbage collection in it where a WOM coding scheme with t write cycles is applied. The rectangles represent the erase-blocks and the number inside each rectangle represents the expected number of valid pages. Assuming the uniform invalidation of pages, and noting that all erase-blocks at the beginning of each queue are fully written to (i.e., have N_p valid pages), it is easy to see that all queues have the same expected length. We are now interested to calculate the expected write amplification, i.e., the ratio between the rate of physical and user data written to the drive. According to the diagram of Fig. 3.15, it is clear that write amplification is denoted by A, the rate of writing physical erase-blocks, normalized by the the rate of user data written to the drive.

Let the expected number of active sequences of $1/R$ codes pages in the erase-block at

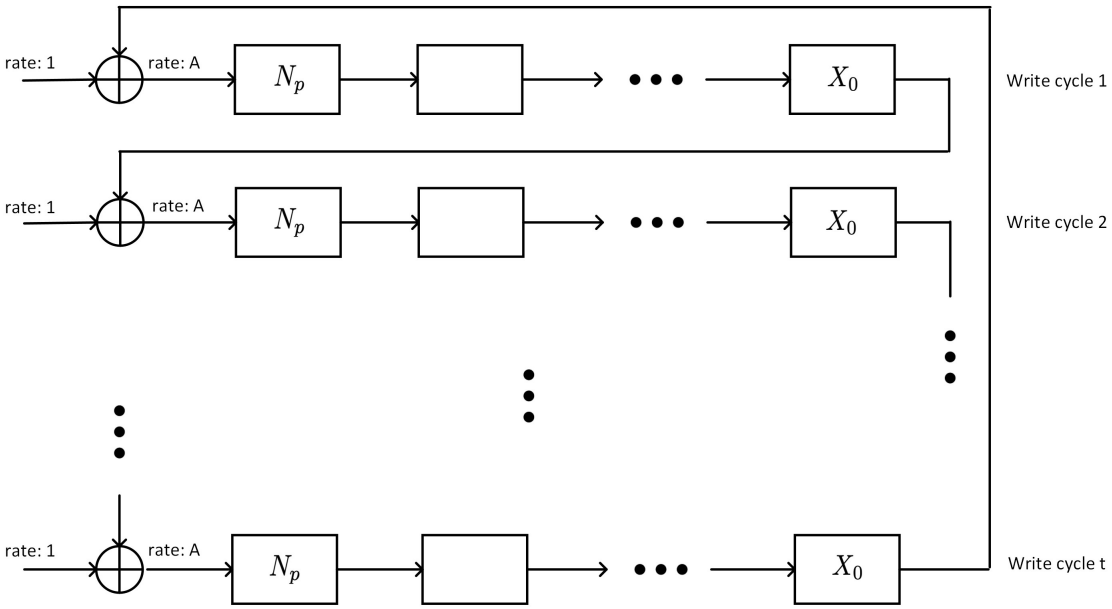


Figure 3.15: The LRW garbage collection scheme with a t-write-cycle WOM coding scheme. User data is written to the first block of each row which are called active blocks.

the end of the queue associated to each write cycle be denoted by as X_0. Given that for every single user data block written to the drive we have A physical block writes, and assuming the steady state of the system, we have:

$$1 = A(1 - \frac{X_0}{RN_p}). \tag{3.2}$$

Moreover, we have T/t erase-blocks in each queue, and they move by a normalized rate A blocks towards the end of the queues. Therefore, for each sequence of $1/R$ codes pages, the expected number of $1/R$-pages invalidation operations happening between it is first written to the drive at the beginning of one of the queues until its associated erase-block reaches the end of the queue is

$$\frac{TN_pR}{A}.$$

Hence we have,

$$X_0 = RN_p\left(1 - \frac{1}{UN_pR}\right)^{\left(\frac{TN_pR}{A}\right)} \approx RN_p\mathrm{e}^{\left(\frac{-R}{A}\right)}, \tag{3.3}$$

where the approximation holds when UN_p is a very large number.

From (3.2), and (3.3), we have,

$$A = \frac{1}{1 - \mathrm{e}^{\left(\frac{-\alpha}{A}\right)}},$$

which we can rearrange as

$$\mathrm{e}^{\left(\frac{\alpha}{A}\right)} = \frac{-\alpha}{\frac{\alpha}{A} - \alpha}.$$

Now, multiplying both sides by $\left(\frac{\alpha}{A} - \alpha\right) \mathrm{e}^{-\alpha}$, we have

$$\left(\frac{\alpha}{A} - \alpha\right) \mathrm{e}^{\left(\frac{\alpha}{A} - \alpha\right)} = -\alpha \mathrm{e}^{-\alpha}. \tag{3.4}$$

Solving (3.4) for A, we then have,

$$A = \frac{\alpha}{\alpha + \mathrm{W}\left(-\alpha \mathrm{e}^{-\alpha}\right)}. \tag{3.5}$$

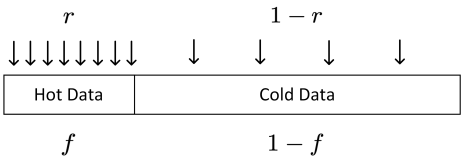
Note that A is merely the rate at which the physical erase-blocks are moving in the diagram of the steady state in Fig. 3.15, assuming the user data is written at rate t in parallel write cycle queues, and does not necessarily match the write amplification in this scheme. In fact, the actual write amplification, which is defined as the ratio between the physical and user writes, would depend on how we treat the valid pages in the erase-blocks leaving from the end of queues 1 to $t-1$. If the valid pages in the erase-blocks at the end of queue i, for $i < t$, remain in place and new pages would be written on top of the invalid pages, then write amplification would be calculated as,

$$\text{Write Amplification} = \frac{t}{\frac{t}{R} + \frac{AX_0}{RN_p}} \approx \frac{tR}{t + RA\mathrm{e}^{\left(\frac{-\alpha}{A}\right)}}. \tag{3.6}$$

Now, finally to calculate the erase factor we just note that the rate of physical erase-blocks exiting all t queues in the diagram depicted in Fig. 3.15 is the same, and they all have the same expected number of valid pages that should be carried over to the first erase-block of the next queue. Hence, for one set of t erase blocks exiting from the end of all t queues, only one of them gets erased, and the expected total amount of new user data that could be written to them when they go to the beginning of the next queue is, $t(1 - X_0/N_p)/R$ blocks. Therefore,

$$\text{Erase Factor} = \frac{1}{t\left(1 - \frac{X_0}{RN_p}\right)} \approx \frac{1}{t\left(1 - \mathrm{e}^{\frac{-\alpha}{A}}\right)} \tag{3.7}$$

Figure 3.16: The hot/cold model of data invalidation.



3.7.11 Hot and Cold Data Model

In this subsection, following the works of [156] and [41], we will consider a simple non-uniform model for page invalidation probability in an attempt to have a more realistic approximation of the real-world workloads. To this end we use the Rosenblum's hot/-cold model [156], to separate the pages into two categories, namely "cold" and "hot", as demonstrated in Fig. 3.16. In each invalidation operation we assume the invalidated page is a hot page with probability $r > 0.5$, and it is a cold page with probability $1 - r$. Hence the normalized rate of invalidation and arrival of the new pages for hot and cold data is r and $1 - r$ respectively. Moreover, we denote the expected fraction of space occupied by hot pages by f.

Similar to the discussion in the previous subsection, when a WOM coding scheme with t write cycles is applied along with the LRW garbage collection, the steady state process of the drive can be described by the diagram depicted in Fig. 3.17.

From Fig. 3.17, one can see that for the rate of moving valid hot pages from a queue to the next, denoted by H, we have.

$$H = (H + r)\left(1 - \frac{r}{UN_p f}\right)^{\frac{\alpha UN_p}{A}} \approx (H + r)\,\mathrm{e}^{-\frac{\alpha r}{Af}}. \tag{3.8}$$

Similarly, for the rate of moving valid cold pages to the next queue, denoted by C, we have

$$C = (C + (1 - r))\left(1 - \frac{1 - r}{UN_p(1 - f)}\right)^{\frac{\alpha UN_p}{A}} \approx (C + (1 - r))\,\mathrm{e}^{-\frac{\alpha(1-r)}{A(1-f)}}. \tag{3.9}$$

Finally for the rate A of moving erase-blocks to the right in each queue we have,

$$A = H + C + 1,$$

where, replacing H and C from (3.8), and (3.9), respectively results in

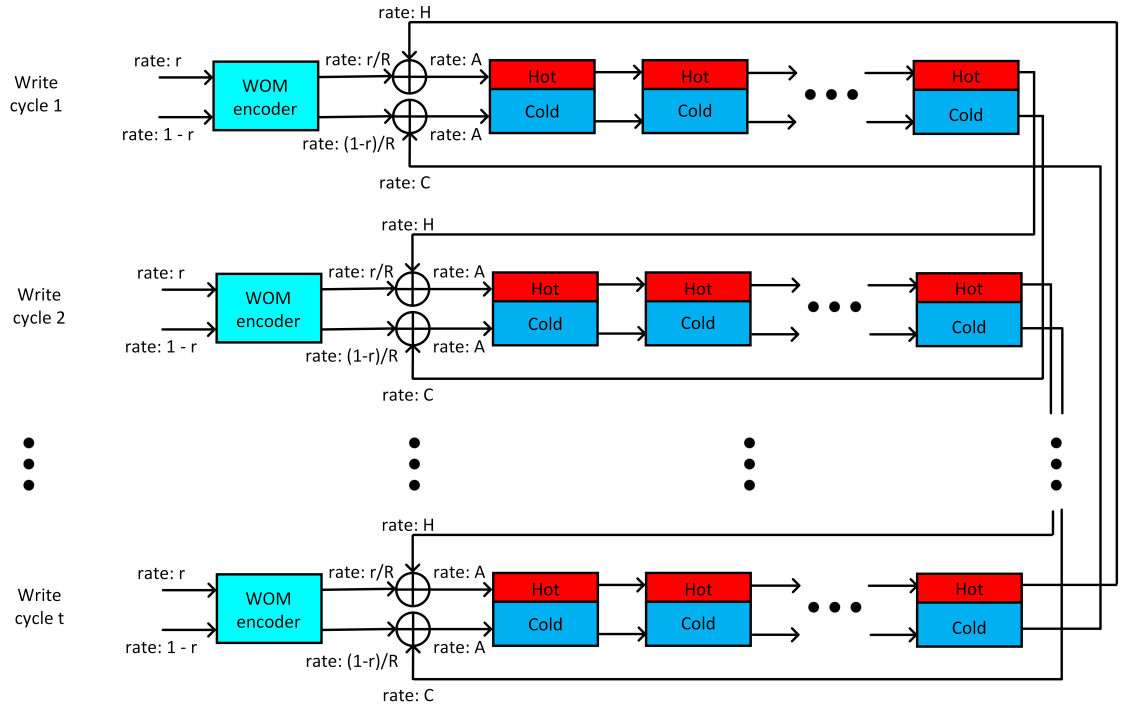


Figure 3.17: The steady state of the LRW garbage collection with hot and cold data invalidation model. User data is written to the first block of each row called the active block.

$$A = 1 + \frac{r}{e^{\frac{\alpha r}{Af}} - 1} + \frac{1-r}{e^{\frac{\alpha(1-r)}{A(1-f)}} - 1}. \tag{3.10}$$

Now note that for every t blocks of user data written at the beginning of the t parallel queues in Fig.3.17, we have A erase-blocks leaving from the end of each queue. However, only the A blocks leaving the queue associated with write cycle t will be erased, and hence we have,

$$Erase\ Factor = \frac{A}{t},$$

where A can be calculated numerically from (3.10) for any particular value of r, f, α, and the WOM coding rate R.

3.7.12 Evaluating the Analytical Models

In this subsection we compare the theoretical results introduced in the previous subsections with the trace simulations in order to demonstrate how valid the simplifying

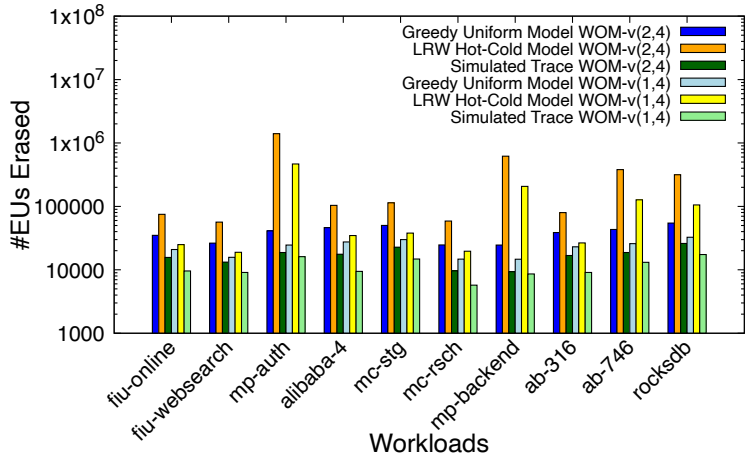


Figure 3.18: Difference in number of Erase Units (EUs) erased of a Greedy Uniform distribution and a Hot-Cold separated LRW analytical modeling with Simulated trace results. We note that in most cases, the Hot-Cold Separated Erase Units form an upper bound on the Erase Units obtained using a real world trace.

assumptions made for analysis have been, and how accurate the analytical results can predict the required number of erase operations for the real-world traces.

For the hot and cold data model, as introduced in the previous subsection, we approximate the real-world workload by considering two parameters in the model, namely r, the rate of update for hot data, and f the fraction of storage space occupied by hot data. Although, the actual update frequency of data in a real-world workload is much more complex, this provides a simple and tractable abstract model to incorporate the differences in temperature spectrum of different segments of stored data in various workloads into the analysis, and hence enables the model to differentiate between different workloads based on their temperature profile. To this end, for each trace we considered the hottest pages contributing to 90% of the updates as the hot pages and evaluated the fraction of storage space occupied by them as the f parameter for that trace. The corresponding value of f is then plugged in along with $r = 0.9$ into equation (3.10), to derive the corresponding value of the parameter A, and from there the erase factor is calculated for each trace. The erase factor multiplied by the total number of written blocks would provide the predicted number of EUs erased based on this model.

Figure 3.18 shows a comparison of the required number of erase operations in the real world traces when using WOM-v codes with GC-Opt, and what could be predicted using each of the theoretical models, namely the Greedy Uniform Model, assuming greedy GC and uniform page invalidation distribution based on prior theoretical results [187], and the LRW Hot-Cold Model with LRW GC and skewed popularity distribution introduced in the previous subsection. We find that both overestimate number of require erases, but that the model based on greedy GC and uniform popularity distribution provides

tighter bounds. However, in many cases neither of the theoretical models bounds are tight. The reason is most likely that the analysis relies on several simplifying assumptions in order to be tractable, including for example an assumption that page invalidations happen uniformly at random, or we use LRW GC rather than greedy. This emphasises the need of a realistic simulation framework for better evaluation of WOM coding gains.

3.8 Related Work

WOM Coding schemes[153, 52] were initially explored for tape-drives and compact-disks. Recently, WOM codes have been introduced for MLC and TLC flash [184, 183, 190]. The implementation and limitations of such approaches considering hardware constraints have been evaluated [191, 110, 111].

Yadgar *et. al* [191], propose ReusableSSD, where invalid pages inside an SSD are reused and overwritten with encoded data. During the first iteration of write to the drive, the data is written as-is. In the second write, invalidated pages are reprogrammed and the disks overprovisioning space is used to handle increased storage overhead. The analyisis limits itself to two writes. More than two overwrites are not possible.

Margaglia *et. al* [111] implement WOM code on hardware MLC flash. The key contribution of this work is that the practical gains of WOM coding scheme for MLC flash is lower than the theoretical gains expected. Other factors such as workload distribution, underlying media reliability and Overprovisioning space determine the eventual reliability increase in WOM codes. This work does not cover higher degree of WOM coding schemes suitable for TLC and QLC flash devices.

Further, biasing code-words towards one type of data format as opposed to another to keep voltage levels to a minimum have been studied [186, 200]. Recent work also suggests how encoding data such that eventual code is kept at a lower or middle voltage level helps reduce program interference errors [200, 179, 186].

3.9 Summary and Implications

Below we summarize our findings and their implications.

1. WOM-v codes reduce the amount of erase cycles by 4.4-11.1x for high-density QLC SSDs. This directly improves the lifetime of flash media and reduces the overhead of purchasing, replacing and restoring data from an older drive.
2. Even for workloads that exhibit high-write amplification, WOM-v codes present a novel opportunity to delay garbage collection in GC_OPT mode until the entire erase unit is no longer programmable. Such optimization is possible *only* in WOM-v codes where data can be overwritten without erasing previous data on a SSD.
3. Even for performance-critical workflows deployed on high-density SSDs, drive endurance can be enhanced to a significant degree using WOM-v codes without com-

promising on performance using the no-reads optimization.

4. The space amplification of WOM-v codes can be a cause of concern for SSDs that exhibit high space utilization. For such drives, we recommend routinely transitioning between WOM-v codes with different code rates based on the space requirement of the drive. We leave evaluation of such dynamically transitioning coding schemes based on space-utilization as part of future work.
5. Although this paper demonstrates the effective use of WOM-v codes and discusses the implementation, evaluation, and optimization of WOM-v codes in the context of QLC SSDs, WOM-v codes can be easily extended to higher density, futuristic SSDs such as PLC SSDs.

3.10 Use cases and Impact

WOM-v coding scheme presents promising results for use cases where there is a large amount of repeated overwrites for the same type of data. As SSDs do not have in-place writes, a mechanism could be developed within the flash translation layer to overwrite the erase units with data that is similar to previously written data. One way to achieve this would be to use scrambler [108] that already controls and complements bits being written to the underlying flash drive to reduce overall write entropy.

Another common use case could be taking data snapshots and version less backups where older data needs to be discarded to make way for new data. In such scenarios, WOM-v codes could simply overwrite previously existing data without having to relocate to another location on the drive.

This work has been published at three different venues [76, 77, 75] and has received strong acceptance, including a best paper award [76] and an invitation in the special edition of ACM TOS [75]. We have also received interest from industry including researchers at Samsung about our technique and it's feasibility on their recent Solid State Memory design.

3.11 Future Work

A new family of WOM codes [76, 77, 75] for high density flash drives leads to some open questions for future research:

1. **Using Over-provisioned Space v/s ECC** Once flash cells start reaching EINVAL state, one way to not rely on ECC would be to write data of any unprogrammable cell to the over-provisioned space in the SSD. This brings additional management overhead but does not depend on internal ECC. This also improves the read performance as we no longer use ECC to correct the cells in EINVAL.
2. **Existing QLC RBER** During our preliminary discussions with some industry partners, we observed that there is a large variation in accepted values of RBER (Raw

Bit-Error Rate) between different SSD vendors and across different SSD types. More discussion should be done to determine what is the industry accepted range of RBER rates for QLC flash and what error rates should academics assume for future reliability based research on dense flash storage media.

3. **Voltage Transition Limitations** Some voltage levels may be reserved. Our WOM code model can discard such reserved voltage levels while dividing different voltage levels into generations. Hence, our model would continue to work irrespective of forbidden voltage levels in hardware.
4. **Differential error rates at different voltage levels** All voltage thresholds are not equally error prone, and some states may have a higher error rate than others [179, 200]. One approach to implement WOM-v scheme would be give a larger voltage range between error prone voltage states. In voltage based model this can be achieved by merging multiple voltage states at a higher voltage level together and assigning a single code word to that voltage range.
5. **Re-programming Feasibility** Previous reliability studies attribute erase operation as a primary cause for flash wear. WOM codes involve reprogramming the same cell multiple times. It would be interesting to explore the impact of re-programming on flash wear and if we should take the wear into account while quantifying increased QLC flash lifetime, as discussed for MLC flash in [189, 192].
6. **Hardware Tool Kits for QLC experiments** There exist limited support for QLC flash in open sourced tool kits and simulators [182, 133] to do benchmarks and experiments. Although our work provides a software testbed for code evaluation it would be good to obtain a hardware where new class of WOM codes can be programmed more efficiently.
7. **Impact on simplifying SSD circuit design** Our WOM coding scheme restricts the number of transitions possible from one code word to another by 1/number of generations. For example, with NO WOM, a voltage may transition to one of the 16 transition levels. However, with WOM-v(2,4), we may only transition to the next 4 voltage levels. A future line of research could be towards simplifying SSD circuitry.

Chapter 4

A Framework to evaluate Near-Storage-Computation Applications

4.1 Introduction

The end of Moore's law coupled with the massive growth in data storage requirements has lead to increased interest in disaggregated storage in both single node and distributed storage environments. Disaggregated storage involves keeping the storage nodes at a separate location than the compute nodes so that they can scale independently of each other. In a non-disaggregated storage system, data is accessed and processed locally. However, in a disaggregated storage system, data storage is done at a remote location and transferred from a remote storage to the local compute node for processing. The continuous back-and-forth of data between compute node and storage node contributes to network bottleneck in two ways. First, there is a limited interconnect bandwidth between the compute node and the storage node. Second, repetitive data transfer between the compute node and the storage node leads to an increased latency in serving the application I/O requests. One solution to reduce the data transfer for computation is to offload the data computation to the storage node. This is possible by adding an embedded processor within the storage device.

Recent advances in the SSD hardware help us realise such specialized devices. Traditionally, SSDs were built using HDD interconnects such as SATA, SAS and Fiberchannel. Such interconnects help SSDs to be backward compatible with HDDs. However, SSDs gained both speed and parallelism over time. Hence, PCI-e interconnect with low latency and high bandwidth was introduced to access data on the SSDs. The NVMe protocol [131] built on top of the PCI-e interconnect helps us perform data transfer at lower latency and utilize the large PCI-e bandwidth.

The advantage of using NVMe protocol is not restricted to optimal data transfer. It

also enables sending control information to the disk. This capability of NVMe protocol helps the host to use the rich NVM-e command set to send computational instructions to the device. Such instructions can be used by the device to perform specific computation on the data within the device before storing it in the media or returning back to the host. Such modern NVMe-enabled SSDs are a natural fit to be used as *Near Storage Compute* (NSC) devices.

Computational storage devices give us the following advantages:

1. **Reduced Bandwidth usage:** Since computation is done near the data stored within the device, only final, processed data is transferred from the device to the host.
2. **Reduced Latency:** Placing an on-disk processor near the data source helps coalesce multiple instructions within the disk and send final output to the user thereby reducing the overall latency involved in serving the request.
3. **Reduced Host Computation:** The host CPU gains free cycles by offloading a part of data computation to the device. This additional host computational power can be utilized to serve other foreground requests.

Multiple storage software vendors, such as Samsung [160], NGD Systems [128], ScaleFlux [163, 198] and EidetiCom [129] have built custom Computational Storage Devices. Although such storage devices are in high demand and are increasingly being deployed in production, there are severe constraints that limit the use of such devices for evaluating application computation offload and computational storage research:

1. **High Cost** Samsung's SmartSSD or other specialized hardware are difficult to acquire and are currently not under mass production. Any application computation offload on such devices for evaluation in academia/research is costly and limited.
2. **Device Programability** The hardware device is usually an FPGA hardware or other sophisticated on-disk accelerator that requires domain expertise in low level Verilog and VHDL programming language. This increases overall time to write, offload and port application computation code to the device.
3. **Usefulness for the Application** Application computation offload need not always provide application speedup. The transfer time or the time to resolve the data-dependency between computations may exceed the computation gains achieved, which may lead to an overall reduced performance.

To address these limitations, we build **NESCAFE**, a **Ne**ar **S**torage **C**omputationa**l** **F**ram**e**work to evaluate the gains of offloading storage based computation to a near-storage computational device. A storage-based application is instrumented to direct the underlying device to perform computation on the data being accessed. The user specifies the semantics of computation that takes place inside the device before the data is stored or retrieved. The application I/O requests are intercepted in the read/write path and computation is performed on the data read or written by the application on the device. The application may also invoke a certain computation workflow and leave the computation to take place inside the device. Finally, the original application performance can be compared with the instrumented application performance to measure the gains (if any)

that can be achieved on a particular configuration of host and on-disk CPU configuration and device interconnect.

Nescafe requires $0 additional hardware cost - no specialized FPGA or hardware accelerator is required to evaluate computational offload gains. Nescafe is written in C. A new computation type on the data can be implemented in C without the use of a low-level language. Nescafe also provides application performance gains with high fidelity, taking storage system parameters such as caching and device read ahead into account. Nescafe is built on QEMU and can be compiled against different hardware architectures.

Nescafe speeds up developer effort in offloading and benchmarking x86 based system and application code before deploying the code on real hardware. Nescafe also helps us evaluate whether computational offload is a good fit for an application as many applications may not have compute-intensive tasks that gain performance by computation offload to the on-disk processor.

4.2 Background

4.2.1 Computational Storage Devices

With the end of Moore's law, there is increased interest in pushing computation to domain-specific compute-fabrics. There is increased interest in hybrid-CPU/GPU computing [119], Programmable NICs [96], AI/ML inference chips [85] and computational storage devices [163, 198, 128, 129, 160]. A computational storage device (CSD) is capable of receiving data as well as computational instructions that should be performed on the data. A CSD contains one or multiple CPU's and optionally a FPGA IC board. A particular computation type can typically be specified in the FPGA. Either all data or user specified data read or written to the device can be operated on by the computation logic. Hosts can interact with underlying CSDs using NVMe protocol.

4.2.2 NVMe Protocol

The NVMe (Non-Volatile Memory express) Protocol is the industry standard for Non-volatile memory (flash and persistent memory) fabrics. NVMe can run on a PCIe interconnect. It provides two types of commands the `NVMe Admin Command` and the `NVMe I/O Command`. All interaction between the host and the guest take place using two circular queue pairs - the Submission Queue (`SQ`) and the Completion Queue (`CQ`) created by the host during NVMe device registration. The number of queue pairs match the number of cores in the device. This helps the queues to function in a lockless manner. A request is first queued by the host on the `SQ`. This request contains the Start Logical Block Address `SLBA` the Physical Region Page `PRP List` and a few other instructions. The host then writes the `Doorbell` register to inform the device of an incoming NVMe request. This updates the tail pointer of `SQ`. The device then picks up the NVMe request, processes it and

returns the end result of the request in the `CQ`. The device then sends an interrupt request to the host. The host then services the interrupt by fetching the NVMe I/O completion request and writing again to the `Doorbell` register. This updates the head of `CQ` register and the I/O request is cleared from the `CQ`.

4.3 Prior Work

Riedel et al. [2] proposed one of the first designs of using a processor as a computation unit within the computational storage device. This work proposes a stream based I/O interface for processing data within a disk. Computation takes place while reading and writing data to the disk. The application code run inside the disk is called a *disklet*. The rest of the application is called a *host-resident component*. A disklet is sandboxed component that only works on pre-specified memory and file storage system. This limits the possibility of a malicious program running inside a disk. The key drawback of disklets is that it cannot perform block I/O. Most storage applications assume a block I/O interface for storage applications. Further, disklets cannot initiate or continue a computation on their own limiting the capability of the computational storage device to initiate I/O requests to the underlying media.

Insider [158] provides a FPGA based reconfigurable drive controller as the in-storage computation unit. On the software side, Insider provides a virtual file system interface. This reduces the host side application modifications. It also helps in providing security and isolation to the on-disk application code. However, Insider requires compiling FPGA HLS code into RTL code. Furthermore, complicated programs such as KV-Stores cannot be naturally offloaded to Insider. The main reason for this is the separation of data and control plane. The on disk controller code which is the data plane has no metadata information to initiate sub-computations on its own. The on-disk controller relies entirely on the host to initiate computation. This may not be sufficient for complex, reenterant computations found in modern applications.

Multiple specialized application computation offloading techniques have been proposed. CognitiveSSD [104] provides a power and compute efficient deep learning and graph search (DLGx) computational SSD storage. The key idea here is to program graph search and deep learning within the SSD for indexing massive unstructured data. This helps in using the internal SSD storage bandwidth instead of CPU-Disk interconnect for the data transfer. Since CognitiveSSD is a specialized hardware to perform graph search and deep learning, it cannot be customized to evaluate other host applications. Similarly, ScaleFlux [203] uses FPGA computational unit for PostGreSQL with inbuilt compression that leads to reduced storage footprint, it is able to achieve the same IOPs as non-compressed PostGreSQL buffer. Further, PolarDB [33] provides an interface for offloading analytical workloads in a Relational Database Management System. These are customized hardware solutions for single applications instead of a generic application computation offload framework.

Computational offload has also been explored in a distributed environments. For in-

stance, Active Disks [152] proposed moving computation to networked attached storage disks. More recently, Caribou [72] explored near data processing in database engines. It provides a key value interface over TCP/IP interconnect for accessing data in SSDs at line speed. They also provide optimizations for efficient lookups using custom bitmaps, query over both structured and unstructured data, and provide fault tolerance through replication. The Caribou framework is good for streamed computation, for instance stream selection or decompression. However, Caribou falls short for real world applications that require metadata to be stored within the disk. Also, the performance gains achieved over TCP/IP cannot be achievable over the PCI-e interconnect that is an order of magnitude faster, specifically for embedded applications such as KV stores (RocksDB) or embedded databases such as SQLite.

LeapIO [100] presents a technique to offload computation to ARM SoC co-processors instead of x86 processors. The emphasis is on reducing the x86 CPU cycles in performing storage tasks. They also demonstrate reduced power consumption for storage stacks in the cloud. Adaptive Placement for In-memory Storage Functions [4] provides flexibilty of computation function offload. It emphasises on moving storage functions adaptively between server and client to save microsecond level latency for in-memory media.

Modeling Analytics for Computational Storage [151] demonstrates the gains achievable through computation offload. They show performance improvement in OLAP TPC-DS queries on SparkSQL and Presto. The key contribution is a predictability model for improved latency and bandwidth for disk bound queries. However, this work does not implement and evaluate systems implementation of the computational framework. Hence the theoretical modeling can be considered as a guideline to determine upper bound of gains from computation offload. But a more generic, implementation specific modeling is required for real world application offload assessment.

Willow [166] proposes changing block based interface to SSD storage to enable in storage computation. Each SSD's computation core is treated as a specialized storage processing unit (SPU). Each application runs on a host RPC endpoint (HRE) that send instructions to the underlying SPU. This requires restructuring the standard block interface and the applications that use the block interface.

A virtual object abstraction is proposed in [3] that respects the block interface. Each virtual object stores minimal metadata required to process storage requests. This work is close to our work but involves using a file system interface between application and the underlying computational device. Further, other works such as SSDSim 2.0 [63], OpenExpress [134] and FEMU [101] provide SSD emulators. They need to be modified to incorporate a on-disk processing unit to evaluate application performance gains with data computation at source.

Nescafe can be run entirely in software, supports multiple interconnect latencies and can easily be modified to add new computations.

Computation Storage	Full S/W Emulation	Multiple interconnect Support	Customized Application Offload
Active Disks [2]	✓	✓	✗
Insider [158]	✗	✓	✓
Cognitive SSD [104]	✗	✗	✗
ScaleFlux [203]	✗	✗	✗
PolarDB [33]	✗	✓	✗
Caribou [72]	✗	✓	✗
LeapIO [100]	✗	✓	✓
Willow [166]	✗	✗	✓
NeSCAFe	✓	✓	✓

Table 4.1: *Nescafe compared to other computational storage frameworks. Nescafe provides a software only, multi-interface, multi-application framework to evaluate near storage computation offload.*

4.4 Analytical Model

We present an analytical framework to evaluate storage workloads run on our near-storage compute framework. The goal of our framework is to enable tradeoff comparisons between running a storage workload on the host and running on a NSC architecture. We evaluate three metrics: request latency, host CPU utilization, and device bandwidth consumption. Table 4.2 summarizes the terms we will use in our model. We note that these terms are coarse-grained, intended to capture intuition. In Section 4.6, we show how our simulator can give more precise estimations of performance.

4.4.1 Single Request Model

Variable	Description
lat_{setup}	**setup latency**: time taken by the host to setup and send a request to the device
lat_{media}	**media latency**: time taken to fetch a data block from the media into the device's memory
$xfer(\cdot)$	**transfer latency**: time to transfer a data block across the host-device interconnect
$exec_x(\cdot)$	**execution latency**: time to perform computation on the given data block. $x \in \{host, device\}$.
$\mathbb{D}_{orig}$	original (raw) data read from the device
$\mathbb{D}_{result}$	resulting data object after computing on $\mathbb{D}_{orig}$

Table 4.2: *Variables used in our analytical model.*

We are interested in workflows that first read *raw data* from the storage device, perform computation on the raw data, and derive *result data*.

In traditional architectures, computation is executed on the host. The host sends a read request and performs computation after the raw data is available in host memory. On the other hand, in NSC architectures, the computation is shipped to the device. This enables the device to both retrieve the raw data and perform computation. In this case, only the result data is sent back to the host.

Request Latency. Our model breaks request latency into four main contributors: (1) submitting a request to the device, (2) fetching the data within the device, (3) transferring the data back to the host, and (4) executing computation on the host. These individual latencies are summarized with corresponding variables in Table 4.2. The total latency for the workflow in a traditional architecture is:

$$lat_{host} = lat_{setup} + lat_{media} + xfer(\mathbb{D}_{orig}) + exec_{host}(\mathbb{D}_{orig})$$

A request latency in an NSC architecture also has four main contributors: (1) submitting a request to the device, (2) fetching the data within the device, (3) executing the compute operator in the NSC compute substrate, and (4) transferring only the result data back to the host. The total latency in a NSC architecture is:

$$lat_{NSC} = lat_{setup} + lat_{media} + exec_{device}(\mathbb{D}_{orig}) + xfer(\mathbb{D}_{result})$$

Then the difference in **request latency** between the two approaches is:

$$\begin{aligned} lat_{host} - lat_{NSC} = &(xfer(\mathbb{D}_{orig}) - xfer(\mathbb{D}_{result})) \\ &+ (exec_{host}(\mathbb{D}_{orig}) - exec_{device}(\mathbb{D}_{orig})) \end{aligned} \tag{4.1}$$

The difference in **bandwidth consumption** is simply the difference in data size transferred:

$$bw_{host} - bw_{NSC} = \|\mathbb{D}_{orig}\| - \|\mathbb{D}_{result}\| \tag{4.2}$$

Finally, the host CPU utilization saved is captured by $exec_{host}(\mathbb{D}_{orig})$. While this term is a latency rather than utilization measurement, it approximates the logical time that is freed up on the host CPU when offloading.

Equation 4.1 shows that latency savings in an NSC framework comes from two high-order terms: the reduction in transfer latency due to data size reduction, and a reduction in the amount of time it takes to execute the desired computation. These two terms translate to (1) compute functions that greatly reduce the size of data transferred (*bandwidth reduction*) (2) compute functions that can be easily accelerated on a computational device processor. Equation 4.2 confirms the intuition derived from the transfer latency term in Equation 4.1: compute workloads where $\|\mathbb{D}_{orig}\| - \|\mathbb{D}_{result}\|$ is large, i.e. high bandwidth reduction, are excellent candidates for offloading.

4.4.2 Generalizing to Data Flows

We extend the single request model to multi-access model which is representative of real world workloads. This extends the model to workflows where the compute is a data flow graph. This generalizes the number of computations performed during an operation offload. Let i denote the number of sequential computations in the data flow graph. Then:

$$\begin{aligned} lat_{host} &= \sum_i \left(lat_{setup,i} + lat_{media,i} + xfer(\mathbb{D}_{orig,i}) + exec_{host}(\mathbb{D}_{orig,i}) \right) \\ lat_{NSC} &= lat_{setup} \\ &\quad + \sum_i \left(lat_{media,i} + exec_{device}(\mathbb{D}_{orig,i}) \right) \\ &\quad + xfer(\mathbb{D}_{result}) \end{aligned}$$

Note that when $i = 1$, the latencies degenerate to the equations in Section 4.4.1. Then the difference in request latency is:

$$\begin{aligned} lat_{host} - lat_{NSC} &= \sum_{i-1} lat_{setup,i} \\ &\quad + \sum_i \left(exec_{host}(\mathbb{D}_{orig}) - exec_{device}(\mathbb{D}_{orig}) \right) \\ &\quad + \left(\sum_i xfer(\mathbb{D}_{orig,i}) \right) - xfer(\mathbb{D}_{result}) \end{aligned} \tag{4.3}$$

Bandwidth consumption is determined by the total amount of data transferred:

$$\begin{aligned} bw_{host} &= \sum_i \left\| \mathbb{D}_{orig,i} \right\| \\ bw_{NSC} &= \left\| \mathbb{D}_{result} \right\| \end{aligned}$$

Bandwidth gains are given by:

$$\Rightarrow bw_{host} - bw_{NSC} = \left(\sum_i \left\| \mathbb{D}_{orig,i} \right\| \right) - \left\| \mathbb{D}_{result} \right\| \tag{4.4}$$

The host CPU utilization saved is $\sum_i exec_{host}(\mathbb{D}_{orig,i})$.

Equation 4.3 suggests that computation that are good offloading candidates should run faster on the NSC processor, i.e. $exec_{device}(\mathbb{D}_{orig}) > exec_{host}(\mathbb{D}_{orig})$ or greatly reduce the bandwidth transferred between the host and device. The bandwidth condition is also highlighted in Equation 4.4.

Equation 4.3 also introduces i, a third condition that affects whether a compute workflow should be offloaded. Equation 4.3 suggests that the larger i is, the better a workflow is for offloading. While this is generally true as long as each additional operator i is a compute operator, this argument does not extend to general computation.

4.4.3 Case Studies

We apply our generalized analytical framework on simple computational tasks. We show how our analytical framework is a useful tool for estimating computational gains.

Latency reduction

A *pointer chase* operation is a data-dependent operation that involves multiple round trips between host and the disk. A series of blocks are accessed, where the address of the next block is embedded in the previously accessed block. In a traditional disk, each block is requested by the host, the next block address is dereferenced, and a subsequent read request is sent back to the device. In a NSC implementation, the pointer chase takes place within the computational device. The operation completes when end of this on-disk link-list is reached. The workload is a good example of demonstrating latency gains through computation offload. We show how speedup provided in Equation 4.3 can be customized to give us an estimate of latency reduction. If both the $exec_{host}$ and $exec_{device}$ are the same, the speed gains become a function of the round trip time as shown in Equation 4.5.

$$\begin{aligned} lat_{host} - lat_{NSC} = \sum_{i-1} lat_{setup,i} \\ + \left(\sum_{i} xfer(\mathbb{D}_{orig,i}) \right) - xfer(\mathbb{D}_{result}) \end{aligned} \tag{4.5}$$

Bandwidth reduction

Predicate filtering is an operation commonly selected for offloading. For the filtering operation, $exec_{host}(\mathbb{D}_{orig}) \approx exec_{device}(\mathbb{D}_{orig})$, but $\left\|\mathbb{D}_{orig}\right\| > \|\mathbb{D}_{result}\|$. For example, good filtering operations can have selectivity between 0.02 and 0.04 [66], where *selectivity* denotes the fraction of data that is returned from the filter operation, i.e. $\frac{\|\mathbb{D}_{result}\|}{\left\|\mathbb{D}_{orig}\right\|}$. Assuming a selectivity of 0.03, then $\frac{\|\mathbb{D}_{result}\|}{\left\|\mathbb{D}_{orig}\right\|} = 0.03 \Rightarrow \|\mathbb{D}_{result}\| = 0.03 \cdot \left\|\mathbb{D}_{orig}\right\|$.

Plugging these variables into Equation 4.1 yields:

$$lat_{host} - lat_{NSC} \approx xfer(\mathbb{D}_{orig}) - xfer(0.3 \cdot \mathbb{D}_{orig})$$

Assuming $xfer$ is effectively linear for large enough data sizes, we get:

$$lat_{host} - lat_{NSC} \approx xfer(0.97 \cdot \mathbb{D}_{orig})$$

For large data sizes, the transfer time can be substantial. For example, for a 160GB SQL table transmitted over a PCIe Gen 3.0 link that has a transfer rate of 1GB/s, the latency saved by offloading can be around 155 seconds, which is substantial [66]. Workloads that enable bandwidth reduction are often cited as offloading candidates, which confirms our analytical model.

Compute Acceleration

For operations that can run faster on an FPGA or ASIC, $xfer(\mathbb{D}_{orig}) \approx xfer(\mathbb{D}_{result})$ but $exec_{host}(\mathbb{D}_{orig}) > exec_{device}(\mathbb{D}_{orig})$ For example [168] showed that regular expression matching, when carefully optimized on an FPGA implementation, can take as low as 1-10% of the latency when compared to a host-side CPU. In order words, $\frac{exec_{host}(\mathbb{D}_{orig})}{10} = exec_{device}(\mathbb{D}_{orig})$.

Plugging these variables into Equation 4.1 yields:

$$lat_{host} - lat_{NSC} \approx 0.9 \cdot exec_{host}(\mathbb{D}_{orig})$$

4.4.4 Need for a Computational Simulator

Our analytical measurement framework gives us theoretical latency, bandwidth and computation gains achievable for simple computational tasks. However, we see that even for simple intuitive micro-benchmarks, an analytical model may not give exact speedup gains. This is because multiple device and system level nuances are not captured by the model. For instance, the device and computation inter-process communication latency is not taken into account. Further, the equation fails to capture system patterns such as device read ahead and host side caching. For capturing such features, a software system that emulates actual hardware and physically runs an offloaded computation is required.

Moreover, for more complicated tasks, all inputs to the equation may not be available. For instance, to observe speedup gains in benchmarks such as *zlib compression*, the number of compressed block sizes may not be known in advance. For offloading *journal checksums*, the overall gains may depend on multi-threaded nature of journaling. Also, the transaction size may not be known in advance. Hence predicting the number of checksums that need to be computed may not be feasible. A *key-value* store may offload LSM-Tree compaction to the device. Since compaction is workload dependent, it would be difficult to assess how many compaction operations are issued. Furthermore, applications like *Spark* have an in-memory computation model. Most computations are deferred until the results of the computation are requested by the application. Modelling such applications may be difficult with an analytical framework.

Finally, the variance will only increase when we analyze benchmarks where gains come from multiple sources - latency, bandwidth and host CPU usage. In the next section, we describe the design of our software based Near-Storage Compute simulator that gives more accurate speedup measurements than the analytical framework taking all system nuances into account during a computation offload to the device.

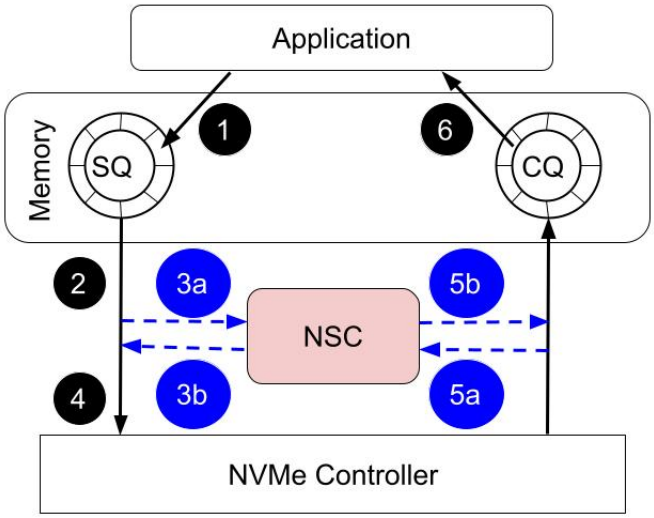


Figure 4.1: *Nescafe Architecture: A NSC (Near Storage Compute) engine communicates with NVMe device polling thread through IPC. 3a and 3b are performed for write based computations. 5a and 5b are performed for read based computations.*

4.5 Architecture

In this section, we describe the Nescafe architecture. We build Nescafe on top of FEMU [101], a NVMe device emulator. Nescafe does not require additional FPGA hardware or knowledge of hardware level programming. It can be natively run within a QEMU Virtual Machine environment.

More recent fast storage devices such as SSDs and persistent Memory devices such as Intel 3D-Xpoint [195] use NVMe protocol [99, 131]. NVMe protocol exposes a queue pair model for I/O processing. Each queue pair consists of a Submission Queue (**SQ**) and a Completion Queue (**CQ**). The host creates these queues in the host-device shared memory. The `NVMe Admin command` is used to create these queues during a host-device handshake. The number of queue pairs correspond to the number of cores in the host. This enables lockless access to each queue pair by each core in the host. Once the queues are created, `NVMe I/O commands` can be issued to the underlying device.

A typical `NVMe I/O command` workflow is as shown in Figure 4.1. Each NVMe I/O command is a 64 byte read or a write operation that is accompained by other metadata such as start LBA (`SLBA`), `PRP` (Physical Page Region) List, and `NLBA` (Number of logical block Addresses). A NVMe command is queued in the submission request queue (1). The host then updates a `doorbell` register to update the tail pointer of the circular queue. This triggers the guest to fetch the NVMe instruction from the **SQ** and process it in the device (2,4). Once the request is processed, the device constructs a end of completion request. This completion message is queued in the **CQ** (6). Next, the device sends an

interrupt to the host notifying the host about a completion request. Finally, the host reads the completion request and updates the **CQ** head to clear the I/O.

FEMU [101] is a Flash Emulator that uses RAM as a backend device substrate. This helps the emulator to customize and place configurable I/O access latency to emulate a real NVMe device. In FEMU, the NVMe interrupt mechanism on completion is replaced by a NVMe polling thread. FEMU also removes `doorbell` that provides significant performance gains.

We build Nescafe on top of FEMU using the following approach: First, we fork a near storage computation (NSC) process from the NVMe poller thread. This computational process runs on a dedicated core other than the NVMe poller thread and the host vCPUs. the device process and computational process interact using `FIFO pipes`. Second, we build an interface for the application to inform the NSC process about the type of computation required for each I/O send to the namespace using NVMe directives. Third, we build a number of fixed function interfaces in the NSC process for storage specific computational tasks.

4.5.1 Near Storage Compute (NSC) Engine

Figure 4.1 shows the overall workflow of an offloaded storage computation. Before beginning any I/O operation, the application sends an `NVMe directive` to the device. The `directive` informs the NSC (Near Storage Compute) unit about the requested computation on subsequent data I/O requests. The additional dataflow in the device added to support NSC is highlighted in blue. The application sends request that is queued in the **SQ** (1) and is picked by the guest to be processed further (2). If the Application requested for a write based computation on the data, the data is sent to the Near Storage computational process (3a). The data is either altered or computed upon before being sent back to the NVMe thread (3b). Note that steps (3a), (3b) and (4) may be performed iteratively before proceeding to the next steps. For a read based workload, (3a) and (3b) are bypassed, data is read from the underlying device, and sent to the NSC process to perform appropriate computation (5a). The NSC process may (5b) or may not return the data after performing a specific computation. It may also issue multiple I/Os (4) before storing the request completion result in the **CQ** (6). The application then reads the I/O completion status from the **CQ** and optionally computed data corresponding to a read I/O. Once the application has completed I/O computation, it sends an `NVMe directive` to reset and disable specific computation on all subsequent I/Os.

Next, we describe Nescafe's user-facing interfaces, types of applications that can offload their computation, and how Nescafe can be extended to support new applications. The host-application needs to be altered to add `NVMe directives`. The computation logic needs to be added to the NSC engine. The Guest OS remains unchanged.

4.5.2 Interface

Host-Device Interface

The host informs about the type of computation to the device through `NVMe directives`. A `NVMe directive` is a mechanism to enable host and NVM subsystem or controller information exchange [131]. Each NVMe device is divided into namespaces. A `NVMe directive` can be used to create one or more streams for each namespace. When an application opens an file descriptor within a device, the device internally associates the file descriptor with one of the allocated streams within the namespace. A `directive` can then be imposed on the open file descriptor. This enables a specific type of computation on any data written on the file descriptor.

At the device level, each NVMe command is directed to a particular NVMe Namespace. A NVMe Namespace has a `NVMeDirId` structure that stores directive context. When the application in the host enables or disables a directive, the corresponding flag in this structure is set. The device checks this directive during `nvme_rw` processing, and sets all subsequent I/Os through a specific computation pipeline until the directive is disabled by the application.

We use FEMU's polling mechanism for sending and receiving data to and from the underlying device. A configurable delay is added in the I/O return path in the **CQ** to emulate device interconnect latency. For our purposes, we use PCIe latency of `900ns` [127]. As PCIe interconnect is one of the fastest interconnects, more gains can be achieved by computational storage on high latency, networked, disaggregated storage with slower interconnects.

Device-NSC Interface

The NSC Process is forked from NVMe device polling process and runs on another vCPU. The NVMe device process and NSC Process communicate over `FIFO pipes`. Each computation operation inside NSC is assigned an `opcode`. Additional computation operations can be specified by adding new `opcode` and specifying the corresponding computation logic in a function.

First, the device process relays the computation type requested by host `NVMe directive` to the NSC process through the FIFO pipe. Next, for a write operation, the device process sends all blocks over FIFO pipe to NSC. The NSC performs computation on the block and optionally sends output back to the device for writing output data to disk. For a read operation, the device first reads the block from the device. It then sends the block to the NSC process that optionally returns output data to the device process. The device process then copies any computed output data to shared memory and updates the **CQ**.

Since we cannot increase the physical CPU utilization of a processor, we emulate a relatively powerful NSC processor by reducing host CPU processor. We use `cpulimit` to reduce processor utilization of host CPU process. The `cpulimit` tool sends `SIGSTOP`

and `SIGCONT` signals to a process running on a target CPU. The amortized processor speed is calibrated to the user specified value. This makes CPU utilization of host v/s nsc processor configurable. We run each benchmark multiple times and see negligible difference in our results on using same CPU utilization threshold.

Device-Media Interface

The Device communicates with the in memory-emulated media. We reuse FEMU FTL's device media latency model. Fixed latency is added for `read`, `write` and `erase` operation for the emulated SSD. FEMU emulates a MLC device by default, and additionally supports TLC and QLC flash. This latency is configurable and can be increased to emulate remote network attached storage.

4.5.3 Computation Types

The computation operations offloaded to the NSC engine have varied requirements. Some computations require a simple helper function to be offloaded and run on the device. Others may require complicated I/O path. We classify computation into three categories - *block*, *stream* and *app-metadata* computation. Using these APIs, we are able to support a large variety of computation offloads.

Block Based Computation

A block based computation involves performing computation on a single block before reading or writing to the device. Each input block I/O strictly generates a single output computed block. The original data block and the output data block is of the same size. The `pointer-chase` operation, `checksum` computation, `count` operation are examples of such computation.

Stream Based Computation

A stream based computation involves performing computation on a sequence of blocks. This may or may not compute and create an output block after each input block is processed. The computed output may be of a different size than the input data. The NSC may require multiple blocks from the NVMe device process to complete computation. Typically, an end cleanup function is required once a disable directive or `EOF` is received. This cleanup function clears any temporary in-memory buffers. Examples of stream based computation applications include `compression` and `encoding`.

App-Metadata Computation

Some advanced applications such as key-value stores and big data applications require additional metadata information for computation. For example, a `key-value` store may require on-disk LSM-Tree layout to perform compaction. Similarly, `spark` may require SQL query metadata and table schema to perform *project*, *scan* or *filter* operations. We categorize such applications as `app-metadata` computations. In these applications, additional metadata needs to be transferred to the device to perform computation. We modify the application running inside the host and move the computation to our fixed-function NSC interface. We relay relevant app-metadata to NSC which is used as in input to the fixed computation function inside the NSC. The output (if any) is returned to the host application.

Adding Customized Computation Type

Nescafe can be extended incorporate additional computation offloads as follows:

1. A `NVMe directive opcode` is assigned for the application and the `NVMe Namespace`.
2. Two functions `enable_computation()` and `disable_computation()` are added to the application before and after it sends data to the computational device.
3. A fixed computation function is added corresponding to the `opcode` in the NSC.
4. The logic for computation is moved from the application to the NSC fixed-function.

A number of storage applications fall into one of the three categories block, stream or app based computation (4.5.3). In the next section, we demonstrate how applications belonging to the three categories can be offloaded. The same code can be naturally extended to support similar computation types.

4.6 Evaluation

We evaluate Nescafe on a 14-core machine with 32 GB memory. We create a 10GB block device on the host RAM and emulate it as a NVMe SSD. To emulate PCIe latency, we add 2μs latency between the host and the device [127]. This is a tunable parameter. For emulating SSD device latency, we add 1ms and 10ms latency in the read and write device access respectively.

4.6.1 Micro-Benchmarks

In order to evaluate the latency, bandwidth and computation gains that we can achieve using computational storage discussed in Section 4.4, we design 3 micro-benchmarks that are widely used in many storage systems. First, we design a *pointer-chase* benchmark that has latency gains when offloaded to the underlying computational device. Second, we design a *regular-expression filtering* benchmark that only returns blocks that contain a particular expression on the disk to the host. This benchmark is designed to show

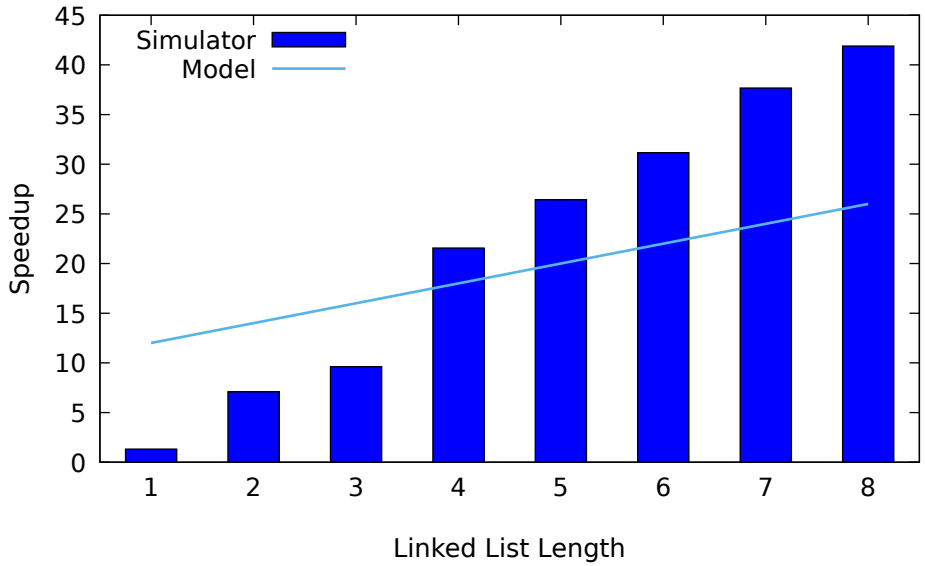


Figure 4.2: *Speedup with increase in linked list length during a pointer chase operation. Speedup is defined as the difference in execution time in Host and Near Storage Compute configuration as described in the Analytical Model in Section 4.4.*

latency gains when only part of the data is returned to the host. Third, we design a *counting* benchmark that sums the number of 1's in a device. This benchmark shows the latency gains one can achieve when a computational intensive task such as encryption or checksumming is done on the device. We show using the analytical framework discussed in Section 4.4 that these simple benchmarks when run on Nescafe indeed give speedups close to those predicted by our analytical model.

Pointer Chase

We run a pointer chase benchmark with 100K lists of length varying from 1 to 8. The relative speedup achieved on the simulator is shown in Figure 4.2. We compare this speedup with the analytical model expressed in Equation 4.5. For short linked-list length, we observe that the time to setup a request and send the request to the underlying device is much higher. Hence the simulator performs much lower than the analytical model estimate for short linked-list sizes. For longer linked-lists, we observe that the time to fetch a block from the underlying device exceeds the setup time. Also, a number of blocks get cached in the page cache. If repeated set of blocks or blocks fetched due to device read ahead are accessed, the application request does not go to the disk and is serviced directly by cache, thereby saving the PCI-e interconnect round trip. Hence the amount of speedup achieved in practice on the simulator for pointer-chase is higher than that predicted by the analytical model.

Regular Expression Filtering

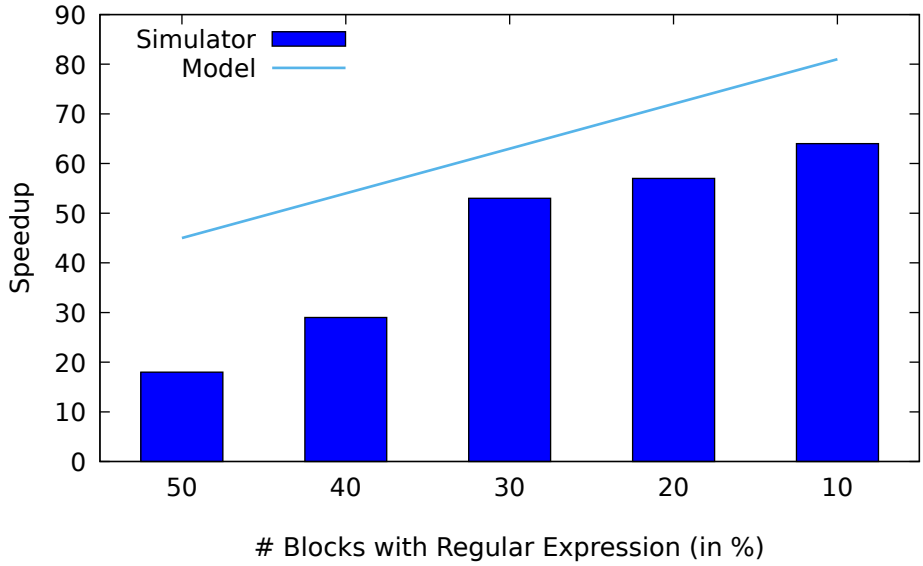


Figure 4.3: *Speedup achieved in regex match by transferring only blocks that match the regular expression pattern. The NESCAFE Simulator under-performs as additional step is required to communicate the next offset from which a scan should begin in the NSC configuration.*

A regular expression benchmark takes a sequence of bits as an input and checks if the blocks on the device contain that regular expression. In the host configuration, all blocks are read from the device and read and processed inside the host. The regex term is matched with contents of each block. On a successful match, the regex counter is incremented.

In a NSC configuration, a read request is sent to the device for a block. If the block contains the requested regular expression, that block is returned. However if the term is not present in the block, the subsequent sequence of blocks is scanned for the regex term until a match is found.

Figure 4.3 shows the speedup achieved in NSC configuration with respect to the host configuration on the simulator and the model. The simulator shows lower speedup as compared to the model. This is because in the NSC configuration there is additional time spent on communicating the next offset from which scan should restart after a block has been found and returned to the host. This step is not required in the Host configuration since the block device is scanned linearly one block after the other without resetting the disk-pointer to a new location in the device. The speedup received by the model is derived from Equation 4.4.3 and is a function of the total time taken in filtering the regex in the host configuration. We conclude that for regex filtering operation, the mathematical model is the higher bound of speedup that can be achieved using a real NSC engine.

Byte Counting

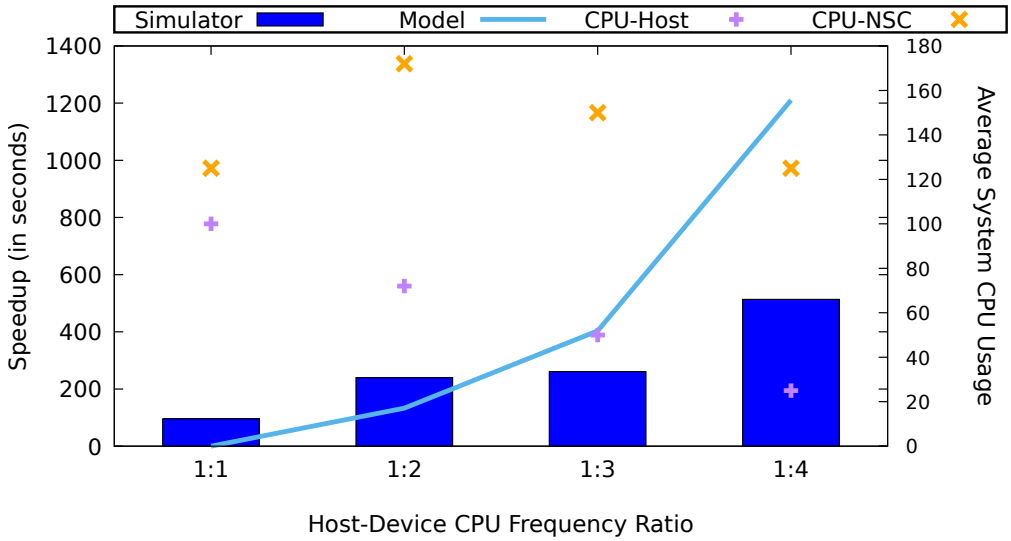


Figure 4.4: *Speedup with Relatively Powerful computational device for count operation. Speedup increases for both the simulator and the analytic model as the on-disk CPU speed relative to host CPU is increased. The System CPU utilization for NSC configuration is calculated as the sum of on-host CPU and on-disk CPU usage and may be up to 200% of a single CPU core usage. We gain higher performance with additional on-disk CPU core.*

Figure 4.4 shows a micro-benchmark that fetches blocks from the underlying device and computes the number of 1's in the device. For each configuration on the X-axis, we vary the ratio of host CPU and on-disk CPU frequency. Since we cannot increase the CPU frequency on a host more than its maximum allowed physical value, we reduce the host-CPU by limiting the maximum allowed CPU usage using cpulimit [39]. We record both the speedup achieved on our simulator, and the speedup predicted by our analytical model based on equation 4.4.3. We also show the cumulative CPU utilization in host configuration where only host CPU is used, and NSC configuration where both host CPU and on-disk CPU is active. Since the System CPU utilization for the NSC configuration is calculated as the sum of on-host CPU and on-disk CPU usage and may be up to 200% of a single CPU core usage.

As the relative CPU speed of on-disk CPU with respect to host-CPU increases, we achieve higher speedup in both simulator and the analytical model. The simulator speedup is lower than the analytical model for higher host-CPU to on-disk CPU configuration as the host-CPU is also involved in sending and receiving the block requested from the underlying device, which is not accounted for in the analytical model.

The CPU utilization for host configuration remains close to maximum host CPU frequency allowed in a configuration. For the NSC configuration, the host CPU utilization

is relatively lower for configurations where CPU does not get saturated (1:1 configuration with 100% host CPU). In a more CPU constrained environment (1:4 configuration with host-CPU set to 25% and on-disk CPU set to 100%), we observe that host CPU continues to remain saturated, and the on-disk CPU remains 100% utilized as well.

4.6.2 Real World Workloads

In Section 4.6.1, we described simple micro-benchmarks to show that our simulator provides speedups predicted by our analytical model presented in Section 4.4. These micro-benchmarks derive speedups based on only one of the 3 factors - latency reduction, bandwidth reduction and compute acceleration. However, as we analyze more complicated applications where the speedup achieved is a function of multiple factors, including systems parameters such as device read-ahead and caching, the analytical model is unable to give realistic estimates of the real world speedups that can be achieved on a Computational Storage Disk. Instead, prototyping the application offload on NESCAFE that retains real systems parameters gives us more accurate speedup estimates for application computation offload on a real computational storage disk.

In this section, we offload three real-world applications that are computationally intensive in NESCAFE. We show how it is simple to deploy such applications on NESCAFE. We show how NESCAFE can be used to evaluate the amount of on-disk CPU power required for a given host-CPU, and the achievable speedup in a Near Storage Compute (NSC) configuration. We also show that not all applications gain performance when their computation is offloaded to the underlying device. Sometimes, host-CPU or transfer time may be a bottleneck for application computation offload. Further, the application computation itself may not be compute intensive enough to gain from an offloade to the on-disk CPU.

We select one benchmark each from the three computation types described in 4.5.3: 1. **zlib compression** - A stream based compression library that consumes high host-CPU. 2. **ext4 journal checksum**, a block based computation intensive application. and 3. **sLSM tree compaction**, an application based computation where file system metadata is transferred to the near-storage CPU and the entire compaction takes place within the drive.

zlib - Compression

Compression is a computation intensive task that has been widely used to reduce amount of data on disk-storage or on data-streams [139]. We port zlib - a popular compression library to Nescafe. zlib compression library works on streamed data i.e. data compression takes place as additional bytes are available. zlib does not have a block based interface. Internally, zlib uses a variant of LZ77 [43] compression algorithm. Compression is done using the `deflate()` function that takes an input and output stream. Data compression is done in an in-memory buffer. The data sent to `deflate()` may or may not result in an

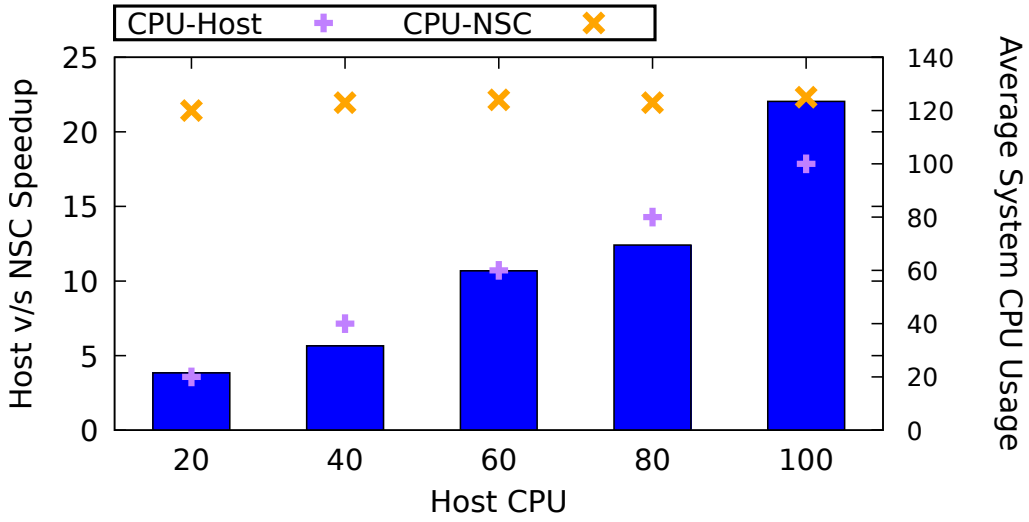


Figure 4.5: *Speedup achieved for 2 GB file compression offload in NSC configuration. We set the on-disk CPU limit to 100% CPU usage and vary the host-CPU. Speedup is achieved with more powerful host-CPU. The Average System CPU Utilization takes both on-disk CPU and host CPU into account and may be up to 200%.*

output being flushed to the disk. Output is only flushed when the in-memory buffer is full or `deflate()` encounters an `EOF` in the input stream.

We split the NSC compression operation into two functions, `compression_init()` and `compression_cleanup()`. The `compression_init()` function initializes two file descriptors. The first file descriptor corresponds to the incoming stream of blocks from the NVMe device process and is directed to the input stream of `deflate()`. The second file descriptor corresponds to the incoming stream of compressed blocks from the `deflate()` function. The second file descriptor is directed back to the NVMe device process. Once compression is completed, the application in the host disables stream compression. This triggers a `NVMe Admin` command to reset stream directive in the device. Next, a call to the `compression_cleanup()` method is made which then sends an `EOF` to the zlib library. This automatically closes the output file descriptor setup in `compression_init()` and writes any remaining in-memory buffer blocks to the device.

Figure 4.5 shows the speedup that we achieve by offloading zlib file compression to the on-disk CPU. A 2GB file is read from a secondary storage media and compressed within the host before being written to the disk. It is then compared with the NSC configuration where the host sends raw data and compression is performed by the NSC process running on the on-disk CPU. For each run, we limit the host-CPU utilisation to a specific value indicated on the X-Axis. We do not limit the on-disk CPU usage in NSC configuration and we observe it always utilizes 100% on-disk CPU. We notice that the speedup is an order of magnitude higher with no host CPU limitation (100% on X Axis). This is because

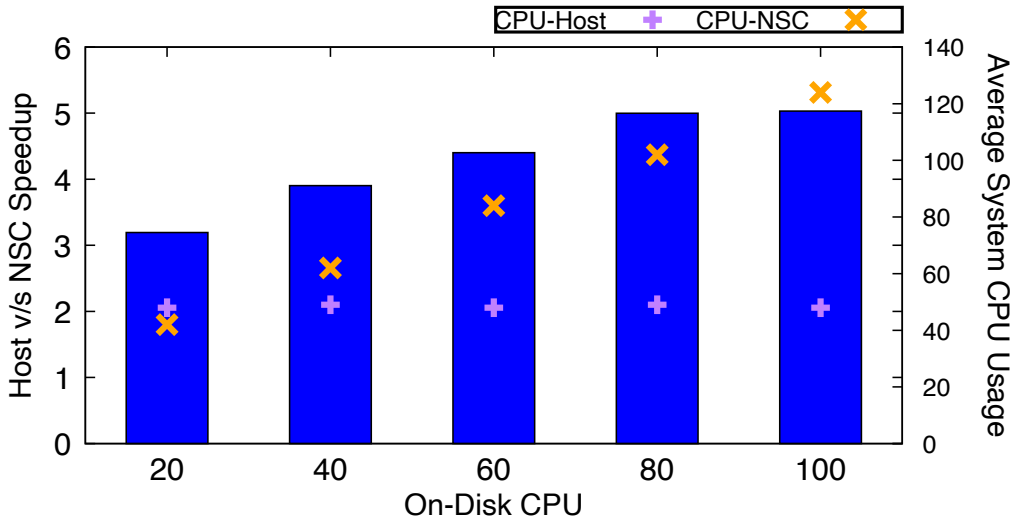


Figure 4.6: *Speedup achieved on 2GB file compression with fixed host CPU usage at 50% and varying on-disk CPU. We observe increased speedup with increased on disk-CPU usage. The Average System CPU Utilization takes both on-disk CPU and host CPU into account and may be up to 200% for NSC configuration.*

in the host configuration, the file read, zlib compression and disk write to the device takes place on the same host-CPU. However in the NSC configuration, the host only reads and sends the data to the device that takes up to 20% host CPU, and the on-disk CPU keeps compressing and writing any data that it receives from the host utilizing 100% CPU.

In host configuration, the CPU utilization remains at its peak with increased host-CPU usage. Hence the host CPU increases from 20% to 100% as we vary the CPU limit on the host. In NSC configuration, the average system CPU utilization (host-CPU and on-disk CPU) remains constant, with host-CPU utilization not increasing more than 20% when compression is offloaded to the device.

In Figure 4.6, we study the impact of increasing on-disk CPU utilization after limiting the the host CPU to 50% of the maximum CPU usage. We vary the on-disk CPU usage on the X-Axis. We observe that even with increased on-disk CPU, there is an increased speedup in the overall compression process.

In the host configuration, the host-CPU utilization is at its peak capacity (50%). For the NSC configuration, the host-CPU utilization remains close to 20% however the on-disk CPU utilization remains at the highest CPU utlization available. Hence the total system CPU utilization ranges from 45%- 120% for the NSC configuration.

We conclude that for zlib compression, an order of magnitude speedup can be achieved by offloading compression to the on-disk processor. Compression speedup is achieved primarily because we decouple data read, data transfer and data compression in NSC configuration. A combination of high powered on-disk CPU and host-CPU power is required to achieve optimal speedup results.

ext4 Journal Checksum

`ext4` [32] is a journal based file system. An atomic record of file-system metadata changes that continues to keep the file system consistent is called a *transaction*. ext4 uses the journal to store all intermediate transitions that take place on the file system. A file system may crash due to power-loss or errors in the underlying media. In such cases, a journal can be used to provide crash consistency in the file system [141].

The journaling mechanism works as follows: First, the file-system metadata is written from the in-memory buffer cache to the journal. Second, the file-system metadata is written from the in-memory buffer cache to the main file system. If the file system crashes, the in memory metadata is lost. However, any previously written valid journal entries are persisted on the journal block device. Before remounting the file system on a reboot, all valid contents of the journal are read back and replayed on the main file system. This ensures metadata integrity in retained on the event of a file system crash.

ext4 data structure	**Description**	**crc32c components**
journal_superblock_s	superblock checksum with s_checksum set to 0.	superblock
journal_block_tag3_s	one tag for each meata-data block is stored in the journal descriptor block.	journal UUID, sequence number and metadata block
jbd2_journal_revoke_header jbd2_journal_revoke_tail	list of metadata blocks journaled earlier but no longer replayed.	journal UUID, revocation block
commit_header	checksum of the commit block	journal UUID, commit block

Table 4.3: *ext4 journal fields that require crc32 checksum computation*

`jbd2` can be maintained on an external block device which drastically improves the file system performance [94, 113, 44, 69]. `jbd2` maintains *transactions* in the form of a circular log. Each transaction contains multiple blocks. A typical transaction contains a journal descriptor block, many data or revocation blocks, and a journal commit block. In order to provide more resliency to the file system against device corruptions, the journal stores checksums in both the journal descriptor block and the journal commit block. Table 4.3 shows the relevant fields on which each checksum is computed. We offload the checksum computation performed on the different journal data-structures to the on-disk CPU in Nescafe. We evaluate performance gains using *filebench* varmail workload [174] that generates a large amount of file system metadata.

We evaluate `jbd2` checksum offload in two setups. First, we evaluate how increasing on-disk processor power can improve checksum computation. Figure 4.7 shows number of *varmail* IO per second (in thousands) with varying on-disk CPU utilization. We note that increased NSC processor speed increases *varmail* performance. The average system CPU utilization does not increase more than 60%.

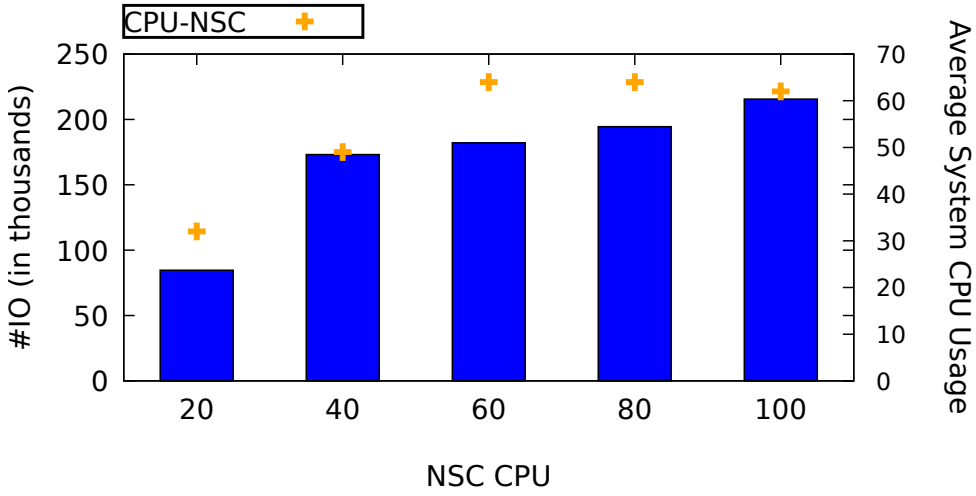


Figure 4.7: *The host-CPU is limited to 50% and the on-disk CPU is varied. varmail I/O per seconds increase with increased on-disk CPU limit. Increasing on-disk CPU power increases varmail I/O performance.*

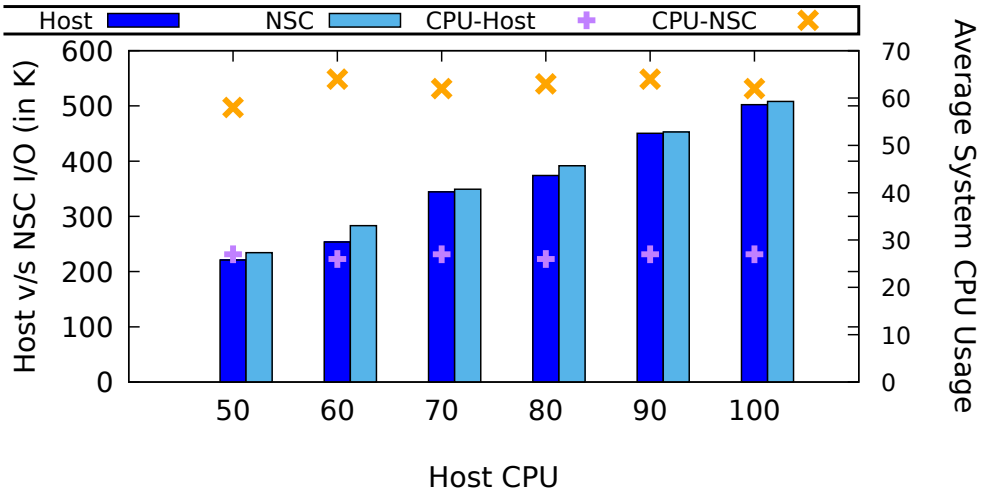


Figure 4.8: *The number of varmail I/O transactions processed with increasing Host CPU usage keeping the on-disk CPU usage constant. The total number of transactions increase with increased host-CPU power, however the relative increase in transactions i.e. speedup does not increase with more powerful host-CPU.*

Second, we evaluate Host v/s NSC performance. In Figure 4.8, we notice that there are no speedup gains in host v/s NSC computation. The amount of CPU host uses is 25%. As the CPU limit of host increases, we observe increased I/Os in both host and NSC configuration. But the relative speedup between host and NSC configuration does not change. This shows that although a powerful host CPU increases the number of I/Os, a relative speedup increase may not be possible in NSC configuration. This shows that journal checksum computation is not a processor intensive computation and is not an ideal candidate for computation offload to a Computational Storage Device.

LSM Tree Compaction

Log-Structured Merge (LSM) Tree [132] is a data structure commonly used in key-value stores. A LSM Tree has two main components: 1) an in-memory buffer to increase random key inserts and delete throughput, and 2) on-disk log files that store key value pair tuples sorted by keys.

LSM tree is multi-layered, with each layer storing key-value pairs written down from the higher layers. All key-value pair inserts are first stored in the in memory component of the LSM tree called the `L0` Layer. A deleted key is also stored as a key-value pair where the key's value is set to a reserved number called a *tombstone* [16, 162]. All key-value pairs are sorted by keys. Whenever a threshold on the total number of keys in `L0` Layer is reached, its contents are written to a new file on disk. Each file created on the disk is called a *run*. After a certain number of *runs* are created on the `L1` layer, all key-value pairs in each run in `L1` layer are sorted and merged to form a single run in the subsequent `L2` layer. Any previously deleted key recorded as a tombstone value is removed during the merge operation. As a result of multiple runs in top layer merging into single run in the subsequent layer, the run file size increases in each subsequent layer.

The mechanism of sorting and merging multiple runs in the upper layer into a single run in the lower layer is called *compaction*. Compaction is a computation intensive activity where sorted key-value pairs in multiple runs in the previous layer are merged into a sorted single run in the next layer. The sorting is done using a priority queue. If the value of the input key-value pair from previous *run* is a *tombstone* value, we ignore the key. If the key has a valid value, we store the key-value pair in the output run in the subsequent level. Any active I/O on the key-value store is paused when compaction is taking place to ensure consistency. This often leads to performance variability in KV-stores [197].

We evaluate offloading compaction operation of a skip-list based LSM Tree (sLSM Tree) [170] on Nescafe. sLSM tree uses a skipList based in-memory data structure to store key-value pairs. Once the skipList is full, it is written to `L1` on-disk level in the form of a single run file. If a number of run files on `L1` level exceed maximum number of levels allowed (default 4), all runs in `L1` are compacted and output as a single new run file on `L2`. This compaction continues until a) reach the last allowed level in the LSM tree, or b) the number of runs in the level are less than the maximum allowed runs.

We offload the `heapmerge` computation function [170] in NSC configuration to the underlying on-disk CPU. The Heapmerge function requires the input run files from the input layer and pre-allocated output run file in the output layer. In the host configuration, the merge operation is called whenever the last run in the input layer has reached its maximum capacity. Once the last run is full, all runs in the input layear are merged and written into a single run on the underlying disk. All runs in sLSM tree are pre-allocated and `mmap`ed in memory for faster access. Further, no additional file-system metadata is required to identify individual blocks of the input file and scan through the key-value records.

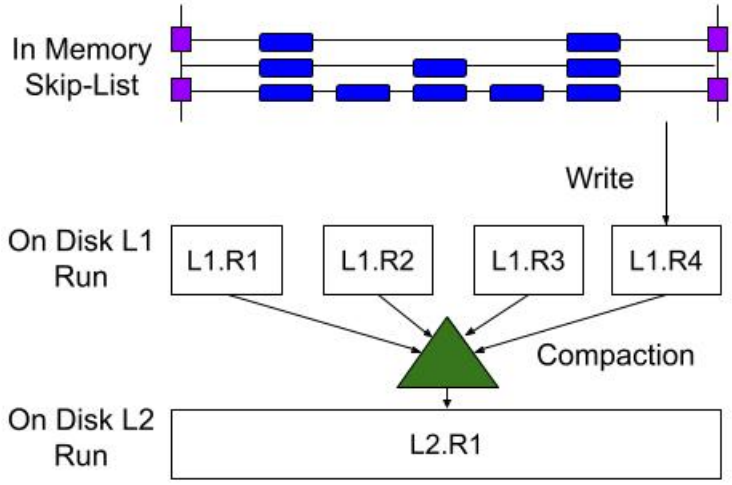


Figure 4.9: *skipList Log structured Merge Tree workflow. In-memory component is a linked list with bloom filters (purple) for fast access. Each flush from in-memory skipList creates a new run file in Level0 (`L0`). When `L0` has 4 runs, they are compacted (merged) to a single run in the next level `L1`. This continues to subsequent levels until the last level or a level with runs less than maximum allowed runs is reached.*

However, in a NSC configuration, the underlying disk does not have any knowledge of the input run file structure. The logical continuous nature of input files in the host cannot be assumed inside the disk due to fragmentation and unique file system allocation policies. In order to identify and scan through each input file, the file system metadata needs to be transferred to the underlying device to perform heap-merge operation.

The following operations are performed to inform the underlying disk about merge operation during compaction: 1) All `mmap`ed runs are flushed to disk and page cache is cleared. We clear the page cache to disk so that the application reloads the on-disk modified run files after compaction into memory again. If the page cache is not cleared, stale key-value pairs are read by the application. 2) The LBAs of all input run files and the preallocated output run file is identified using the hdparm tool and is sent to the computational disk. 3) all input run key-value pairs are sorted and written to the output run. 4) we remap the input and output runs in the application to get the updated merged runs.

Unlike sending data blocks or data streams through file-descriptor in zlib compression or jbd2 checksum, we use the file descriptor to send file system metadata to the underlying device. First, we open a file descriptor and set its computation type to `LSM_MERGE`. This makes the NSC process aware that the subsequent blocks will contain identifying file system metadata to perform LSM tree compaction. Second, we write a sequence of input file block ranges that we derived from the hdparm tool to this file descriptor. We then send the pre-allocated output file block range. Finally, the host waits until compaction is completed and the last offset of the output run is returned to the host. The resultant

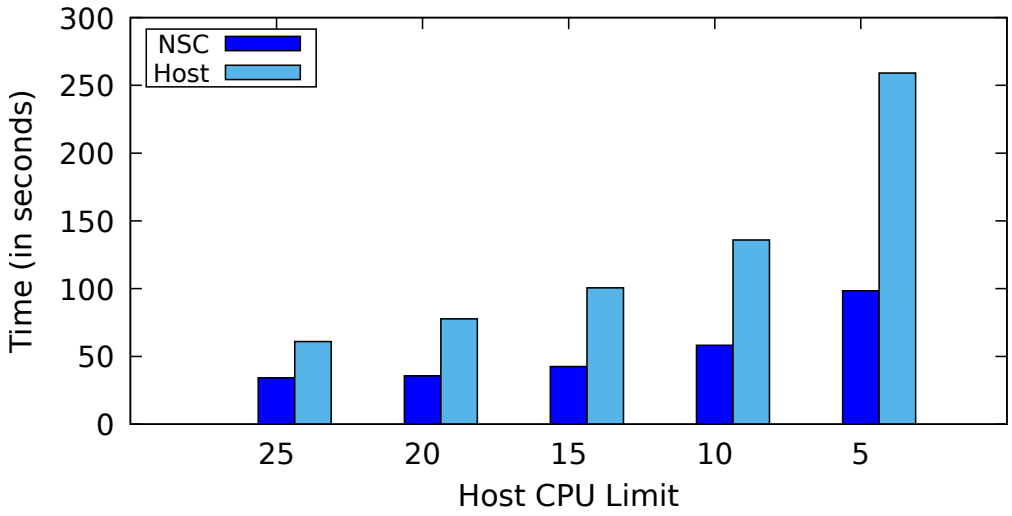


Figure 4.10: *LSM Tree compaction overall time for insert-delete-lookup workflow. We vary the host-CPU in both Host and NSC mode and do not bottleneck the on-disk CPU in NSC configuration. The merge operation performs better in NSC configuration than host Configuration in a CPU constrained environment.*

last offset of the output run file after compaction may be smaller than the earlier pre-allocated output run size as deleted keys are eliminated from the resultant output run during compaction. The returned output file run offset from the disk is used to reset the output run file by the host if required.

We note that sLSM tree's compaction does not have a significant amount of CPU usage. This is different from prior work [197] where FPGA's are used for compaction. The speedup on FPGA's is achieved as a combination of pipelining of various workflows such as decoding a prefix-encoded key, performing compaction and then encoding the resultant keys. For sLSM tree, we find that the compaction operation is not extremely compute intensive - it consumes 30% of host CPU. We vary the amount of host-CPU utilization and see the impact on insert-delete and a subsequent lookup.

Figure 4.10 shows the amount of time taken in the host-CPU and the NSC configuration for the insert-delete-lookup benchmark in the sLSM Tree. The benchmark inserts 1 million integer key-value pairs into the LSM tree and subsequently updates and deletes a fraction of the original set of inserted keys. Each L1 run contains 512 Key-Value pairs. Each layer contains 4 runs. As we step down from a higher to a lower layer, the run size quadruples to accommodate compacted runs from the previous layer.

We compare the total time to perform compaction in host configuration and NSC configuration. We observe that as we reduce the limit of the host-CPU, the Host configuration performance degrades significantly. For a similar host-CPU usage, offloading the compaction task to the on-Disk CPU in NSC configuration continues to keep the total operation time lower. Hence we get good speedup even with lower host-CPU usage for LSM tree compaction.

4.7 Power Aware Computational Storage Disks

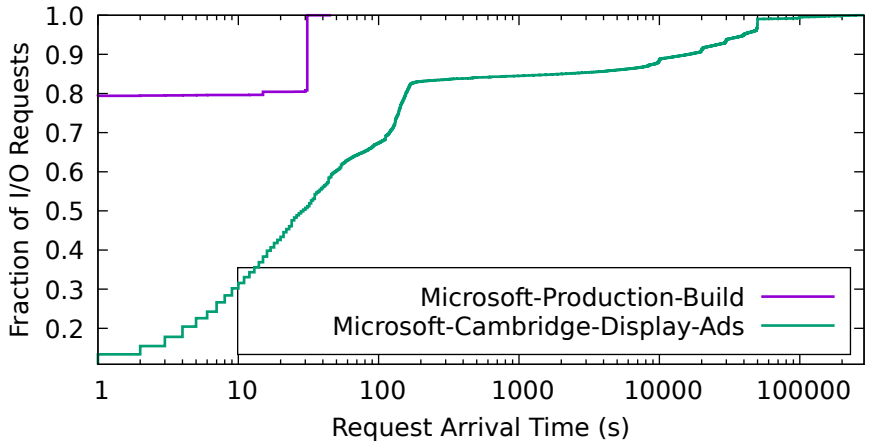
Deployment of Computational Storage Disks on a large scale in data-centers brings additional challenges. One of the challenges is the amount of additional power consumed due to the presence of an additional on-disk CPU. A number of efforts in large scale data-centers have been made to measure and reduce the power consumption of host servers [148, 146, 51, 13] and auxiliary computation devices in the data -center [88, 130].

The Case for Energy-Proportional Computing [13] discusses high energy costs in terms of total cost of ownership (TCO) and CO2 emissions. The paper shows that most workloads in their data-center (Google) use between 10-50% of the maximum CPU utilization. Further, active servers consume 50% of CPU power, even when there is 0% CPU utilization. CPU is the biggest power consuming component of a server. CPU's inside the Google servers studied in [13] consumed 45% of the total CPU power. Efforts such as clock gating and dynamic frequency scaling can drastically help improve the total power utilization of the data-center.

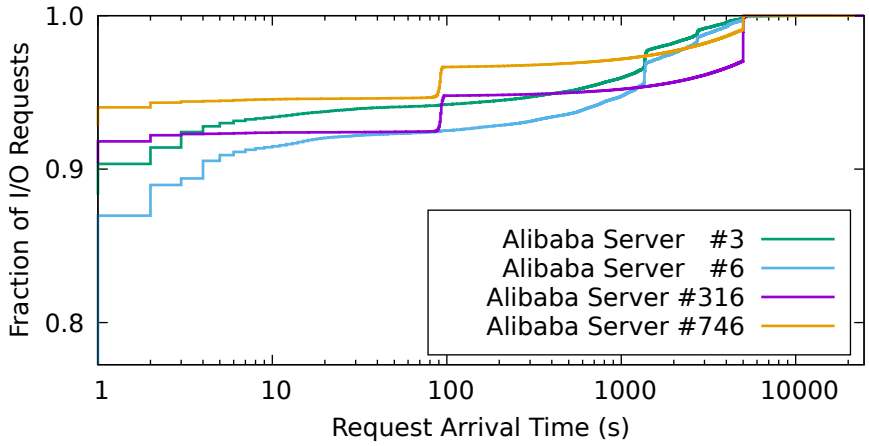
In [148] a new blade architecture is discussed where the power is managed and enforced at the level of the enclosure (or the chassis). Such an architecture recognizes trends across multiple systems and implements power effectiveness at a larger scale. In [146], authors propose coordinated power delivery, heat management and electricity consumption to reduce the amount of overall power consumed by the data-center. In Power Provisioning for a Warehouse-sized Computer [51] the power consumed by over 15 thousand servers is analyzed. The servers host 3 types of workloads - Websearch, Webmail and Map-reduce. The paper concludes that in field, 7-16% lower power than theoretically estimated power consumption is utilized in the real world and this power can be used to run more workloads on each server.

Although these studies focus on host-side active servers that have to remain powered-on all the time, on-disk cores that perform computation on the event of receiving an I/O request need not be active all the time. In-fact, in periods of long inactivity, the disks can be powered-down. This power-awareness can be built inside the Computational Storage SSD. On the event of idle cycles or no-active workload being written to the drive, the disk cores can be turned-off entirely or run in low-power mode using a CPU governor to reduce power usage.

In order to evaluate the usefulness of this approach, we analyse the disk I/O request arrival times of two storage data center workloads - Microsoft and Alibaba. In Figure 4.11a, we notice two trends in I/O requests. For the Microsoft build server, around 85% of the requests arrive within a second of the previous request. However, the remaining I/O (15%) takes 30-35 seconds to arrive before the previous request. For the display ads data server we see that the disk may not receive an I/O request for 30000 seconds (8 hours). Figure 4.11b shows the arrival time of requests for four storage disks running in the Alibaba Cloud. We note that 5% of the disk requests arrive after 100 or more seconds of disk inactivity. For disks serving intermittent burst traffic followed by long time intervals of



(a) *Microsoft I/O Request Intervals*



(b) *Alibaba I/O Request Intervals*

Figure 4.11: *I/O request arrival distribution for 4.11a Microsoft and 4.11b Alibaba Data Centers. Most requests arrive within a second of the previously sent I/O request.*

inactivity, we can regulate or completely turn of the on-disk CPU core after a few seconds of inactivity and save power.

Although the disks power-down for 5-10% of I/O requests, the total time duration for which the CPU is turned off will be much higher. For instance, if 95% of 1 million requests take less than 1 second to arrive, and 5% of the requests take 10 or more seconds, the total time to serve the requests is more than 1.45 million seconds. If the disk stays idle and powered off or in a low power consumption after a second of inactivity, the disk will save power for atleast 0.5 million seconds or 33% of the total time. Hence with a power-aware Computational Storage Disk, the total power consumption can be significantly reduced, especially for data-centers where there are burst requests with intermittent long duration of inactivity.

4.7.1 Related Work

Our work is closest in spirit to XRP [205, 204] that uses BPF to push light-weight storage functions into the Linux NVMe driver. XRP has shown great gains on a custom Key-Value store and WiredTiger database. However, replacing the entire kernel storage stack is neither desirable nor feasible due to limitations with BPF [38].

A number of other approaches have been suggested for accelerating computation by using auxiliary hardware. ASIC processors provide great opportunities with order of magnitude improvements for image and video compression [147, 8]. Unfortunately, ASIC processors are non-programmable and can only be designed for specific applications. For example [147] is designed specifically for neural network based image compression whereas [8] focusses on optimizing Fractal image compression.

StreamBox [116] and StreamBox-HBM [115] propose a model of performing transformation of streamed data that may arrive out of order. They exploit parallelism and memory hierarchy of out of memory hardware. The near compute storage is used at a lower level of storage hierarchy and can work complementary to in-memory computation approaches. The advantage of performing computation at storage is less data is forwarded to in-memory structures for processing thereby saving High bandwidth memory (HBM) for other critical data such as metadata.

Accelerating computation over auxiliary processors such as Graphic Processor Units (GPUs) [140, 80, 138], ASICs [147, 8], FPGAs [14] and TPUs [87, 86] have also been widely explored. Our current approach does not aim to replace such hardware based approaches but complement them. For example, computation that needs to be done on TPUs can first be performed on a near storage compute framework before it is offloaded to a Tensor Processor Unit.

Approximate Storage [159, 149, 82, 106, 125] proposes relaxing correctness in storing media friendly data on the underlying media and computing the correctness on demand. For e.g., most data structures employ correction strategy already to recover from bit level media errors. This helps writing media friendly data to the underlying storage, especially Solid-state memories that have differential voltage levels for different types of data being written. We see near storage computation as an opportunity to accelerate approximate storage. We could compute exact data from approximate storage by using mechanisms such as Error Correction Codes (ECC) within the media prior to returning the exact data to the host. Such computation performed within the storage reduces the computation required on the host computation unit or CPU.

4.8 Conclusion and Future Work

We propose a software-only framework to evaluate application computation offload to a near storage computational device. We present both an analytical mathematical model and a full system implementation that describes how applications can leverage computa-

tion offload to achieve higher performance. We show with both micro-benchmarks and real world applications that it is easy to realise and evaluate application offload using Nescafe. We also show that not all applications benefit from offload or a more powerful on-disk CPU. In particular, we demonstrate that a) zlib compression can get order of magnitude speedup using a Near Storage Computation disk. This speedup is largely due to raw data read and transfer being decoupled from writing data to the disk rather than a powerful on-disk processor. b) jbd2 checksum computation and metadata intensive operations such as varmail are entirely dependent on the host-CPU and give negligible speedups when offloaded to a Near Storage Computation engine. Finally, c) LSM tree compaction is not a compute intensive task and gives high speedup only in CPU constrained environments.

To summarize, Nescafe helps us to understand the amount of speedup that can be achieved, the on-disk CPU power required to obtain speedup and whether an application can benefit from the use of a Computational Storage Device. Nescafe framework opens a range of possibilities to conduct software-only computational offload investigation and research:

1. **Revisiting IPC interface:** Our current Nescafe model assumes that hardware accelerators in SSDs have a disaggregated memory model - there is no shared memory between the device process and the near storage computation process. We rely on named pipes to perform inter-process communication. Inter-process communication over named pipes are less efficient than shared memory communication. We would like to explore this research direction for Nescafe.

2. **Multithreaded Application Offload:** Our current Model has a uni-threaded computation process that can only process one I/O block at a time. Extending this model to real-world applications such as RocksDB that do multi-threaded compaction would help expand the range of applications that Nescafe can support.

3. **High Level Language Application Support:** A number of in-memory, big-data processing applications such as spark are written in scala or other high-level JVM based runtime environment. Porting such applications to a C based testbed is challenging, because of limitations in transferring Java programs to C or using JNI [81] interface.

Chapter 5

Summary and Future Work

This thesis presents the reliability and performance challenges and solutions to evolving Solid State Drive media. As SSD media gets denser due to huge space and low operating cost requirements, the errors in such media will only increase over time. Furthermore, the reliability of SSDs are largely dependent on the number of erase operations the drive encounters. The reduction in the P/E cycle limit as SSDs get denser is a cause of concern. Hence design, implementation and evaluation of novel coding schemes that reduce erase operations by overwriting on previous data is the need of the hour.

From a performance perspective, devices such as NIC cards and SSDs are being equipped with specialized on-chip or on-disk CPUs. We have presented a software only mechanism to evaluate the speedup gains one can obtain from such computational storage disks. We have also developed a framework that makes it easy to offload real world block, stream and entire application workflows to the underlying Computational Storage Device.

This thesis forms a basis for the following future research directions:

Storage Reliability Evaluation Frameworks

In this thesis, we present a file-system focussed error injection framework specifically in the context of SSDs. An in-depth study of other types of emerging media, such as ZNS SSDs [18], Persistent Media [194] and Glass [9] can open a range of possibilities in enhancing our error injection framework. The file systems we analyzed can be readily checked for errors on other media specific errors once the error injection mechanisms are enhanced. On the software front, we can analyze a range of other block based software such as bare-metal hypervisors [12] and bare-metal key-value stores [5] that run without the presence of an intermediate file system layer while interacting with the SSD.

Improving SSD Hardware Reliability

The Voltage based Write-Once-Memory codes presented in this work treats underlying SSD cells as monotonically increasing voltage based media. Although the code is high-performance with encode and decode mechanisms requiring only a single table lookup, the code is not space optimal - i.e. there is significant space overheads in using WOM-v codes. It would be interesting to revisit other, low-performance codes that require multiple retires but are space optimal in the context of SSDs being treated as voltage based media instead of groups of cells. The WOM-v coder framework presented in this thesis can be readily extended to do such experiments on other coding schemes without any modifications to the remaining LightNVM and FEMU workflow. Further, in this thesis we limit our gains to QLC drives, but our framework can be easily extended to PLC or higher density drives where WOM-v code gains would be much higher.

Performance aware Computational Storage Offload

The Near Storage Computational Framework (Nescafe) presented in this thesis evaluates application computation offload to a PCI-e connected on-disk CPU for a single node storage medium. We can extend Nescafe to operate on distributed file systems and volume managers, object stores and key-value stores to asses gains in large-scale, distributed computation offload. The PCI-e latency can be increased to reflect TCP/IP or RDMA latency. Our evaluation considers the storage substrate to be a MLC SSD. Extending the gains of computational storage to a QLC SSD or Persistent media could be easily done my changing the latency configuration of the underlying media. Further, the current implementation of Nescafe considers only one on-disk CPU core. Multiple cores within the same disk could be added to run multi-threaded computation offloads that are prevalent in advanced storage systems such as RocksDB. Hence the Nescafe framework can be extended in multiple research directions based on the real-world environment that we want to test the application computation offload gains.

Bibliography

[1] *4 QLC workloads and why they're a good fit for QLC NAND flash*. https://www.techtarget.com/searchstorage/tip/4-QLC-workloads-and-why-they-are-a-good-fit-for-QLC-NAND-flash. 2022.

[2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. "Active Disks: Programming Model, Algorithms and Evaluation". In: *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VIII. San Jose, California, USA: Association for Computing Machinery, 1998, pp. 81–91. ISBN: 1581131070. DOI: 10.1145/291069.291026. URL: https://doi.org/10.1145/291069.291026.

[3] Ian F. Adams, John Keys, and Michael P. Mesnier. "Respecting the block interface – computational storage using virtual objects". In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotstorage19/presentation/adams.

[4] "Adaptive Placement for In-memory Storage Functions". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/bhardwaj.

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

[6] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. "A Five-Year Study of File-System Metadata". In: *5th USENIX Conference on File and Storage Technologies (FAST 07)*. San Jose, CA: USENIX Association, Feb. 2007. URL: https://www.usenix.org/conference/fast-07/five-year-study-file-system-metadata.

[7] 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 '08)*. Vol. 57. 2008.

[8] F. Ancarani, D. De Gloria, M. Olivieri, and C. Stazzone. "Design of an ASIC architecture for high speed fractal image compression". In: *Proceedings Ninth Annual IEEE International ASIC Conference and Exhibit*. 1996, pp. 223–226. DOI: 10.1109/ASIC.1996.551998.

[9] Patrick Anderson, Richard Black, Ausra Cerkauskaite, Andromachi Chatzieleftheriou, James Clegg, Chris Dainty, Raluca Diaconu, Rokas Drevinskas, Austin Donnelly, Alexander L. Gaunt, Andreas Georgiou, Ariel Gomez Diaz, Peter G. Kazansky, David Lara, Sergey Legtchenko, Sebastian Nowozin, Aaron Ogus, Douglas Phillips, Antony Rowstron, Masaaki Sakakura, Ioan Stefanovici, Benn Thomsen, Lei Wang, Hugh Williams, and Mengyang Yang. "Glass: A New Media for a New Era?" In: *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. Boston, MA: USENIX Association, July 2018. URL: https://www.usenix.org/conference/hotstorage18/presentation/anderson.

[10] L. N. Bairavasundaram, M. Rungta, N. Agrawa, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. "Analyzing the effects of disk-pointer corruption". In: *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. Vol. 00. 2008, pp. 502–511. DOI: 10.1109/DSN.2008.4630121.

[11] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. "An Analysis of Data Corruption in the Storage Stack". In: *ACM Transactions on Storage (TOS)* 4.3 (2008), p. 8.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the art of virtualization". In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 164–177.

[13] Luiz André Barroso and Urs Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40.12 (2007), pp. 33–37. DOI: 10.1109/MC.2007.443.

[14] Matěj Bartík, Sven Ubik, and Pavel Kubalik. "LZ4 compression algorithm on FPGA". In: *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*. IEEE. 2015, pp. 179–182.

[15] Hanmant P Belgal, Nick Righos, Ivan Kalastirsky, Jeff J Peterson, Robert Shiner, and Neal Mielke. "A new reliability model for post-cycling charge retention of flash memories". In: *Proceedings of the 40th Annual International Reliability Physics Symposium*. IEEE. 2002, pp. 7–20.

[16] Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C Kuszmaul, Donald E Porter, Jun Yuan, and Yang Zhan. "An Introduction to B-trees and Write-Optimization". In: *login; magazine* 40.5 (2015).

[17] Nicolas Bitouzé, Alexandre Graell i Amat, and Eirik Rosnes. "Using short synchronous WOM codes to make WOM codes decodable". In: *IEEE transactions on communications* 62.7 (2014), pp. 2156–2169.

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

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

[20] Simona Boboila and Peter Desnoyers. "Write Endurance in Flash Drives: Measurements and Analysis". In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*. USENIX Association, 2010, pp. 115–128.

[21] Adam Brand, Ken Wu, Sam Pan, and David Chin. "Novel read disturb failure mechanism induced by FLASH cycling". In: *Proceedings of the 31st Annual International Reliability Physics Symposium*. IEEE. 1993, pp. 127–132.

[22] *Btrfs Bug Report*. https://bugzilla.kernel.org/show_bug.cgi?id=198457. 2019.

[23] *Btrfs mkfs man page*. https://btrfs.wiki.kernel.org/index.php/Manpage/mkfs.btrfs. [Online; accessed 23-Oct-2019]. 2019.

[24] Sarit Buzaglo and Tuvi Etzion. "Tilings With n-dimensional chairs and their applications to asymmetric codes". In: *IEEE transactions on information theory* 59.3 (2012), pp. 1573–1582.

[25] 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* 105.9 (2017), pp. 1666–1704.

[26] Yu Cai, Saugata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich F Haratsch. "Vulnerabilities in MLC NAND flash memory programming: experimental analysis, exploits, and mitigation techniques". In: *23rd International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2017, pp. 49–60.

[27] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. "Error patterns in MLC NAND flash memory: Measurement, Characterization, and Analysis". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. EDA Consortium. 2012, pp. 521–526.

[28] Yu Cai, Yixin Luo, Erich F Haratsch, Ken Mai, and Onur Mutlu. "Data retention in MLC NAND flash memory: Characterization, optimization, and recovery". In: *21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2015, pp. 551–563.

[29] Yu Cai, Onur Mutlu, Erich F Haratsch, and Ken Mai. "Program interference in MLC NAND flash memory: Characterization, modeling, and mitigation". In: *31st International Conference on Computer Design (ICCD)*. IEEE. 2013, pp. 123–130.

[30] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai. "Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime". In: *30th International Conference on Computer Design (ICCD)*. IEEE. 2012, pp. 94–101.

[31] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. "PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems". In: *Proceedings of the 2018 International Conference on Supercomputing*. ACM. 2018, pp. 1–11.

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

[33] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. "POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database". In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 29–41. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/cao-wei.

[34] Paolo Cappelletti, Roberto Bez, Daniele Cantarelli, and Lorenzo Fratin. "Failure mechanisms of Flash cell in program/erase cycling". In: *Proceedings of the IEEE International Electron Devices Meeting*. IEEE. 1994, pp. 291–294.

[35] Yuval Cassuto and Eitan Yaakobi. "Short Q-ary fixed-rate WOM codes for guaranteed rewrites and with hot/cold write differentiation". In: *IEEE transactions on information theory* 60.7 (2014), pp. 3942–3958.

[36] Feng Chen, David A. Koufaty, and Xiaodong Zhang. "Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives". In: *Proceedings of the 2009 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*. Seattle, WA, 2009, pp. 181–192. DOI: 10.1145/1555349.1555371. URL: http://doi.acm.org/10.1145/1555349.1555371.

[37] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "Consistency without ordering". In: *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*. USENIX Association. 2012, pp. 9–9.

[38] Johnathan Corbett. *Bounded Loops in BPF Programs*. Linux Weekly News. Linux Plumbers Conference, 2018.

[39] *CPU Limit*. Linux CPU Limit Utility. URL: https://manpages.ubuntu.com/manpages/xenial/man1/cpulimit.1.html.

[40] Robin Degraeve, F Schuler, Ben Kaczer, Martino Lorenzini, Dirk Wellekens, Paul Hendrickx, Michiel van Duuren, GJM Dormans, Jan Van Houdt, L Haspeslagh, et al. "Analytical percolation model for predicting anomalous charge loss in flash memories". In: *IEEE Transactions on Electron Devices* 51.9 (2004), pp. 1392–1400.

[41] Peter Desnoyers. "Analytic models of SSD write performance". In: *Proceedings of the 5th Annual International Systems and Storage Conference*. 2012.

[42] Peter Desnoyers. "What Systems Researchers Need to Know about NAND Flash". In: *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*. HotStorage'13. San Jose, CA: USENIX Association, 2013, p. 6.

[43] P. Deutsch. *RFC1951: DEFLATE Compressed Data Format Specification Version 1.3*. USA, 1996.

[44] Borislav Djordjevic and Valentina Timcenko. "Ext4 file system in linux environment: Features and performance analysis". In: *International Journal of Computers* 6.1 (2012), pp. 37–45.

[45] *dm-inject: A device mapper error Injector framework*. URL: https://github.com/uoftsystems/dm-inject.

[46] David Domingo and Sudarsun Kannan. "pFSCK: Accelerating File System Checking and Repair for Modern Storage". In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 113–126. ISBN: 978-1-939133-20-5. URL: https://www.usenix.org/conference/fast21/presentation/domingo.

[47] Jake Edge. *File-level Integrity*. https://lwn.net/Articles/752614/. [Online; accessed 23-Oct-2019]. 2018.

[48] *F2FS Bug Report*. https://bugzilla.kernel.org/show_bug.cgi?id=200635. 2019.

[49] *F2FS Bug Report - Write I/O Errors*. https://bugzilla.kernel.org/show_bug.cgi?id=200871. 2019.

[50] *F2FS Patch File*. https://sourceforge.net/p/linux-f2fs/mailman/message/36402198/. 2019.

[51] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. "Power Provisioning for a Warehouse-Sized Computer". In: *SIGARCH Comput. Archit. News* 35.2 (June 2007), 13–23. ISSN: 0163-5964. DOI: 10.1145/1273440.1250665. URL: https://doi.org/10.1145/1273440.1250665.

[52] Amos Fiat and Adi Shamir. "Generalized'write-once'memories". In: *IEEE Transactions on Information Theory* 30.3 (1984), pp. 470–480.

[53] Hilary Finucane, Zhenming Liu, and Michael Mitzenmacher. "Designing floating codes for expected performance". In: *2008 46th Annual Allerton Conference on Communication, Control, and Computing*. IEEE. 2008, pp. 1389–1396.

[54] Daniel Fryer, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. "Recon: Verifying File System Consistency at Runtime". In: *ACM Transactions on Storage (TOS)* 8.4 (Dec. 2012), 15:1–15:29. ISSN: 1553-3077. DOI: 10.1145/2385603.2385608. URL: http://doi.acm.org/10.1145/2385603.2385608.

[55] *fs-verity: File System-Level Integrity Protection*. https://www.spinics.net/lists/linux-fsdevel/msg121182.html. [Online; accessed 23-Oct-2019]. 2019.

[56] Fang-Wei Fu and AJ Han Vinck. "On the capacity of generalized write-once memory with state transitions described by an arbitrary directed acyclic graph". In: *IEEE Transactions on Information Theory* 45.1 (1999), pp. 308–313.

[57] Tomonori Fujita and Mike Christie. "tgt: Framework for storage target drivers". In: *Proceedings of the Linux Symposium*. Vol. 1. Citeseer. 2006, pp. 303–312.

[58] Ryan Gabrys and Lara Dolecek. "Constructions of nonbinary WOM codes for multilevel flash memories". In: *IEEE Transactions on Information Theory* 61.4 (2015), pp. 1905–1919.

[59] Eran Gal and Sivan Toledo. "Algorithms and Data Structures for Flash Memories". In: *ACM Computing Survey (CSUR)* 37.2 (June 2005), pp. 138–163. ISSN: 0360-0300. DOI: 10.1145/1089733.1089735.

[60] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions". In: *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. Santa Clara, CA: USENIX Association, 2017, pp. 149–166. ISBN: 978-1-931971-36-2. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan.

[61] Om Rameshwar Gatla, Mai Zheng, Muhammad Hameed, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojevic, Cyril Guyot, and Robert Mateescu. "Towards Robust File System Checkers". In: *ACM Trans. Storage* 14.4 (Dec. 2018), 35:1–35:25. ISSN: 1553-3077. DOI: 10.1145/3281031. URL: http://doi.acm.org/10.1145/3281031.

[62] *Github Code Repository*. https://github.com/uoftsystems/dm-inject. 2019.

[63] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung. "Amber*: Enabling Precise Full-System Simulation with Detailed Modeling of All SSD Resources". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 469–481.

[64] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. "Characterizing Flash Memory: Anomalies, Observations, and Applications". In: *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2009, pp. 24–33. DOI: 10.1145/1669112.1669118.

[65] Laura M Grupp, John D Davis, and Steven Swanson. "The bleak future of NAND flash memory". In: *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*. USENIX Association. 2012.

[66] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. "Biscuit: A Framework for Near-Data Processing of Big Data Workloads". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 153–165. DOI: 10.1109/ISCA.2016.23.

[67] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. "EIO: Error Handling is Occasionally Correct". In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*. San Jose, CA, 2008, 14:1–14:16.

[68] 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: *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 1–14. ISBN: 978-1-931971-42-3. URL: https://www.usenix.org/conference/fast18/presentation/gunawi.

[69] Dave Hitz, James Lau, and Michael A Malcolm. "File System Design for an NFS File Server Appliance". In: *USENIX winter*. Vol. 94. 1994.

[70] Qin Huang, Shu Lin, and Khaled AS Abdel-Ghaffar. "Error-correcting codes for flash coding". In: *IEEE transactions on information theory* 57.9 (2011), pp. 6097–6108.

[71] S Hur, J Lee, M Park, J Choi, K Park, K Kim, and K Kim. "Effective program inhibition beyond 90nm NAND flash memories". In: *Proc. NVSM* (2004), pp. 44–45.

[72] Zsolt István, David Sidler, and Gustavo Alonso. "Caribou: Intelligent Distributed Storage". In: *Proc. VLDB Endow.* 10.11 (Aug. 2017), pp. 1202–1213. ISSN: 2150-8097. DOI: 10.14778/3137628.3137632. URL: https://doi.org/10.14778/3137628.3137632.

[73] Shehbaz Jaffer. "Reclaiming good transactions from a corrupt journal". In: (2019).

[74] Shehbaz Jaffer. "WOM-v Code Simulator". In: 2021.

[75] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. "Improving the Endurance of Next Generation SSD's Using WOM-v Codes". In: *ACM Trans. Storage* 18.4 (2022). ISSN: 1553-3077. DOI: 10.1145/3565027. URL: https://doi.org/10.1145/3565027.

[76] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. "Improving the Reliability of Next Generation SSDs using WOM-v Codes". In: *20th USENIX Conference on File and Storage Technologies (FAST 22)*. Santa Clara, CA: USENIX Association, Feb. 2022, pp. 117–132. ISBN: 978-1-939133-26-7. URL: https://www.usenix.org/conference/fast22/presentation/jaffer.

[77] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. "Rethinking WOM Codes to Enhance the Lifetime in New SSD Generations". In: *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/hotstorage20/presentation/jaffer.

[78] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. "Evaluating file system reliability on solid state drives". In: *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 2019, pp. 783–798.

[79] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. "The Reliability of Modern File Systems in the Face of SSD Errors". In: *ACM Trans. Storage* 16.1 (Mar. 2020). ISSN: 1553-3077. DOI: 10.1145/3375553. URL: https://doi.org/10.1145/3375553.

[80] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. "Heterogeneous isolated execution for commodity gpus". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 455–468.

[81] *Java Native Interface*. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/.

[82] Djordje Jevdjic, Karin Strauss, Luis Ceze, and Henrique S. Malvar. "Approximate Storage of Compressed and Encrypted Videos". In: *SIGPLAN Not.* 52.4 (2017), 361–373. ISSN: 0362-1340. DOI: 10.1145/3093336.3037718. URL: https://doi.org/10.1145/3093336.3037718.

[83] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. "Improving SSD lifetime with byte-addressable metadata". In: *Proceedings of the International Symposium on Memory Systems*. 2017, pp. 374–384.

[84] Seok Jin Joo, Hea Jong Yang, Keum Hwan Noh, Hee Gee Lee, Won Sik Woo, Joo Yeop Lee, Min Kyu Lee, Won Yol Choi, Kyoung Pil Hwang, Hyoung Seok Kim, et al. "Abnormal disturbance mechanism of sub-100 nm NAND flash memory". In: *Japanese Journal of Applied Physics* 45.8R (2006), p. 6210.

[85] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[86] Norman P. Jouppi, Cliff Young, Nishant Patil, and David A. Patterson. "A domain-specific architecture for deep neural networks". In: *Commun. ACM* 61.9 (2018), pp. 50–59. DOI: 10.1145/3154484. URL: https://doi.org/10.1145/3154484.

[87] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 2017, pp. 1–12. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246.

[88] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), 1–12. ISSN: 0163-5964. DOI: 10.1145/3140659.3080246. URL: https://doi.org/10.1145/3140659.3080246.

[89] Myoungsoo Jung and Mahmut Kandemir. "Revisiting Widely Held SSD Expectations and Rethinking System-level Implications". In: *Proceedings of the 2013 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '13)*. Pittsburgh, PA, 2013, pp. 203–216. ISBN: 978-1-4503-1900-3. DOI: 10.1145/2465529.2465548. URL: http://doi.acm.org/10.1145/2465529.2465548.

[90] Arpith K and K. Gopinath. "Need for a Deeper Cross-Layer Optimization for Dense NAND SSD to Improve Read Performance of Big Data Applications: A Case for Melded Pages". In: *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/hotstorage20/presentation/k.

[91] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. "Characterization of storage workload traces from production windows servers". In: *2008 IEEE International Symposium on Workload Characterization*. IEEE. 2008, pp. 119–128.

[92] Steve R Kleiman et al. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX." In: *USENIX Summer*. Vol. 86. 1986, pp. 238–247.

[93] Ricardo Koller and Raju Rangaswami. "I/O deduplication: Utilizing content similarity to improve I/O performance". In: *ACM Transactions on Storage (TOS)* 6.3 (2010), pp. 1–26.

[94] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. "High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System". In: *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. Santa Clara, CA: USENIX Association, 2017, pp. 197–212. ISBN: 978-1-931971-36-2. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/kumar.

[95] BM Kurkoski. "Notes on a lattice-based WOM construction that guarantees two writes". In: *Proc. 34th Symp. Information Theory and Applications*. 2011, pp. 520–524.

[96] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. "UNO: Uniflying host and smart NIC offload for flexible packet processing". In: *Proceedings of the 2017 Symposium on Cloud Computing*. 2017, pp. 506–519.

[97] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. "F2FS: A New File System for Flash Storage". In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*. Santa Clara, CA: USENIX Association, 2015, pp. 273–286. ISBN: 978-1-931971-201. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[98] Jae-Duk Lee, Chi-Kyung Lee, Myung-Won Lee, Han-Soo Kim, Kyu-Charn Park, and Won-Seong Lee. "A new programming disturbance phenomenon in NAND flash memory by source/drain hot-electrons generated by GIDL current". In: *Non-Volatile Semiconductor Memory Workshop, 2006. IEEE NVSMW 2006. 21st*. IEEE. 2006, pp. 31–33.

[99] Huaicheng Li. *Evolving Storage Stack for Predictability and Efficiency*. PhD. Thesis, University of Chicago. University of Chicago, 2020.

[100] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. "LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 591–605. ISBN: 9781450371025. DOI: 10.1145/3373376.3378531. URL: https://doi.org/10.1145/3373376.3378531.

[101] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. "The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator". In: *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*. Oakland, CA: USENIX Associa-

tion, 2018, pp. 83–90. ISBN: 978-1-931971-42-3. URL: https://www.usenix.org/conference/fast18/presentation/li.

[102] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. "An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production". In: (2020).

[103] Peng Li, Wei Dang, Congmin Lyu, Min Xie, Quanyang Bao, Xiaofeng Ji, and Jianhua Zhou. "Reliability Characterization and Failure Prediction of 3D TLC SSDs in Large-Scale Storage Systems". In: *IEEE Transactions on Device and Materials Reliability* 21.2 (2021), pp. 224–235. DOI: 10.1109/TDMR.2021.3063164.

[104] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. "Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 395–410. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/liang.

[105] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. "Optimizing NAND flash-based SSDs via retention relaxation". In: *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*. San Jose, CA: USENIX Association, 2012, p. 11. URL: https://www.usenix.org/conference/fast12/optimizing-nand-flash-based-ssds-retention-relaxation.

[106] Jan Lucas, Mauricio Alvarez-Mesa, Michael Andersch, and Ben Juurlink. "Sparkk: Quality-scalable approximate storage in DRAM". In: *The memory forum*. 2014, pp. 1–9.

[107] 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: *24th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 504–517.

[108] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. "Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation". In: *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '18. Irvine, CA, USA: ACM, 2018, pp. 106–106. ISBN: 978-1-4503-5846-0. DOI: 10.1145/3219617.3219659. URL: http://doi.acm.org/10.1145/3219617.3219659.

[109] Yixin Luo, Saugata Ghose, Yu Cai, Erich F. Haratsch, and Onur Mutlu. "Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation". In: *Proceedings of the 2018 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '18)* 2.3 (Dec. 2018), 37:1–37:48. ISSN: 2476-1249. DOI: 10.1145/3224432. URL: http://doi.acm.org/10.1145/3224432.

[110] Fabio Margaglia and André Brinkmann. “Improving MLC flash performance and endurance with extended P/E cycles”. In: *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2015, pp. 1–12.

[111] Fabio Margaglia, Gala Yadgar, Eitan Yaakobi, Yue Li, Assaf Schuster, and Andre Brinkmann. “The Devil Is in the Details: Implementing Flash Page Reuse with WOM Codes”. In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 95–109. ISBN: 978-1-931971-28-7. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/margaglia.

[112] Ashlie Martinez and Vijay Chidambaram. “CrashMonkey: A Framework to Automatically Test File-System Crash Consistency”. In: *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, 2017.

[113] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. “The New ext4 Filesystem: Current Status and Future Plans”. In: *Proceedings of the Linux symposium*. Vol. 2. 2007, pp. 21–33.

[114] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. “A Large-Scale Study of Flash Memory Failures in the Field”. In: *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '15)*. Portland, OR, 2015, pp. 177–190. ISBN: 978-1-4503-3486-0. DOI: 10.1145/2745844.2745848. URL: http://doi.acm.org/10.1145/2745844.2745848.

[115] Hongyu Miao, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. “StreamBox-HBM: Stream Analytics on High Bandwidth Hybrid Memory”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, 167–181. ISBN: 9781450362405. DOI: 10.1145/3297858.3304031. URL: https://doi.org/10.1145/3297858.3304031.

[116] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. “StreamBox: Modern Stream Processing on a Multicore Machine”. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 617–629. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/miao.

[117] Neal Mielke, Hanmant P Belgal, Albert Fazio, Qingru Meng, and Nick Righos. “Recovery Effects in the Distributed Cycling of Flash Memories”. In: *Proceedings of the 44th Annual International Reliability Physics Symposium*. IEEE. 2006, pp. 29–35.

[118] Neal Mielke, Todd Marquart, Ning Wu, Jeff Kessenich, Hanmant Belgal, Eric Schares, Falgun Trivedi, Evan Goodness, and Leland R Nevill. "Bit error rate in NAND flash memories". In: *Proceedings of the 46th Annual International Reliability Physics Symposium*. IEEE. 2008, pp. 9–19.

[119] Sparsh Mittal and Jeffrey S Vetter. "A survey of CPU-GPU heterogeneous computing techniques". In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–35.

[120] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. "Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association. Carlsbad, CA, 2018.

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

[122] Kevin P Murphy. "Elements of information theory". In: *In: Advances in Knowledge Discovery and Data Mining, Fayyad, U*. Citeseer. 1998.

[123] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. "Write off-loading: Practical power management for enterprise storage". In: *ACM Transactions on Storage (TOS)* 4.3 (2008), pp. 1–23.

[124] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. "SSD Failures in Datacenters: What? When? And Why?" In: *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*. Haifa, Israel, 2016, 7:1–7:11. ISBN: 978-1-4503-4381-7. DOI: 10.1145/2928275.2928278. URL: http://doi.acm.org/10.1145/2928275.2928278.

[125] Jacob Nelson, Adrian Sampson, and Luis Ceze. "Dense approximate storage in phase-change memory". In: *ASPLOS Ideas & Perspectives* (2011).

[126] *Nescafe: A near Storage Computational Framework. Private Repo to be published with paper*. URL: https://github.com/shehbazj/femu_iscos.

[127] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. "Understanding PCIe Performance for End Host Networking". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 327–341. ISBN: 9781450355674. DOI: 10.1145/3230543.3230560. URL: https://doi.org/10.1145/3230543.3230560.

[128] *NGD Systems*. https://www.ngdsystems.com/technology/computational-storage. URL: https://www.ngdsystems.com/technology/computational-storage.

[129] *NoLoad U.2 Computational Storage Platform*. https://www.eideticom.com/uploads/images/NoLoad_U2_Computational_Storage_Product_Brief.pdf. URL: https://www.eideticom.com/uploads/images/NoLoad_U2_Computational_Storage_Product_Brief.pdf.

[130] V NVIDIA. *TESLA K20 GPU accelerator board specification*. 2013.

[131] *NVMe 1.4 Specification*. https://nvmexpress.org/developers/nvme-specification/. URL: https://nvmexpress.org/developers/nvme-specification/.

[132] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (1996), pp. 351–385.

[133] *Open-Source Solid-State Drive Project for Research and Education*. http://openssd.io. Accessed: 2020-03-24.

[134] "OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 649–656. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/jung.

[135] Yuqian Pan, Haichun Zhang, Mingyang Gong, and Zhenglin Liu. "Process-variation Effects on 3D TLC Flash Reliability: Characterization and Mitigation Scheme". In: ().

[136] 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: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 47–62. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/panda.

[137] Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Thomas Mittelholzer, Evangelos Eleftheriou, Charles Camp, Thomas Griffin, Gary Tressler, and Andrew Walls. "Using Adaptive Read Voltage Thresholds to Enhance the Reliability of MLC NAND Flash Memory Systems". In: *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI (GLSVLSI '14)*. Houston, TX, 2014, pp. 151–156. ISBN: 978-1-4503-2816-6. DOI: 10.1145/2591513.2591594. URL: http://doi.acm.org/10.1145/2591513.2591594.

[138] Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens. *Parallel lossless data compression on the GPU*. IEEE, 2012.

[139] Gennady Pekhimenko, Chuanxiong Guo, Myeongjae Jeon, Peng Huang, and Lidong Zhou. "TerseCades: Efficient Data Compression in Stream Processing". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 307–320. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/pekhimenko.

[140] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. "Capuchin: Tensor-Based GPU Memory Management for Deep Learning". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, 891–905. ISBN: 9781450371025. DOI: 10.1145/3373376.3378505. URL: https://doi.org/10.1145/3373376.3378505.

[141] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "Crash consistency". In: *Communications of the ACM* 58.10 (2015), pp. 46–51.

[142] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "Analysis and Evolution of Journaling File Systems." In: *USENIX Annual Technical Conference, General Track*. Vol. 194. 2005, pp. 196–215.

[143] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "IRON File Systems". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*. Brighton, United Kingdom, 2005, pp. 206–220. ISBN: 1-59593-079-5. DOI: 10.1145/1095810.1095830.

[144] *QLC NAND – What can we expect from the technology?* https://www.architecting.it/blog/qlc-nand/. 2022.

[145] *QLC Support in FEMU*. URL: https://github.com/ucare-uchicago/FEMU/pull/47/commits/429ed12f65185ab4b114a4bb13c91afee05e9e95.

[146] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. "No "Power" Struggles: Coordinated Multi-Level Power Management for the Data Center". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIII. Seattle, WA, USA: Association for Computing Machinery, 2008, 48–59. ISBN: 9781595939586. DOI: 10.1145/1346281.1346289. URL: https://doi.org/10.1145/1346281.1346289.

[147] K Venkata Ramanaiah and Cyril Prasanna Raj. "Asic implementation of neural network based image compression". In: *International Journal of Computer Theory and Engineering* 3.4 (2011), p. 494.

[148] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. "Ensemble-Level Power Management for Dense Blade Servers". In: *Proceedings of the 33rd Annual International Symposium on Computer Architecture*. ISCA '06. USA: IEEE Computer Society, 2006, 66–77. ISBN: 076952608X. DOI: 10.1109/ISCA.2006.20. URL: https://doi.org/10.1109/ISCA.2006.20.

[149] Ashish Ranjan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. "Approximate storage for energy efficient spintronic memories". In: *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6. DOI: 10.1145/2744769.2744799.

[150] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Can Applications Recover from fsync Failures?" In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 753–767. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/rebello.

[151] Veronica Lagrange Moutinho dos Reis, Harry (Huan) Li, and Anahita Shayesteh. "Modeling Analytics for Computational Storage". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. ICPE '20. Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 88–99. ISBN: 9781450369916. DOI: 10.1145/3358960.3375794. URL: https://doi.org/10.1145/3358960.3375794.

[152] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. "Active Disks for Large-Scale Data Processing". In: *Computer* 34.6 (June 2001), pp. 68–74. ISSN: 0018-9162. DOI: 10.1109/2.928624. URL: https://doi.org/10.1109/2.928624.

[153] Ronald L Rivest and Adi Shamir. "How to reuse a "write-once memory". In: *Information and control* 55.1-3 (1982), pp. 1–19.

[154] Ohad Rodeh, Josef Bacik, and Chris Mason. "BTRFS: The Linux B-Tree Filesystem". In: *ACM Transactions on Storage (TOS)* 9.3 (Aug. 2013), pp. 1–32. ISSN: 1553-3077. DOI: 10.1145/2501620.2501623.

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

[156] M. Rosenblum. "The Design and Implementation of a Log-structured File System". PhD thesis. University of California at Berkeley, 1992.

[157] Mendel Rosenblum and John K Ousterhout. "The design and implementation of a log-structured file system". In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 26–52.

[158] Zhenyuan Ruan, Tong He, and Jason Cong. "INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 379–394. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/ruan.

[159] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. "Approximate Storage in Solid-State Memories". In: *ACM Trans. Comput. Syst.* 32.3 (2014). ISSN: 0734-2071. DOI: 10.1145/2644808. URL: https://doi.org/10.1145/2644808.

[160] *Samsung Smart SSDs*. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html. URL: https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[161] Marco AA Sanvido, Frank R Chu, Anand Kulkarni, and Robert Selinger. "NAND flash memory and its role in storage architectures". In: *Proceedings of the IEEE* 96.11 (2008), pp. 1864–1874.

[162] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. "Lethe: A tunable delete-aware LSM engine". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 893–908.

[163] *ScaleFlux Computational Storage*. https://www.scaleflux.com/document/. URL: https://www.scaleflux.com/document/.

[164] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. "Flash Reliability in Production: The Expected and the Unexpected". In: *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*. Santa Clara, CA: USENIX Association, 2016, pp. 67–80.

[165] *SCSI Error Handling (EH)*. https://www.kernel.org/doc/Documentation/scsi/scsi_eh.txt. [Online; accessed 23-Oct-2019]. 2019.

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

[167] Amir Shpilka. "Capacity-achieving multiwrite WOM codes". In: *IEEE Transactions on Information Theory* 60.3 (2013), pp. 1481–1487.

[168] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. "Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 403–415. ISBN: 9781450341974. DOI: 10.1145/3035918.3035954. URL: https://doi.org/10.1145/3035918.3035954.

[169] *SLC vs MLC vs TLC vs QLC*. https://memkor.com/slc-vs-mlc-vs-tlc%2Fqlc. 2022.

[170] *sLSM*. A Skiplist based LSM Tree https://github.com/aronszanto/sLSM-Tree. URL: https://github.com/aronszanto/sLSM-Tree.

[171] SNIA IOTTA Trace Repository. *YCSB RocksDB SSD Traces*. http://iotta.snia.org/traces/28568. Sept. 2020.

[172] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, et al. "A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme". In: *IEEE Journal of Solid-State Circuits* 30.11 (1995), pp. 1149–1156.

[173] Amy Tai, Andrew Kryczka, Shobhit O Kanaujia, Kyle Jamieson, Michael J Freedman, and Asaf Cidon. "Who's afraid of uncorrectable bit errors? online recovery of flash errors with distributed redundancy". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 977–992.

[174] Vasily Tarasov, Erez Zadok, and Spencer Shepler. "Filebench: A flexible framework for file system benchmarking". In: *USENIX; login* 41.1 (2016), pp. 6–12.

[175] Thomas N Theis and H-S Philip Wong. "The end of moore's law: A new beginning for information technology". In: *Computing in Science & Engineering* 19.2 (2017), pp. 41–50.

[176] *TLC vs QLC SSDs: The Ultimate Guide*. https://storagereviews.net/tlc-vs-qlc-ssds/. 2022.

[177] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. "Understanding the Impact of Power Loss on Flash Memory". In: *Proceedings of the 48th Design Automation Conference (DAC '11)*. San Diego, CA, 2011, pp. 35–40. ISBN: 978-1-4503-0636-2. DOI: 10.1145/2024724.2024733.

[178] Yongkun Wang, Kazuo Goda, Miyuki Nakano, and Masaru Kitsuregawa. "Early experience and evaluation of file systems on SSD with database applications". In: *5th International Conference on Networking, Architecture, and Storage (NAS)*. IEEE. 2010, pp. 467–476.

[179] Debao Wei, Libao Deng, Peng Zhang, Liyan Qiao, and Xiyuan Peng. "NRC: A nibble remapping coding strategy for NAND flash reliability extension". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.11 (2016), pp. 1942–1946.

[180] *Western Digital and Toshiba talk up penta-level cell flash*. https://blocksandfiles.com/2019/08/07/penta-level-cell-flash/. Accessed: 2020-03-24.

[181] *WOM-v coder: LightNVM extension. Private Repo to be published with paper*. URL: https://github.com/shehbazj/linux_wom.

[182] *Xylinx Boards and Kits*. https://www.xilinx.com/products/boards-and-kits.html. Accessed: 2020-03-24.

[183] Eitan Yaakobi, Laura Grupp, Paul H Siegel, Steven Swanson, and Jack K Wolf. "Characterization and error-correcting codes for TLC flash memories". In: *2012 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, pp. 486–491.

[184] Eitan Yaakobi, Jing Ma, Laura Grupp, Paul H Siegel, Steven Swanson, and Jack K Wolf. "Error characterization and coding schemes for flash memories". In: *2010 IEEE Globecom Workshops*. IEEE. 2010, pp. 1856–1860.

[185] Eitan Yaakobi and Amir Shpilka. "High sum-rate three-write and nonbinary WOM codes". In: *IEEE Transactions on Information Theory* 60.11 (2014), pp. 7006–7015.

[186] Eitan Yaakobi, Gala Yadgar, Nachum Bundak, and Lior Gilon. "A case for biased programming in flash". In: *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. 2018.

[187] Eitan Yaakobi, Alexander Yucovich, Gal Maor, and Gala Yadgar. "When Do WOM Codes Improve the Erasure Factor in Flash Memories?" In: *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*. 2015.

[188] Gala Yadgar, MOSHE Gabel, Shehbaz Jaffer, and Bianca Schroeder. "SSD-Based Workload Characteristics and Their Performance Implications". In: *ACM Trans. Storage* 17.1 (2021). ISSN: 1553-3077. DOI: 10.1145/3423137. URL: https://doi.org/10.1145/3423137.

[189] Gala Yadgar and Roman Shor. "Experience from Two Years of Visualizing Flash with SSDPlayer". In: *ACM Trans. Storage* 13.4 (Nov. 2017). ISSN: 1553-3077. DOI: 10.1145/3149356. URL: https://doi.org/10.1145/3149356.

[190] Gala Yadgar, Eitan Yaakobi, Fabio Margaglia, Yue Li, Alexander Yucovich, Nachum Bundak, Lior Gilon, Nir Yakovi, Assaf Schuster, and André Brinkmann. "An analysis of flash page reuse with WOM codes". In: *ACM Transactions on Storage (TOS)* 14.1 (2018), pp. 1–39.

[191] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. "Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes". In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 2015, pp. 257–271.

[192] Gala Yadgar, Alexander Yucovich, Hila Arobas, Eitan Yaakobi, Yue Li, Fabio Margaglia, André Brinkmann, and Assaf Schuster. *Limitations on MLC flash page reuse and its effects on durability*. Tech. rep. Computer Science Department, Technion, 2016.

[193] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs". In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 15–28. ISBN: 978-1-931971-36-2. URL: https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan.

[194] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. "An empirical guide to the behavior and use of scalable persistent memory". In: *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 2020, pp. 169–182.

[195] Jinfeng Yang, Bingzhe Li, and David J. Lilja. "Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 5.1 (Feb. 2020). ISSN: 2376-3639. DOI: 10.1145/3372783. URL: https://doi.org/10.1145/3372783.

[196] Runyu Zhang, Duo Liu, Xianzhang Chen, Xiongxiong She, Chaoshu Yang, Yujuan Tan, Zhaoyan Shen, and Zili Shao. "LOFFS: A Low-Overhead File System for Large Flash Memory on Embedded Devices". In: *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*. DAC '20. Virtual Event, USA: IEEE Press, 2020. ISBN: 9781450367257.

[197] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. "FPGA-Accelerated Compactions for LSM-based Key-Value Store". In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 225–237. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/zhang-teng.

[198] Tong Zhang. *What is a computational storage drive? Much needed help for CPUs*. https://www.infoworld.com/article/3615497/what-is-a-computational-storage-drive-much-needed-help-for-cpus.html. Info World, 2021. URL: "https://www.infoworld.com/article/3615497/what-is-a-computational-storage-drive-much-needed-help-for-cpus.html".

[199] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Xiaodong Zhang, Nanning Zheng, and Tong Zhang. "LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives". In: *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX, 2013, pp. 243–256. ISBN: 978-1-931971-99-7. URL: https://www.usenix.org/conference/fast13/technical-sessions/presentation/zhao.

[200] Yutong Zhao, Wei Tong, Jingning Liu, Dan Feng, and Hongwei Qin. "CeSR: A Cell State Remapping Strategy to Reduce Raw Bit Error Rate of MLC NAND Flash". In: *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2019, pp. 161–171.

[201] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. "Understanding the Robustness of SSDs Under Power Fault". In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*. San Jose, CA: USENIX Association, 2013, pp. 271–284.

[202] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. "Reliability Analysis of SSDs Under Power Fault". In: *ACM Transactions on Storage (TOS)* 34.4 (Nov. 2016), pp. 1–28. ISSN: 0734-2071. DOI: 10.1145/2992782.

[203] Ning Zheng, Xubin Chen, Jiangpeng Li, Qi Wu, Yang Liu, Yong Peng, Fei Sun, Hao Zhong, and Tong Zhang. "Re-think Data Management Software Design Upon the Arrival of Storage Hardware with Built-in Transparent Compression". In: *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/hotstorage20/presentation/zheng.

[204] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. "XRP: In-Kernel Storage Functions with eBPF". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 375–393. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/zhong.

[205] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. "BPF for Storage: An Exokernel-Inspired Approach". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, 128–135. ISBN: 9781450384384. DOI: 10.1145/3458336.3465290. URL: https://doi.org/10.1145/3458336.3465290.