ABSTRACT

Title of Dissertation: TACKLING PERFORMANCE AND SECURITY ISSUES FOR CLOUD STORAGE SYSTEMS

Luyi Kang
Doctor of Philosophy, 2022

Dissertation Directed by: Professor Bruce Jacob
Department of Electrical and Computer Engineering

Building data-intensive applications and emerging computing paradigm (e.g., Machine Learning (ML), Artificial Intelligence (AI), Internet of Things (IoT) in cloud computing environments is becoming a norm, given the many advantages in scalability, reliability, security and performance. However, under rapid changes in applications, system middleware and underlying storage device, service providers are facing new challenges to deliver performance and security isolation in the context of shared resources among multiple tenants. The gap between the decades-old storage abstraction and modern storage device keeps widening, calling for software/hardware co-designs to approach more effective performance and security protocols. This dissertation rethinks the storage subsystem from device-level to system-level and proposes new designs at different levels to tackle performance and security issues for cloud storage systems.

In the first part, we present an event-based SSD (Solid State Drive) simulator that

models modern protocols, firmware and storage backend in detail. The proposed simulator can capture the nuances of SSD internal states under various I/O workloads, which help researchers understand the impact of various SSD designs and workload characteristics on end-to-end performance.

In the second part, we study the security challenges of shared in-storage computing infrastructures. Many cloud providers offer isolation at multiple levels to secure data and instance, however, security measures in emerging in-storage computing infrastructures are not studied. We first investigate the attacks that could be conducted by offloaded in-storage programs in a multi-tenancy cloud environment. To defend against these attacks, we build a lightweight Trusted Execution Environment, IceClave to enable security isolation between in-storage programs and internal flash management functions. We show that while enforcing security isolation in the SSD controller with minimal hardware cost, IceClave still keeps the performance benefit of in-storage computing by delivering up to 2.4x better performance than the conventional host-based trusted computing approach.

In the third part, we investigate the performance interference problem caused by other tenants' I/O flows. We demonstrate that I/O resource sharing can often lead to performance degradation and instability. The block device abstraction fails to expose SSD parallelism and pass application requirements. To this end, we propose a software/hardware co-design to enforce performance isolation by bridging the semantic gap. Our design can significantly improve QoS (Quality of Service) by reducing throughput penalties and tail latency spikes.

Lastly, we explore more effective I/O control to address contention in the storage software stack. We illustrate that the state-of-the-art resource control mechanism, Linux

cgroups is insufficient for controlling I/O resources. Inappropriate cgroup configurations may even hurt the performance of co-located workloads under memory intensive scenarios. We add kernel support for limiting page cache usage per cgroup and achieving I/O proportionality.

TACKLING PERFORMANCE AND SECURITY ISSUES FOR CLOUD STORAGE SYSTEMS

by

Luyi Kang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2022

Advisory Committee:
Professor Bruce Jacob, Chair/Advisor
Professor Donald Yeung
Professor Manoj Franklin
Professor Gang Qu
Professor Laixiang Sun, Deans of Representative

© Copyright by
Luyi Kang
2022

Acknowledgments

I owe my gratitude to all the people who have made this dissertation possible.

First and foremost I'd like to express my greatest gratitude to my advisor, Professor Bruce Jacob for his continuous guidance, support and encouragement over the past years. His experience and insight has given me an invaluable opportunity to work on challenging and interesting projects. It has been a pleasure to learn from and work with an extraordinary professor with unique wisdom and vision.

I would also like to thank the committee for their time and advice for my defense and dissertation, especially Prof. Yeung, who led me into the world of Computer Architecture and spent many weekly meetings with me. Special thanks to Prof. Sun who serves as the dean's representative.

I would also like to acknowledge help and support from my friends and group members at the University of Maryland. I will remember many of the discussions with Shang Li, Devesh Singh and Candace Walden in A.V. Williams 1418 office. I'd like to express my best wishes for their future careers.

I owe my deepest thanks to my family who have always been loving me and supporting me. They share my pressure and anxiety with the greatest patience and love. I owe everything I achieved to them.

It is impossible to remember all, and I apologize to those I've inadvertently left out.

Table of Contents

Preface ii

Foreword ii

Dedication ii

Acknowledgements ii

Table of Contents iii

List of Tables vii

List of Figures viii

List of Abbreviations xi

Chapter 1: Introduction 1
- 1.1 Overview . . . 2
  - 1.1.1 Characterize Internal Mechanisms of SSDs and Implications . . . 2
  - 1.1.2 IceClave: A Trusted Execution Environment for In-Storage Computing 3
  - 1.1.3 Zoned FTL: Achieve Performance Isolation on Shared SSDs . . . 4
  - 1.1.4 Approaching More Effective I/O Control . . . 5
- 1.2 Organization of the Dissertation . . . 6

Chapter 2: Characterize Internal Mechanisms of SSDs 7
- 2.1 Fundamentals of NVMe SSDs . . . 7
  - 2.1.1 Flash Technology Background . . . 7
  - 2.1.2 Flash Interface and Commands . . . 9
  - 2.1.3 NVMe SSD and Storage Subsystem . . . 12
  - 2.1.4 Internal Architecture . . . 14
  - 2.1.5 Flash Translation Layer . . . 16
  - 2.1.6 Address Mapping . . . 18
  - 2.1.7 Wear Leveling . . . 22
  - 2.1.8 Garbage Collection and Over Provisioning . . . 24
  - 2.1.9 NVMe Protocol . . . 27
  - 2.1.10 Internal Caching . . . 31
- 2.2 Motivations . . . 32

2.3 Performance Factors . . . 33
2.3.1 Flash Layout and Internal Parallelism . . . 33
2.3.2 Page Allocation Scheme . . . 34
2.3.3 Host Interface . . . 35
2.3.4 I/O Transaction Scheduling . . . 38
2.3.5 FTL Algorithms . . . 39
2.3.6 DRAM Data Caching . . . 41
2.4 Simulator Overview . . . 42
2.4.1 Flash Complex . . . 42
2.4.2 FTL Modules . . . 44
2.4.3 Host Interface . . . 44
2.4.4 I/O Flow within the Simulator . . . 45
2.4.5 Key Features . . . 46
2.5 Evaluation . . . 47
2.5.1 IOPs and Latency Under Varying I/O Queue Depth . . . 48
2.5.2 IOPs with Varying Number of Flash Channels . . . 50
2.5.3 Impact of Page Size . . . 52
2.5.4 Impact of Read and Write Interference . . . 55
2.5.5 Impact of Data Buffer . . . 57
2.5.6 Impact of Caching Mapping Table . . . 59
2.5.7 Impact of Data Cache Contention . . . 61
2.5.8 Impact of Page Allocation Scheme . . . 62
2.5.9 Impact of Multi-Plane Operations . . . 63
2.5.10 Impact of Copy-back Operations . . . 65
2.6 Summary . . . 66

Chapter 3: IceClave: A Trusted Execution Environment for In-Storage Computing 67
3.1 Background and Motivations . . . 67
3.1.1 In-storage Computing . . . 67
3.1.2 In-storage Vulnerabilities . . . 68
3.1.3 Existing Security Framework . . . 71
3.1.4 Secure Memory Design . . . 74
3.2 Design Goal . . . 79
3.3 Threat Model . . . 81
3.4 Design and Implementation . . . 82
3.4.1 Challenges of Building IceClave . . . 83
3.4.2 Protecting Flash Translation Layer . . . 85
3.4.3 Access Control for In-Storage Programs . . . 88
3.4.4 Securing In-Storage DRAM . . . 89
3.4.5 IceClave Runtime . . . 94
3.4.6 Put It All Together . . . 95
3.4.7 Discussion and Future Work . . . 98
3.5 Implementation Details . . . 99
3.6 Evaluation . . . 102
3.6.1 Experimental Setup . . . 103

3.6.2 Performance of IceClave . . . 105
3.6.3 Overhead Source in IceClave . . . 107
3.6.4 Impact on Power Consumption . . . 109
3.6.5 Impact of SSD Bandwidth . . . 110
3.6.6 Impact of Data Access Latency . . . 112
3.6.7 Impact of Computing Capability . . . 113
3.6.8 Impact of DRAM Capacity in SSD . . . 114
3.6.9 Performance of Multi-tenant IceClave . . . 114
3.7 Conclusion . . . 115

Chapter 4: Zoned FTL: Achieve Resource Isolation via Hardware Virtualization 117
4.1 Background and Motivations . . . 121
4.1.1 Multi-queue NVMe SSD and NVMe Semantics . . . 121
4.1.2 Unpredictable Performance of SSD . . . 125
4.1.3 I/O Virtualization . . . 128
4.1.4 Evaluation of Performance Interference . . . 132
4.1.5 Access Latency Breakdown . . . 134
4.2 Existing Approaches . . . 136
4.3 Design . . . 137
4.3.1 Design Goals . . . 139
4.3.2 Overview . . . 140
4.3.3 Interface . . . 141
4.3.4 Resource Isolation . . . 142
4.3.5 Block Allocator . . . 145
4.3.6 Split Cache . . . 147
4.3.7 Put It All Together . . . 148
4.4 Evaluation . . . 152
4.4.1 Raw Performance . . . 154
4.4.2 Evaluation with Synthetic Benchmark . . . 155
4.4.3 Evaluation with Realistic Background Interference . . . 158
4.4.4 Impact of Split Cache . . . 161
4.4.5 Sensitivity to GC Intensity . . . 162
4.5 Summary . . . 163

Chapter 5: Approaching More Effective I/O Control 164
5.1 Deep Dive into I/O Stack . . . 165
5.1.1 Linux Block I/O Stack . . . 165
5.1.2 Software is a Thick Layer . . . 169
5.1.3 Page Cache . . . 171
5.1.4 Block I/O Scheduler . . . 177
5.1.5 Linux Containers . . . 183
5.1.6 Control Groups: I/O and Memory Controller Explained . . . 196
5.2 Performance Pitfalls . . . 210
5.3 Design and Implementation . . . 214
5.4 Evaluation . . . 217

5.5 Discussions 219
5.6 Summary 221

Chapter 6: Related Work 222
6.1 Emerging Storage Interface 222
6.2 Improve Performance Isolation 224
6.3 Bypass Software Stack and NON-POSIX Interface 225
6.4 Simulation Platforms 226
6.5 In-storage Computing and Storage Encryption 227

Chapter 7: Future Work 231

Chapter 8: Conclusions 234

Bibliography 242

List of Tables

2.1 Levels of parallelism in SSDs 15
2.2 Configuration of SSD simulation. 48
2.3 Average bandwidth and queuing time of random reads (RR), RR + sequential writes (SW), RR + channel-fixed SW, RR + SW with write suspension, bandwidth (MB/s), Queue waiting time (us) 59

3.1 In-storage workload characterization. 90
3.2 IceClave API 94
3.3 In-storage computing simulator. 100
3.4 In-storage workloads used in our evaluation. 103
3.5 Overhead Source of IceClave. 106
3.6 Extra memory traffic caused by memory encryption and verification when running in-storage workloads. 107

4.1 Experimental configuration. 132
4.2 I/O characteristics of workloads 154
4.3 Parameters of emulated SSD 155

5.1 Average latency breakdown of a random read syscall 170
5.2 Experimental setup. 212
5.3 Read/Write latency (us) of modified kernel compared against base. 218

List of Figures

2.1 Difference between SLC, MLC, and TLC memories and their life cycles. . 9
2.2 Data path comparison between off-chip migration and intra-plane copy-back. 11
2.3 Multi-plane program and die interleave command data path. 12
2.4 NVMe storage subsystem and block I/O. 13
2.5 Internal architecture of flash-based SSDs. 14
2.6 Diagram of page-level mapping table and page update process. 20
2.7 Diagram of block-level mapping table and page update process. 21
2.8 Different types of merge operation. 22
2.9 Typical lifecycle of a flash block. 25
2.10 Host interacting with an NVMe device. 29
2.11 Round-Robin (RR) NVMe queue arbitration. 30
2.12 Weighted Round-Robin (WRR) NVMe queue arbitration. 30
2.13 Page address encoding. 35
2.14 A static page allocation scheme: channel-way-die-plane. 36
2.15 Multi-queue SSD interface and mqblk I/O layer. 37
2.16 Comparison between in-order scheduling and out-of-order scheduling. . . 39
2.17 Overview of the SSD simulator. 43
2.18 Write (W) IOPs comparison between multiple simulation configs and a real SSD with various queue depth. 50
2.19 Heavy-write (HW) IOPs comparison between multiple simulation configs and a real SSD with various queue depth. 51
2.20 Read-write (RW) IOPs comparison between multiple simulation configs and a real SSD with various queue depth. 52
2.21 Heavy-read (HR) IOPs comparison between multiple simulation configs and a real SSD with various queue depth. 53
2.22 Read (R) IOPs comparison between multiple simulation configs and a real SSD with various queue depth. 54
2.23 IOPs trend with varying number of channels. 58
2.24 Normalized average response time with varying page size. 59
2.25 Normalized throughput with varying page size. 60
2.26 Average access latency per request of real-world workloads, less is better. 61
2.27 Normalized IOPs of real-world workloads, higher is better. 62
2.28 Average access latency per request of real-world workloads, less is better. 63

2.29 Bandwidth degradation factor (Y-axis) scaling with queue depth (X-axis) of concurrent Random Writes. 64
2.30 Bandwidth degradation factor (Y-axis) scaling with write cache size (X-axis, MB). 64
2.31 Normalized IOPS of various static page allocation schemes. 65
2.32 Normalized average latency (Y-axis) for various workloads (X-axis, MB) under three plane policies. 65
2.33 Impact on Write Amplification Factor by enabling copy-back operation. . 66

3.1 IceClave enables in-storage trusted execution environments to achieve security isolation between in-storage programs, FTL, and flash chips. . . . 68
3.2 Process of encryption and decryption in counter mode encryption. 76
3.3 Process of encryption and decryption in counter mode encryption. 77
3.4 Diagram of Split Counter. 77
3.5 Using Bonsai Merkle Tree as integrity tree. 78
3.6 Threat model of in-storage computing. 83
3.7 Overview of IceClave architecture. 84
3.8 Memory protection regions in IceClave. 86
3.9 Performance comparison between IceClave and IceClave with FTL mapping table in secure world. Performance is normalized to IceClave. 86
3.10 In-storage memory protection in IceClave. 88
3.11 Memory encryption and integrity trees in IceClave. 91
3.12 Performance comparison of Non-encryption, Split Counters (SC-64), and IceClave. It is normalized to the scheme without memory encryption. . . 92
3.13 The workflow of running in-storage programs with IceClave. 96
3.14 Stream cipher engine design in IceClave. 101
3.15 Performance comparison of Host, Host+SGX, ISC, and IceClave (from left to right). We show the performance breakdown of each scheme. . . . 105
3.16 Energy overhead of IceClave. 109
3.17 IceClave performance (normalized to Host), as we vary the internal SSD bandwidth by using different number of channels. 110
3.18 IceClave performance (normalzied to ISC), as we vary the internal SSD bandwidth by using different number of channels. 110
3.19 IceClave performance as we vary the latency of accessing flash pages. . . 112
3.20 IceClave performance as we vary the in-storage computing capability. . . 113
3.21 IceClave performance as we vary the SSD DRAM size. 114
3.22 IceClave performance as we run two in-storage applications concurrently. 115
3.23 IceClave performance as we run four in-storage applications concurrently. 116

4.1 The Multiqueue I/O architecture. 122
4.2 An overview of NVMe SSD virtualization. 123
4.3 NVMe Command Format. 124
4.4 Breakdown of feature accountability for high latency. 128
4.5 The latency (us) distribution of the latency-critical workload. 134

4.6 The breakdown of average access latency for 4K random read running with RR (random read only), RR + RW (with background random write), and RR + Burst (with multiple write bursts). 135
4.7 Overview of ZFTL architecture. 139
4.8 Virtualized applications interact with ZFTL interface and command flow. . 143
4.9 Processing a read request across ZFTL stack. 149
4.10 Processing a write request across ZFTL stack. 151
4.11 FIO random read latency with varying block sizes. 156
4.12 FIO random read throughput with varying block sizes. 156
4.13 The latency (unit:us) distribution of running alone, baseline, and ZFTL. . 157
4.14 Throughput of Baseline, ZFTL scheme normalized to solely running TPC-C on Baseline system. 158
4.15 99th percentile latency of Baseline, ZFTL scheme normalized to solely running TPC-C on Baseline system. 159
4.16 Normalized 99th percentile latency of Baseline, ZFTL scheme under heavy GC. 160
4.17 Normalized average and 99th percentile latency of Baseline, ZFTL-UC and ZFTL scheme. 161
4.18 The average response time under various GC watermark. 162

5.1 Overview of a Linux I/O stack. 168
5.2 Host interacting with an NVMe device. 169
5.3 Request path in a regular storage software stack. 171
5.4 Diagram of NOOP scheduler. 179
5.5 Diagram of BFQ scheduler. 180
5.6 Diagram of Deadline scheduler. 182
5.7 Simplified diagram of Kyber scheduler. 183
5.8 IOPS of FIO 4KB random reads and writes on different I/O schedulers. . . 184
5.9 IOPS of FIO 1MB sequential reads and writes on different I/O schedulers. 185
5.10 IOPS of PostgreSQL and RocksDB on different I/O schedulers. 186
5.11 Comparison between virtual machine and container hierarchy. 187
5.12 Layered File System of Container. 195
5.13 Process of a Buffered Read Request. 215
5.14 IOPS among three 4KB buffered write workloads, using BFQ. 219
5.15 IOPS among three 4KB buffered write workloads, using page cache limit. 220

List of Abbreviations

DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
FTL	Flash Translation Layer
GC	Garbage Collection
IOPS	I/O Operations Per Second
LPA	Logical Page Address
LUN	Logical Unit
ML	Machine Learning
NVMe	Non-Volatile Memory Express
ONFI	Open NAND Flash Interface
OS	Operating System
PCIe	PCI Express
PLM	Predictable Latency Mode
PPA	Physical Page Address
QoS	Quality of Service
SGX	Software Guard Extension
SLA	System Level Agreement
SR-IOV	Single Root I/O Virtualization
SSD	Solid State Drive
WAF	Write Amplification Factor
WL	Wear Leveling
VM	virtual Machine
ZNS	Zoned Namespace

Chapter 1: Introduction

In these days, Cloud computing environment is the new paradigm to allow flexible, elastic and robust computing resources requested by the user. Users can borrow and use the IT resources and pay. Storage subsystems, the cornerstone of data centers, have evolved tremendously over the past decade. Cloud service providers are actively deploying emerging storage devices (e.g. NVMe SSDs, Optane drives) to serve data-intensive applications. Meanwhile, they are allowing multiple tenants to share the hardware resources via virtualization techniques, as a deployment strategy to maximize utilization and operation revenues. Users can borrow and use the hardware resources to enable their services according to the performance, security and reliability contract - the System Level Agreements (SLA).

However, achieving I/O SLA in a complex virtualization environment is challenging due to resource sharing. Resource sharing can often conflict with performance metrics and security enforcement. For example, latency-critical I/O can be slowed down by more than 10x if the SSD is running heavy background operations. A throughput-oriented workload may not enjoy its fair share of the system bandwidth due to resource contention with other tenants at multiple levels. Most of the SLA violations result from the widening gap between complicated storage devices and the conventional block interface. Modern SSDs behave like a full-fledged system, however, the decades-old block interface hides their

internal activities from applications or hypervisor. Without context from upper layers, or more fine-grained control over SSDs, it is almost impossible for service providers to manage I/O flows to satisfy SLA.

The prevalent resource sharing in storage subsystems presents a significant challenge: How should we design storage systems to provide performance and security guarantees? Specifically, which levels of the system should be re-designed to bridge the gap between SLA and underlying storage? To answer this question, we take a close look at the entire storage stack from device-level to software-level, analyze potential crux and develop solutions at different levels. We discuss these parts in the next sections.

1.1 Overview

1.1.1 Characterize Internal Mechanisms of SSDs and Implications

Cloud providers are replacing hard drives with SSDs to keep up with the increasing storage demand for throughput and latency. A modern SSD device, unlike its predecessor, is structured as a complicated system that consists of an embedded processor, internal DRAM and multiple flash chips organized in a vertical hierarchy. The internal firmware is responsible for managing flash resources, serving I/O requests, and communicating with the host system. However, storage vendors intentionally hide all the firmware details, making it difficult for system architects to build a detailed performance model, let alone figure out the performance bottleneck for workloads. The lack of a detailed SSD model also presents challenges for researchers to understand: (1) the key mechanisms that contribute to the high performance of modern NVMe SSDs; (2) how to build applications,

firmware and OS for better and robust performance. As modern SSDs and their protocols evolve to meet the changing demands of data centers, the system community needs an SSD simulator that reliably models key features. Unfortunately, existing SSD simulators either lack accurate modeling of major performance factors, or fails to scale with more storage resources, resulting in significant deviation compared to commodity products.

In this section, we present an event-based SSD simulator that models the flash internals including modern host interface, firmware and storage backend in detail. We describe our implementation choices in achieving accuracy and scalability. By modeling key features of modern NVMe SSDs, we observe several performance implications for SSD designers and users. We discuss several guidelines to better exploit the potential performance and the future design of SSD and I/O stack.

1.1.2 IceClave: A Trusted Execution Environment for In-Storage Computing

In-storage computing has been a promising technique for accelerating data-intensive applications, especially for large-scale data processing and analytics. It moves computation closer to the data stored in the storage devices like flash-based SSDs, such that it can overcome the I/O bottleneck by significantly reducing the amount of data transferred between the host machine and storage devices. As modern SSDs are employing multiple general-purpose embedded processors and large DRAM in their controllers, it becomes feasible to enable in-storage computing in reality today. It has been proven to be an effective approach to alleviate the I/O bottleneck. To facilitate in-storage computing, many frameworks have been proposed. However, few of them treat the in-storage security as the first citizen.

Specifically, since modern SSD controllers do not have a trusted execution environment, an offloaded (malicious) program could steal, modify, and even destroy the data stored in the SSD, which hinders the wide adoption of in-storage computing. In this chapter, To defend against these attacks, we build a lightweight trusted execution environment, named IceClave for in-storage computing. IceClave enables security isolation between in-storage programs and flash management functions that include flash address translation, data access control, and garbage collection, with TrustZone extensions. IceClave also achieves security isolation between in-storage programs by enforcing memory integrity verification of in-storage DRAM with low overhead. To protect data loaded from flash chips, IceClave develops a lightweight data encryption/decryption mechanism in flash controllers. We develop IceClave with a full system simulator. We evaluate IceClave with a variety of data-intensive applications such as databases. Compared to state-of-the-art in-storage computing approaches, IceClave introduces only 7.6% performance overhead, while enforcing security isolation in the SSD controller with minimal hardware cost. IceClave still keeps the performance benefit of in-storage computing by delivering up to 2.31× better performance than the conventional host-based trusted computing approach.

1.1.3 Zoned FTL: Achieve Performance Isolation on Shared SSDs

Modern NVMe SSDs are widely deployed in data centers to provide orders of magnitude higher throughput and lower response time. Many workloads such as web search, databases, and interactive programs demand a low and stable tail latency for responses to users' requests. Unfortunately, it is challenging to guarantee tail latency as resource sharing

in cloud environments inherently causes 10x-100x longer tail latency due to contention. Resource contention includes but not limited to: (1) The necessary internal management activities performed by the SSD firmware such as garbage collection can block user I/Os, leading to a delay in an order of milliseconds; (2) shared storage resource such as flash chips, data bus and internal buffer tend to incur contentions on the critical I/O path, blocked I/Os suffer significant delays. These tail-latency events slow down overall system performance and amounts to unreliable service. This is because applications are unaware of data layout in the underlying device and the device is unaware of the SLA of I/O requests. The conventional storage system stack lacks full isolation between applications sharing the storage device. To this end, we propose a novel FTL design at device level and system-level isolation that offers performance isolation among multiple virtualized I/O services, by leveraging existing NVMe semantics and hardware virtualization. Our evaluations show that ZFTL can improve throughput by 1.51x and reduce tail latency by up to 4.9x while preserving similar parallelism.

1.1.4 Approaching More Effective I/O Control

Containers are gaining more popularity for virtualization capacity in modern datacenters. One of the major benefits is consolidation of multiple services onto a single physical machine, leading to better resource utilization. Therefore, ensuring resource isolation for container consolidation is a fundamental requirement in such environments. The state-of-the-art virtualization techniques such as Docker rely on Linux cgroup to manage host resources for containers. While our approaches in Chapter 4 could provide performance

isolation in the underlying storage, they couldn't help the I/O control and isolation needed in the kernel I/O stack. In principle, the kernel I/O stack is supposed to provide proportional I/O resources to containers based on weight. However, cgroup I/O management only functions at the block I/O layer, which leaves many I/O requests that are serviced by upper layers out of control. In this chapter, we illustrate that the state-of-the-art resource control mechanism, cgroup is insufficient for I/O resource manipulation. We reveal that inappropriate setups of cgroup may even hurt the performance of co-located workloads under I/O or memory intensive scenarios. To address the problem, we add direct page cache control to the cgroup memcontroller module; we modify the page reclamation scheme to support page allocation and eviction based on priority and weight.

1.2 Organization of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 studies the performance characteristics of SSD internal mechanisms. Chapter 3 proposes a trusted execution environment for secure in-storage computing. Chapter 4 presents a novel FTL design that enables performance isolation among applications. Chapter 5 approaches more effective I/O control by adding kernel support. Chapter 6 and Chapter 7 discuss related work and future work, respectively. Chapter 8 concludes the dissertation.

Chapter 2: Characterize Internal Mechanisms of SSDs

In this section, we present an event-based SSD simulator that models modern features and conventional functionalities in detail. First, we introduce the background of flash-based SSDs. Then, We describe our design choices in achieving better accuracy by taking account of major performance factors. Lastly, we conclude several performance implications for SSD architects and users by evaluating various I/O workloads on different device-level design choices.

2.1 Fundamentals of NVMe SSDs

2.1.1 Flash Technology Background

Flash memories are called solid-state devices because they are composites of transistor gates without any moving mechanical parts. A regular NAND cell can be read or programmed within hundreds of microseconds [4], much faster than the mechanical disk. Flash cells can handle random accesses as fast as sequential ones as there is no need to move the sensing head across the physical disk as mechanical disks do. In terms of capacity, a single NAND die can scale up to several gigabytes and a single package can contain up to 16 dies as the manufacturers have been working on scaling with multi-layered or stacked

NAND for the past decade. This massive density improvement are pushed by two key technologies:

- **3D integration** NAND memory cells can be vertically stacked to form multiple memory layers within the same die. Until 2021, NAND manufacturers have pushed the number of layers towards 200 [199] and more is expected to come in the near future [196].
- **Multilevel storage** Flash storage is built upon the ability of trapping electrons inside a metal-oxide-semiconductor (MOS) transistor. The number of trapped electrons decides the transistor's threshold voltage. By manipulating the numbers of trapped electrons, multiple threshold voltages can be generated and translated into the digital domain. For example, eight voltage values will result in 3b of digital information. Based on the number of voltage levels, NAND memories can be classified as [135]: SLC (two threshold voltages, 1b per memory cell), MLC (four threshold voltages, 2b per memory cell), TLC (eight threshold voltages, 3b per memory cell), TLC (eight threshold voltages, 3b per memory cell), QLC (16 threshold voltages, 4b per memory cell). Figure 2.1 presents the difference between SLC, MLC, and TLC NAND memories.

These cell-level innovations help scale the density but not decrease the access latency. This is because the access latency is tied to physical constants such as the capacitance of NAND cells [166]. Instead, as a memory cell represents more states, it is getting less reliable and exhibit longer latencies because extra operations is needed (error correction) to finish an I/O request [170]. Another major challenging facing new flash technology is

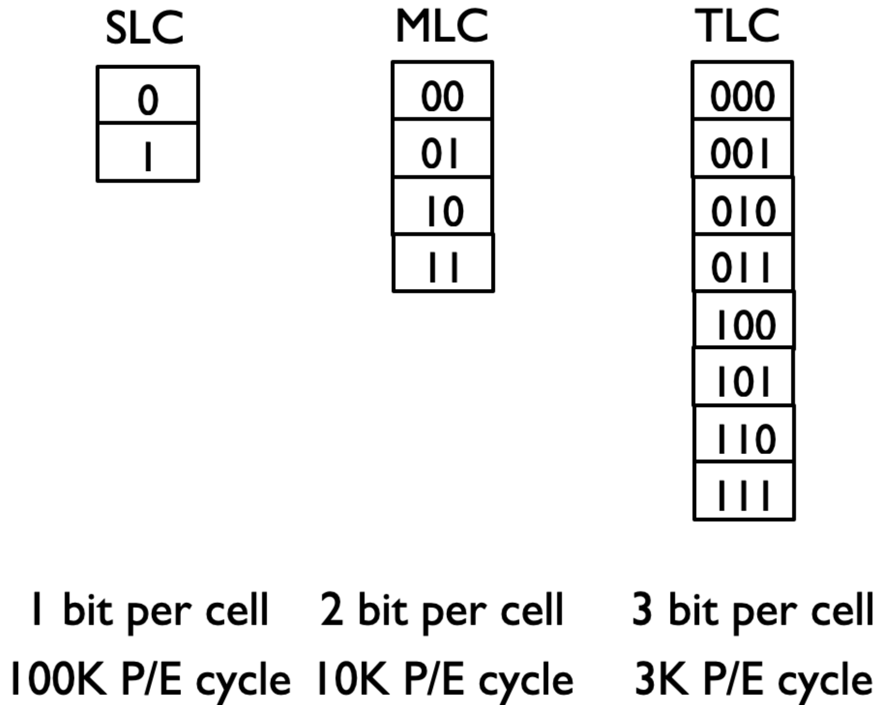


Figure 2.1: Difference between SLC, MLC, and TLC memories and their life cycles.

the limited erase cycles before a block is worn out. The endurance of a typical TLC block is 3,000 erasure times [175]. After wearing out, a block can't be used for storing any data. Thus, the SSD internal architecture has undergone significant changes to adapt to the cell technology shift.

2.1.2 Flash Interface and Commands

All NAND Flash devices use a multiplexed address and data bus. Both 2D NAND and 3D NAND dies are shipped with a standard command interface called Open NAND Flash Interface (ONFI) [151]. The flash controller sends ONFI commands to the flash chip to perform flash operations such as read, write and erase and the flash chip sends data or

operation status via the bus to the controller. The read/program operation is performed at a page-granularity, while the erase operation is performed at a block-granularity by resetting the data to value ”1” in the entire block. Besides the three basic operations in flash chips, read, program (write) and erase for data manipulation, many flash manufacturers provide advanced flash commands, such as multi-plane, copy-back, and multi-die interleave commands [88, 113, 189] to further improve the SSD performance, shown in Figure 2.3. However, aggressively adopting advanced commands does not always bring performance benefit due to their strict working restrictions.

- **Copy-back** moves data from one page to another in the same plane without occupying the interior or exterior I/O bus. The source page and the target page must locate in the same plane, and the page offset must be both odd or both even (this restriction has been relaxed in recent products). Figure 2.2 compares the data path between normal off-chip page migration and intra-plane copy-back.

- **Multi-plane** allows multiple read, program or erase operations in all planes in the same die simultaneously. The cost equals to a single operation. The pages executing multi-plane operations must have the same chip, die, block and page address. Since the pages must be programmed consecutively in the increasing order of page offset within a block, two multi-plane target planes must be aligned. The multi-plane commands provides another level of internal parallelism.

- **Multi-die Interleave** Flash commands execute in different dies of the same chip simultaneously. This process is similar to the idea of data pipeline on the I/O bus, to hide the long flash access latency, as illustrated in Figure 2.3. Once the data is

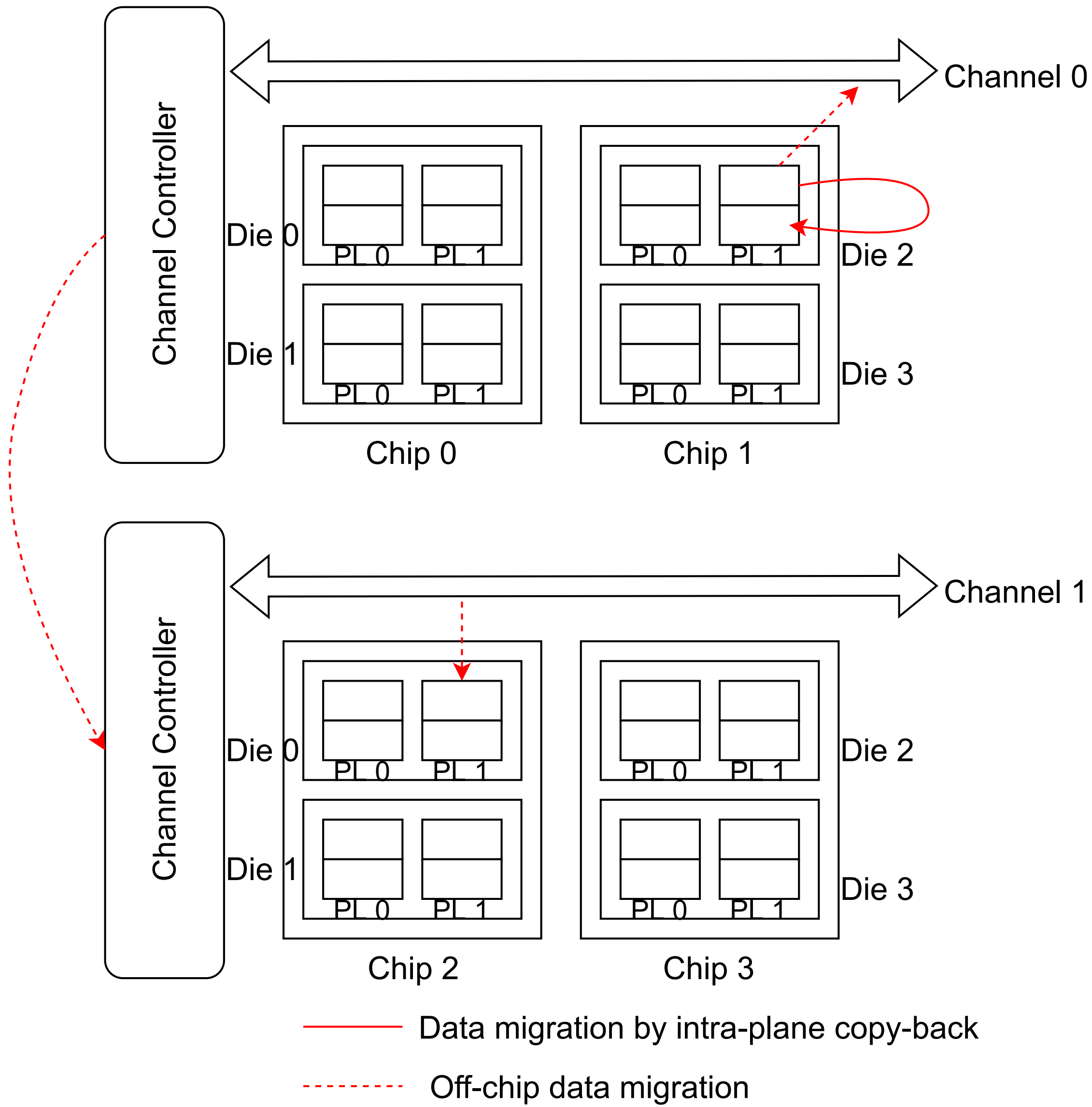


Figure 2.2: Data path comparison between off-chip migration and intra-plane copy-back.

transferred, die 1 can immediately start programming the page.

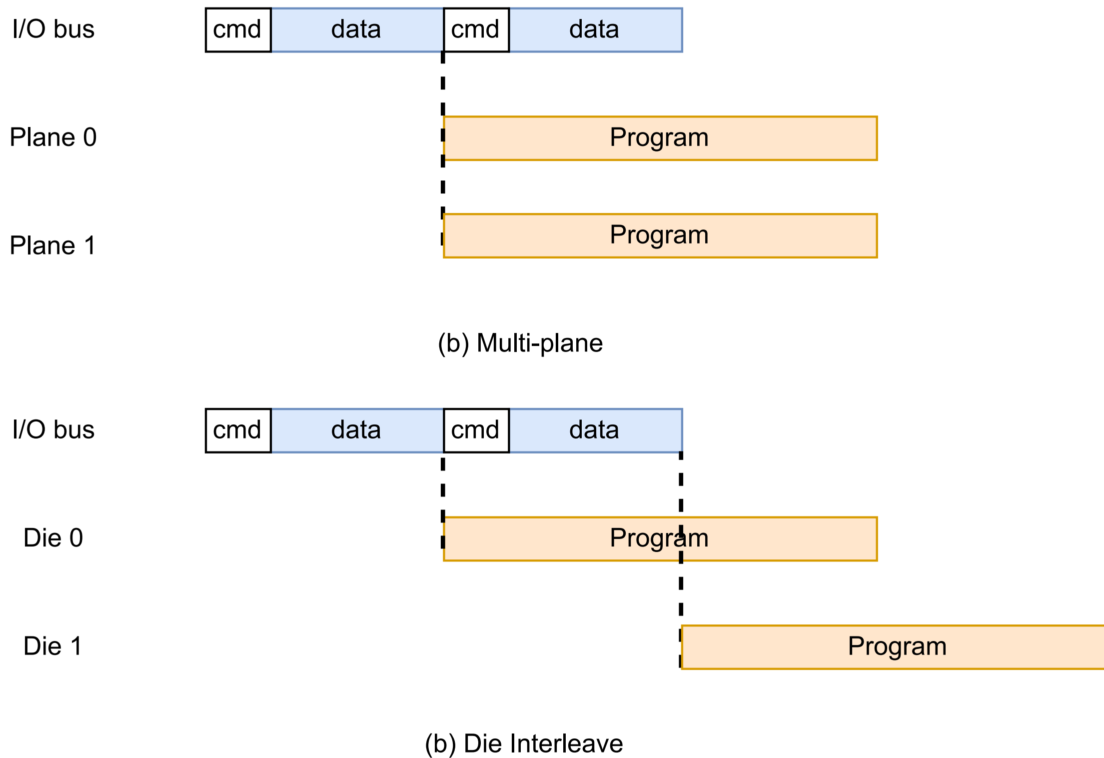


Figure 2.3: Multi-plane program and die interleave command data path.

2.1.3 NVMe SSD and Storage Subsystem

NVMe SSDs have brought earth-shaking changes to the storage industry in recent years. They use the latest PCIe [7] interface and the significantly simplified NVMe protocol [6] to communicate with the host system, which provides stunning throughput and response time with multiple direct queues between the host and the SSD. As of now, NVMe SSDs can provide up to several million I/O operations per second (IOPS) and read latencies as low as 70µs [2, 4]. However, they are not treated differently from traditional disks in a system perspective. They both appear as standard block devices in

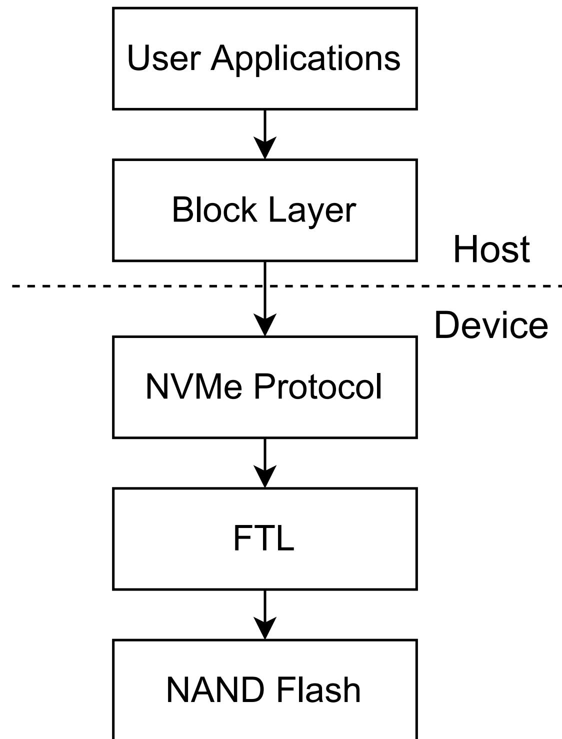


Figure 2.4: NVMe storage subsystem and block I/O.

the Linux environment. As shown in Figure 2.4, it gives upper level systems the same abstraction that NVMe SSDs can be accessed in the same manner as conventional disks, even their performance characteristics are fundamentally different. Nowadays, about 80% of client/consumer and 50% of enterprise SSDs are NVMe based, and these numbers are growing quickly. For instance, Facebook datacenters are building ultra-scale, high-performance and low-latency storage using NVMe SSDs to service billions of daily active users, and trillions of transactions daily [214].

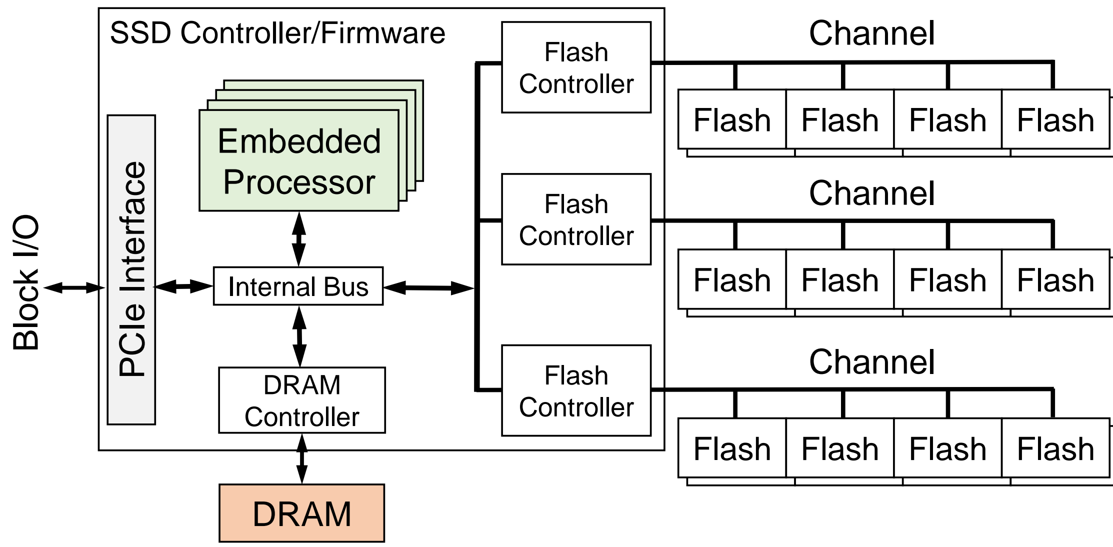


Figure 2.5: Internal architecture of flash-based SSDs.

2.1.4 Internal Architecture

We present the hardware architecture of a typical SSD in Figure 2.5. An SSD has three major components: a set of flash memory packages, an SSD controller having embedded processors with DRAM, and a host interface [85, 240]. At the high level, the SSD controller is responsible for: (1) communication with the host. (2) communication with the backend flash arrays and DRAM. (3) I/O request scheduling and resource management to ensure sustainable performance. (4) Maintenance of flash blocks. (5) data integrity and retention. The internal DRAM is typically used to buffer writes and cache page mapping tables [154, 276]. The flash packages are organized in a hierarchical manner. An SSD has multiple independently-accessed channels, typically between 4 and 32. Each channel has its own signal bus and page register, which acts as a buffer between the flash chip

Table 2.1: Levels of parallelism in SSDs

Levels	**Description**
Channel	fully independent
Chip	serialized over a shared channel bus, limited concurrency
Die	interleaved across dies, data and command serialized over a shared package bus
Plane	can perform same operations in multiple planes at the same page offset due to the shared page register

and NAND flash controller. To maximize the bandwidth, an I/O request can be striped over multiple channels. Each channel contains multiple flash chips (packages), where they share the same channel for data and command transfer. Unlike channels, the access into different flash chips in the same channel are serialized. Each flash chip consists of multiple flash dies, where each die complies to ONFI standard. Each die may further consists multiple planes, plane sharing can activate the same operations at the same offset of all planes. Each plane consists of thousands of flash blocks. A block is an erase unit, and the size of a block determines the Garbage Collection (GC) granularity. Each block consists of thousands of pages. A page is the smallest write unit, which is usually 4 - 16 KB in size. Different levels of parallelism is concluded in Table 2.1.

To leverage the rich parallelism presented in Table 2.1, the internal SSD controller usually strips a data request across multiple channels and logical units (LUN). As such, the data request can be serviced in parallel, resulting in more aggregate bandwidth. Meanwhile, the rich internal resources can hide the penalty of the erase operation. For example, when

a flash chip is performing an erase operation, other flash chips in the same channel can still serve incoming data requests.

2.1.5 Flash Translation Layer

Due to the nature of NAND flash memory, when a free flash page is written once, that page is no longer available for future writes until it is erased. However, an erase operation can be performed only at a block granularity, which is expensive. A thick, sophisticated flash firmware called Flash Translation Layer (FTL) is designed to solve the in-place write problem [185] like a log-structured file system [213]. An FTL maintains a logical address to physical address mapping table to record appending records to log files. As such, writes can be directed to other blocks that have free pages rather than erase the whole block and wait for completion. The out-of-place write requires Garbage Collection (GC) to be performed to clean the obsolete data when the number of free pages drops below a threshold. The GC procedure selects a victim block that has a high mark of invalid pages. The valid pages of the victim block will be migrated into a new target block when the victim block is erased. Since these internal flash management operations takes place in the background, researchers often name these data movements as background I/O. Background I/O incurs additional flash operations and can be blocking to user I/O. How to reduce background I/O and eliminate its impact on user I/O performance is one of the key challenges for designing a high-performance SSD [267]. Most modern flash chips support an advanced flash command called same-plane copyback, which suggests

within each plane, such page migration can occur in the same plane without occupying the bus. This feature has raised many studies on alleviating data migration incurred by GC [114, 115, 226, 254, 255]. Flash cells have limited lifetime: each cell is rated for a few thousand of erase operations. To maintain the factory capacity over years and maximize device lifetime, it is essential for all flash blocks to age uniformly, i.e., wear leveling. Address translation plays a key role in implementing out-of-place updates. FTL maintains a mapping table that stores the translation between a logical page address (LPA) and a physical page address (PPA). The page table is stored in the non-volatile memory and cached in DRAM for faster lookup. The granularity of the mapping scheme can be pure page-level, block-level, or hybrid mapping. Many modern SSDs adopt the page-level mapping algorithm as it provides arguably the best performance and reliability. FTL also manages the so-called page allocation scheme to exploit internal parallelism. It determines how data pages are placed on the SSD, which in turn impacts the available bandwidth exposed to users. Past studies have extensively studied the page allocation strategies to maximize the throughput and resource utilization [132,163,242]. To summarize, FTL is mainly responsible for: (1) translation between logical address and physical address (2) GC, recycle dirty blocks and maintain a number of clean blocks. (3) WL, ensure erasure blocks as evenly as possible, to overcome the inherent shortcomings of the flash media.

2.1.6 Address Mapping

To perform out-of-place writes, the FTL must maintains a mapping table that translates virtual addresses from host (e.g. requests from file systems), to physical addresses on the SSD. There are three types of mapping schemes in general, page-mapping, block-mapping, and hybrid [116, 125, 148, 197, 260]. The page-mapping scheme maintains an L2P (logical-to-physical) mapping table for address translation, as shown in Figure 2.6. Every logical page needs a table entry to store the mapping info (Logical Page Number (LPN) to Physical Page Number (PPN)). Thus, the mapping table size grows proportionally as the number of pages, raising higher demand for on-board DRAM to store the mapping table. The block-mapping scheme maintains a mapping table between logical blocks and physical blocks, as shown in Figure 2.7. However, this approach loses the flexibility - the page offset in a logical block must be identical to the page offset in the corresponding physical block. If an LPN is written to repeatedly, that LPN cannot be written to any other page in this physical block even if there are free pages. All the existing data in the block must be copied to a new clean clock and the old block data is marked as dirty. This operation involves multiple costly write and erase operations. The major benefit of block-mapping scheme is, it drastically reduces the size of the table by n times compared to the page-mapping scheme (n is the number of pages within a block). Hybrid mapping combines the advantage of page-mapping and block-mapping schemes, which maintains a block-level mapping table and a log-page-mapping table. In a hybrid scheme, physical blocks are partitioned into two groups: data blocks and log blocks. Data blocks stores data and log blocks are used as a buffer for new writes. Data blocks basically organizes

and manages all flash blocks by the block-level mapping scheme. Log blocks stores the updated data by the page-level mapping scheme. Each log block is paired with a data block. When a write request arrives, it first writes to the log block and invalidates the data in the corresponding data block. When a log block is full and no clean pages exist, it needs to be merged with the corresponding data block. A merge operation results from reclaiming free space from log blocks that are currently full. There are three types of merge operations depending on different scenarios, as shown in Figure 2.8:

- A partial merge is performed if both log block and data block have some valid pages and the pages in the blocks are in sequential order. Valid pages in the data block is copied to the log block and the log block becomes the data block. The old data block will be recycled.

- A switch merge is performed if the log block contains all valid pages, the log block becomes a data block, and the old data block will be recycled.

- A full merge is performed when both log block and data block contain valid pages but the pages are not in sequential order. In this case, valid pages from both log and data block need to be copied into a new block, and both blocks will be recycled.

The cost of merge operations is significant. To reduce the cost of merge operations, A number of hybrid schemes have been proposed in the past decade, including BAST [107], SuperBlock [131], FAST [108], CAST [260]. Random writes in hybrid schemes induce costly merge (GC) which in turn affects the overall performance. Therefore, none of the hybrid schemes can achieve comparable performance of a page-mapping scheme. More recently, many more optimized page-level mapping FTL, such as DFTL [99]. DFTL uses

page-level mapping FTL to reduce GC overhead against block-level and hybrid mapping FTL, while storing the entire mapping table in the flash memory and caching only the recently used mapping entries. They persists the full mapping table in the flash memories while caching frequent entries in on-board DRAM. Experimental results show that page-level [99] perform noticeably better the hybrid FTL scheme FAST [108]. The main reason behind is they can completely get rid of costly merge operations.

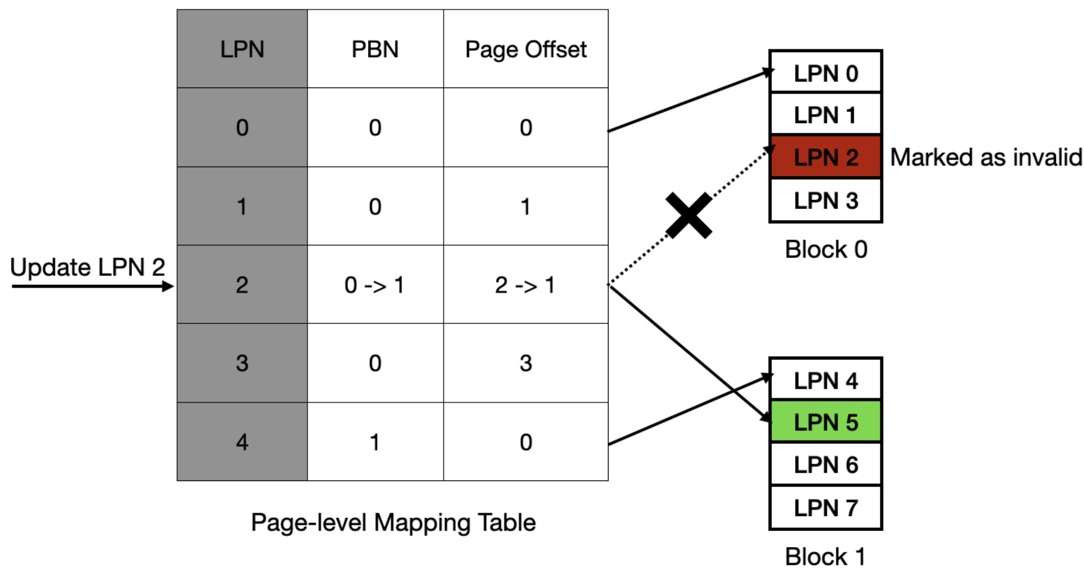


Figure 2.6: Diagram of page-level mapping table and page update process.

Modern SSDs generally adopts a page-level mapping for performance considerations. Assuming the page size is 4KB and each mapping entry takes 4B, the table size is abour 0.1% of the SSD capacity. As flash devices scale to terabytes, the needed volume to store the mapping table is way higher than the available on-board DRAM. Moreover, mapping table recovery after power failure takes more time that is proportional to the mapping

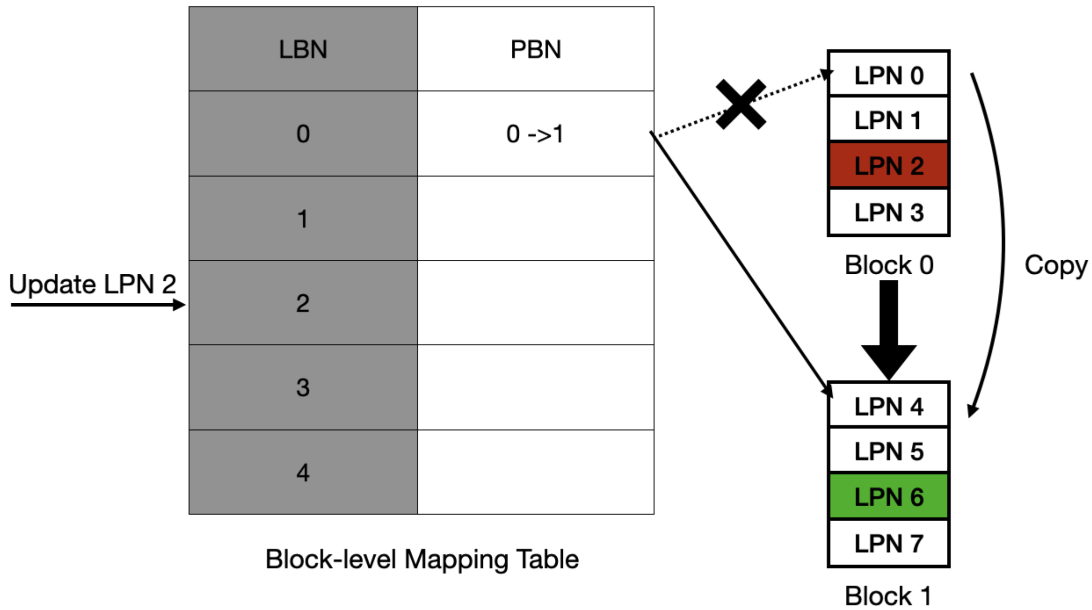


Figure 2.7: Diagram of block-level mapping table and page update process.

table size. Therefore, modern FTL persists the entire mapping table in the flash memories and stores frequently accessed mapping entries in an LRU cache in on-board DRAM for faster access. Updating mapping entries in flash memories also incurs additional I/O thereby hurt performance when the operation is on the critical path [109, 271]. Some manufacturers have even marketed DRAM-less SSDs that do not contain DRAM in the controller to further reduce the power consumption and cost [141, 282]. To alleviate the performance degradation due to removed on-board DRAM buffer, they leverage host memory buffer (HMB) feature of NVMe, which allows SSDs to use host DRAM. HMB is used in the same manner: (1) a read cache, (2) a write buffer, and (3) cache an address mapping table.

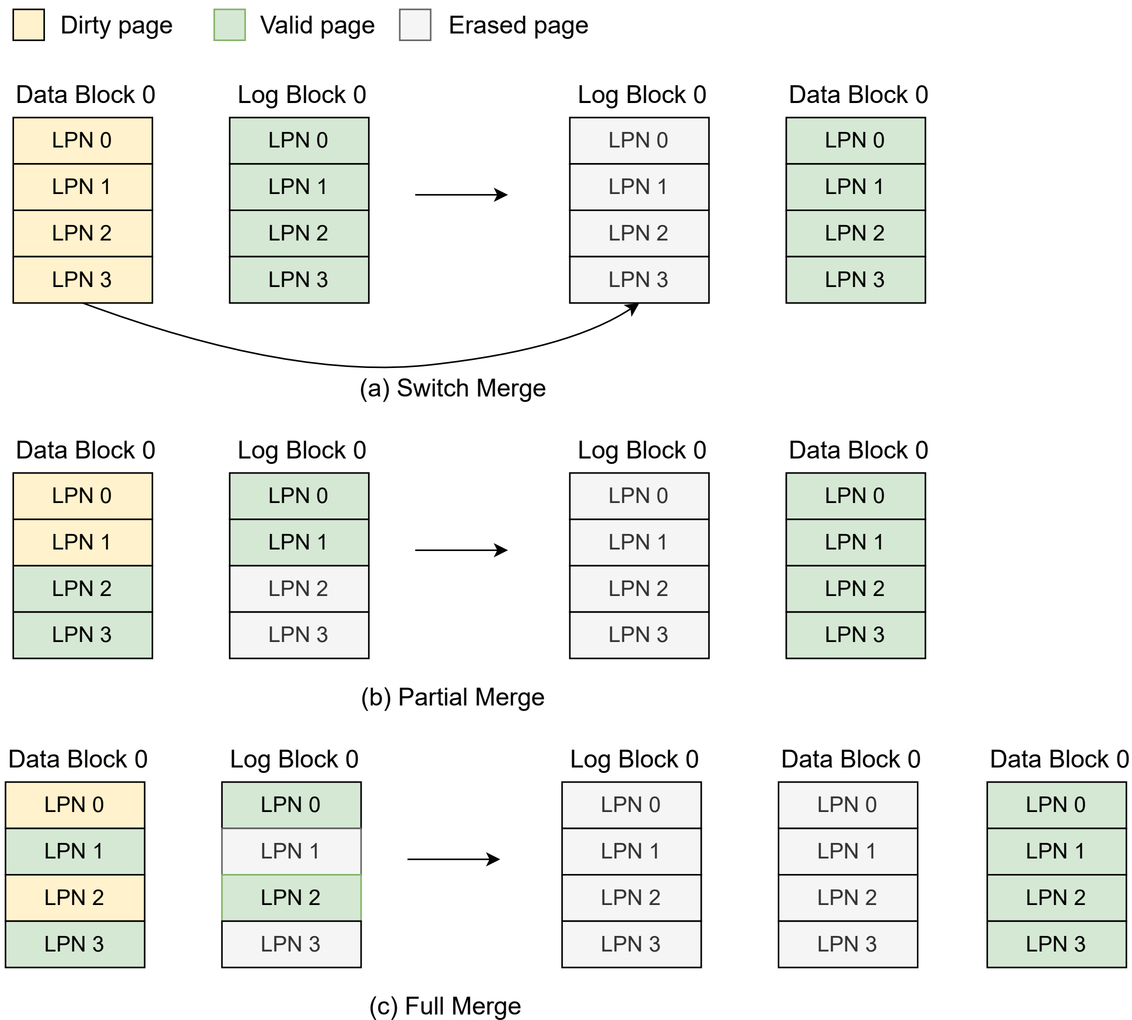


Figure 2.8: Different types of merge operation.

2.1.7 Wear Leveling

Generally, not all the data stored within the same flash die has the same lifetime, some data are more frequently updated while others remain more persistent. Therefore, data blocks containing data that are subject to updates frequently will experience more erase/write cycles than those containing data that is rarely updated. Frequent block erase

operations reduce the lifetime of flash memory. This is known as wear out problem. Due to the physical characteristics of NAND flash memory, the number of times that a block can be erased is limited. For example, an SLC flash can endure around 100K erasure while for a TLC flash the number is several thousand erasure times. The ideal situation is that all blocks age uniformly to maximize the lifetime of the device, which is the primary goal of any so-called wear leveling technique to balance erasure times across the device. The implementation of wear leveling algorithm heavily relies on the strategy of logical to physical mapping, that is, each time the host issues a request targeting a logical address, the flash firmware maps it to a different physical address on flash chips. By tracking the usage of blocks, the FTL can dynamically maps a 'hot' sector onto a different one to guarantee all the physical blocks are evenly used. A way to facilitate wear leveling is to split cold and hot data as much as possible into separate blocks [55], however, it is not easy to acknowledge the FTL such information on current architecture. There are two common wear leveling approaches in general. Dynamic wear leveling algorithm [266] attempts to avoid hot data written to the same block repeatedly so that no block reaches its maximum usage faster than other blocks. Static wear leveling algorithm [56] attempts to migrate cold data to more worn blocks so that the aging of all blocks converge to the average value.

2.1.8 Garbage Collection and Over Provisioning

To prevent worn-out blocks from shrinking the capacity, some spare flash resources (around 7% - 30%) are reserved inside the SSD [231]. These reserved resources are invisible to users from factory, called Over Provisioning (OP) resources. The purpose of OP resources are two-fold: (1) FTL uses them for bad block replacement, to guarantee tagged capacity during its lifetime, (2) FTL uses them to manage high write traffic. The intensive random write requests can be accommodated with these extra blocks temporarily. Once the available capacity of OP blocks drops below a preset threshold, the FTL starts a GC process to free more blocks. Generally, More OP resources has a positive impact on decreasing background traffic and unnecessary block erasure, which in turn improves user I/O performance and device lifetime. This is the main reason that enterprise-level SSDs that are equipped with excessive OP space have much better anti-disturbance capabilities and write endurance [172]. For example, Micron enterprise-level flagship SSD [4] reserves a configurable 30% of the total capacity as OP space. GC is crucial to guarantee the availability of free blocks. For a GC process, a victim block is selected, then every clean page in the block is copied into other clean blocks, eventually the victim blocks are erased. The typical lifecycle of a flash block is illustrated in Figure 2.9. Pages are written to clean, active blocks in a log-structured manner, pages that are not up-to-date are simply marked as invalid. The block remains in active state until it is selected for GC victim block. Before the block is selected as a victim block and erased, its valid pages are copied to other active blocks.

GC is a mechanism that manages out-of-the-place updates and flash idiosyncrasies

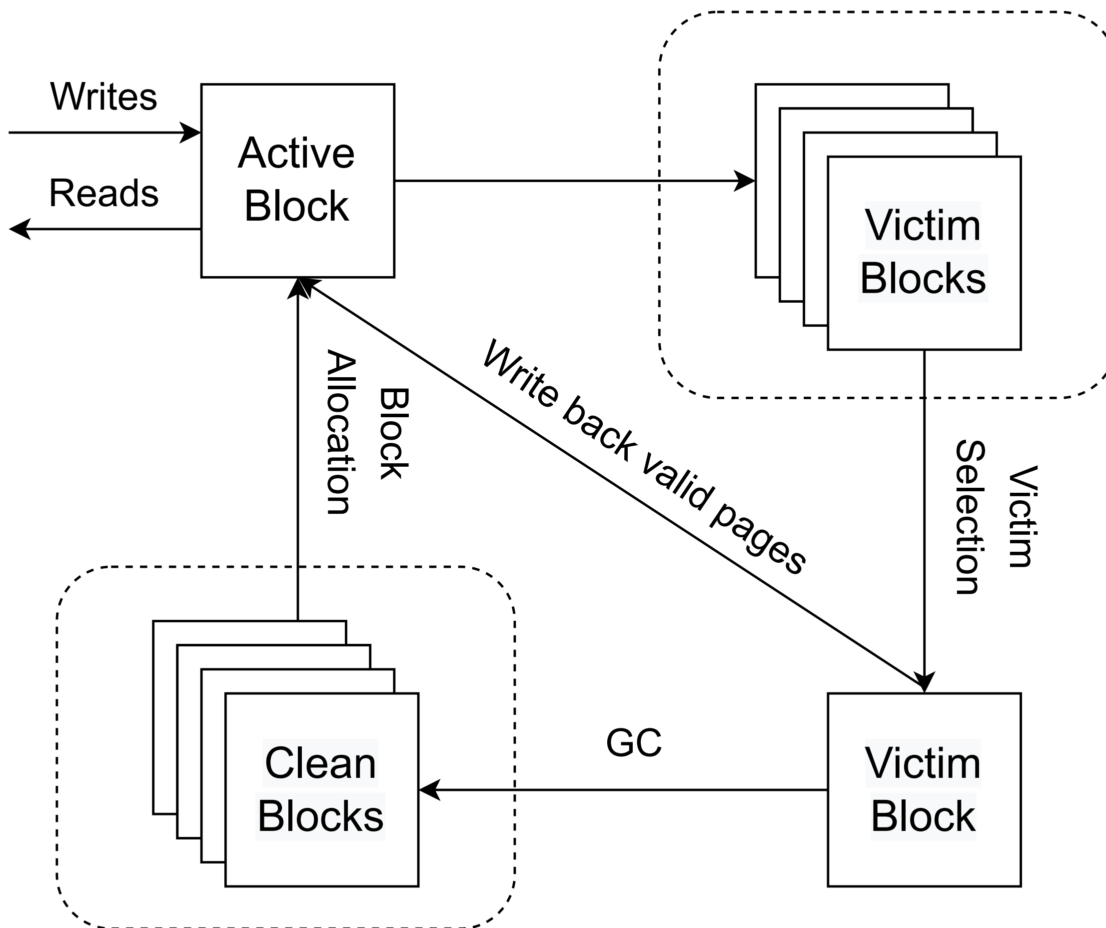


Figure 2.9: Typical lifecycle of a flash block.

but introduces additional data movement. Hence, an efficient GC algorithm should aim to minimize its footprint on performance and endurance. There are several optimization objectives for GC algorithms:

- Low cost for finding potential victim blocks. The time complexity of searching a victim block should be in constant time (or sub-linearly) as the number of flash blocks grows linearly with SSD capacity. Some heuristic victim selection algorithms such as Greedy, least recently written (LRU) policies provide cheap and stable block

selections.

- Minimal data movement. GC introduces additional page writes than actually needed, including mapping table updates and GC incurred data movement. The ratio of total writes committed to flash between total data received from the host is called Write Amplification Factor (WAF) [140]. It is computed as follows:

$$WAF = \frac{total_size_of_data_committed_to_flash}{total_size_of_data_received_from_host} \tag{2.1}$$

  A lower WAF means less data written due to GC. Therefore, the number of clean pages in victim blocks should be kept small to minimize WAF.

- Non-blocking. Once the number of clean blocks in an SSD drops below a pre-defined threshold, GC is invoked in high priority and blocks normal service of user I/Os. An efficient GC algorithm performs GC more intensively when the SSD resource is lightly utilized.

The core elements of a GC algorithm are when to do GC and which blocks to recycle. Many GC victim selection algorithms have been proposed in the past. The Greedy algorithm [50] selects a block with the least number of valid pages. it is a heuristic approach to minimize the cost of data movement. The victim selection can be implemented in constant time maintaining an array of lists where each list stores the number of valid pages in a block. Statistically, greedy selection algorithm performs well for uniform accesses. However, it does not perform very well in realistic scenarios where page access distributions are non-uniform – some blocks that are accessed more

frequently are called hot blocks, while those less frequently accessed are called cold blocks, the non-uniform access pattern disturbs the performance of statistically optimal GC policies. To this end, other victim selection policies are proposed to handle real-world distributions: RGA (Random Greedy Algorithm) [269] algorithm chooses a random set of blocks and selects one with the fewest number of valid pages within them; Cost-benefit (CB) algorithm [123] takes the number of valid pages and time since last page invalidation as parameters to ensure cold blocks are selected sooner. GC algorithms closely coordinate with FTL to decide when to perform GC, in order to minimize its impact on user I/O response time. Preemptive GC [153] hides the GC overhead by allowing an I/O request to preempt an ongoing GC process. [19] redirects writes to GC-free flash dies. TTFlash [263] keeps multiple data copies on different flash dies and serves read requests from a GC-free die.

2.1.9 NVMe Protocol

NVM Express (NVMe) [6] is an inherently parallel and scalable interface designed for today's fast storage devices. NVMe protocol allows fast I/O accesses and millions of I/O operations per second (IOPS) by enabling multiple queues between the host and SSD. Each CPU core is allowed to interact with SSD with an independent, low-overhead hardware queue. Figure 5.2 illustrates the process of host talking to device via NVMe protocol. The device controller exposes specific part of the internal DRAM to the host via PCIe, which is referred as base line address (BAR). Once the device controller completes an I/O request, it directly writes the interrupt into the memory-mapped region (in host

DRAM), called the MSI/MSI-X vector. The host and device controller can manage I/O requests submissions and completions via NVMe queue pairs, BARs and MSI vector. Each queue pair consists of a Submission Queue (SQ) and a Completion Queue (CQ). The host system maintains an Admin SQ and its associated Admin CQ, and up to 64K I/O SQs or CQs. The depth of I/O queues is 64K. A SQ or CQ is a ring buffer and it can be accessed by the device via Direct Memory Access (DMA). A doorbell is a register of the NVMe device controller to record the head or tail pointer of SQ or CQ. I/O requests are serviced by NVMe protocol in the following steps:

- The device driver encapsulates the I/O request and issues the NVMe command (consists of starting logical address, I/O length, physical address of I/O buffers and other metadata), to an NVMe submission queue. The command is inserted at the tail of the queue.
- The doorbell register of the SQ is written to notify new I/O requests for the device.
- Upon I/O completion, the NVMe controller constructs an NVMe completion entry and places at the tail of the corresponding CQ.
- When the corresponding interrupt handler is scheduled, the host fetches an entry of the CQ head and calls a completion function, which eventually wakes up the blocked thread.

The NVMe protocol provides hardware-assisted arbitration mechanisms for I/O command scheduling [127]. The default round-robin scheduling policy processes I/O commands from all submission queues in a round-robin manner. If the Weighted-Round-

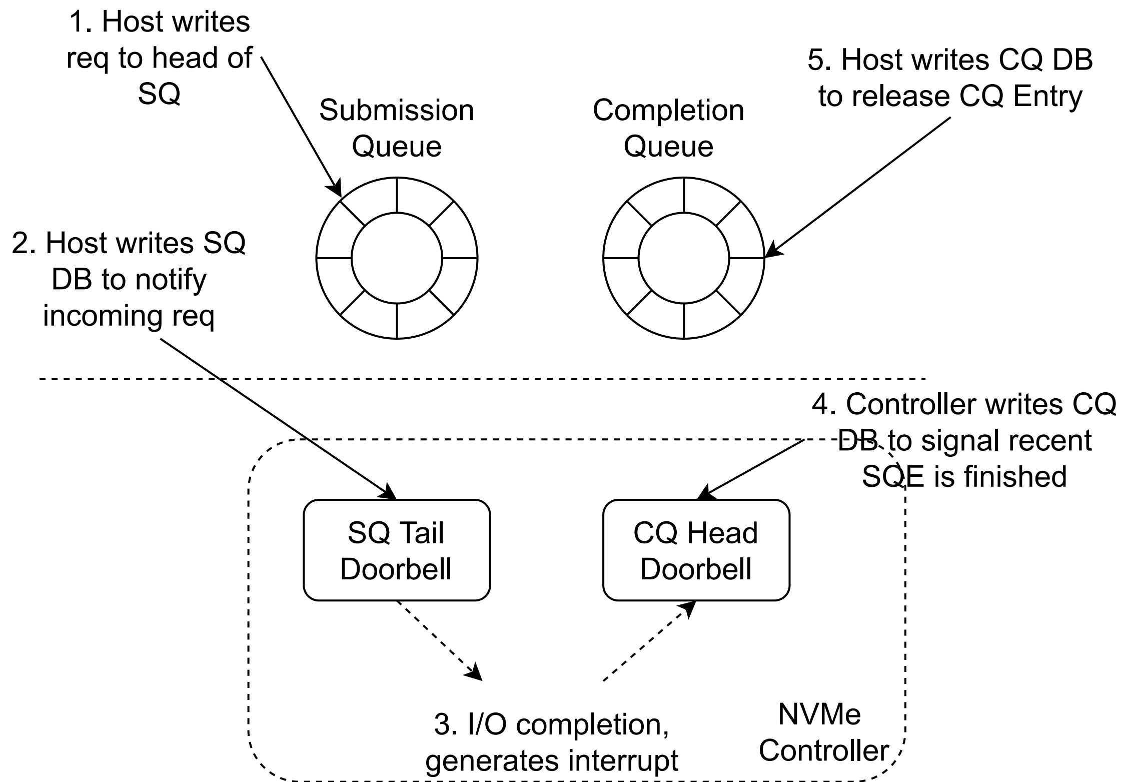


Figure 2.10: Host interacting with an NVMe device.

Robin (WRR) feature is enabled, the SSD firmware fetches I/O commands in a weighed round-robin manner as shown in Figure 2.12. Command queues have three priority classes (low, medium, and high), and queues in each priority class can have an assigned weight ranging from 1 to 256. Queues in the same priority class are first accessed in a round-robin manner, then their weights decide their proportionality in the WRR scheduling queue.

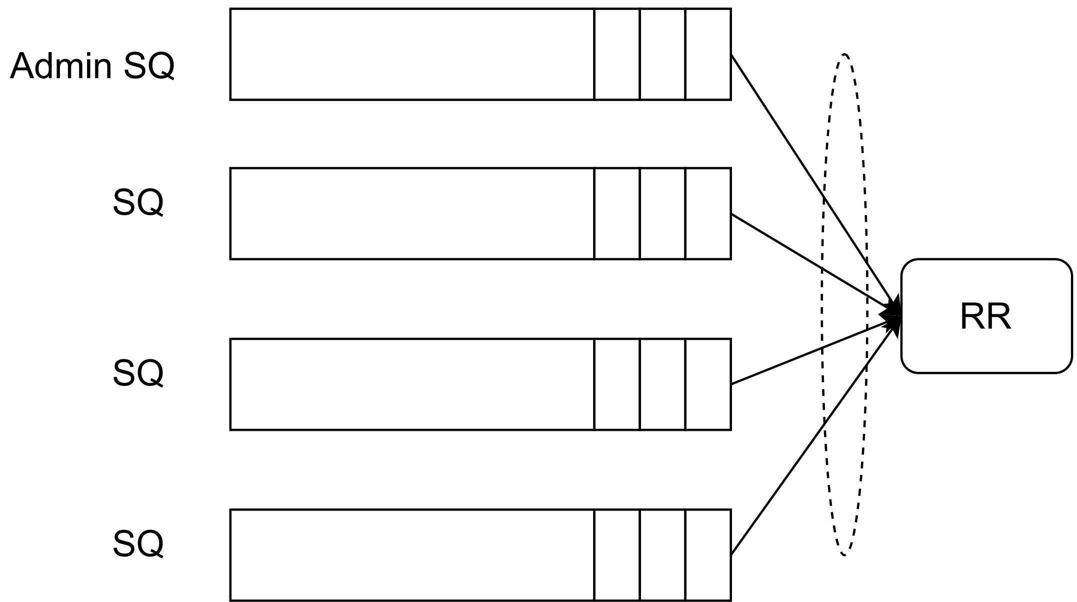


Figure 2.11: Round-Robin (RR) NVMe queue arbitration.

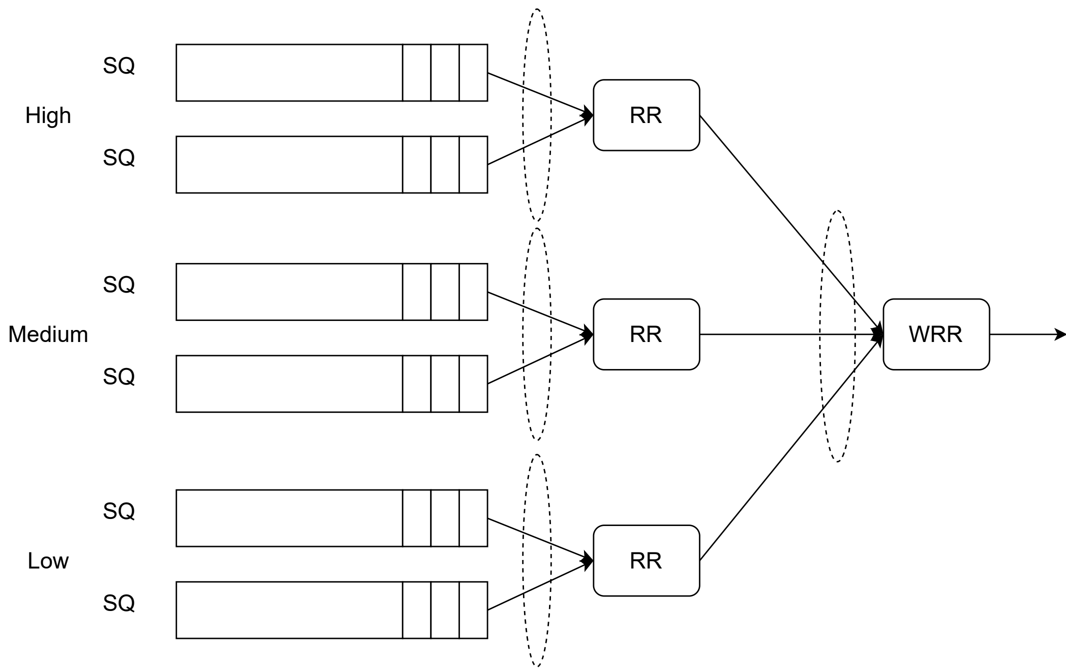


Figure 2.12: Weighted Round-Robin (WRR) NVMe queue arbitration.

2.1.10 Internal Caching

To increase the controller bandwidth and approach theoretical bandwidth of aggregated flash memories, modern SSDs use the on-board DRAM as data buffer cache. The write operation of a flash page takes significantly longer time than a read operation. Thus, caching new data in DRAM buffer and updating multiple pages simultaneously is proposed to hide the long latency of servicing a flash page write. Many commodity SSDs employ write buffers and serve reads off of it, with significant performance improvements [228, 272]. Internal DRAM also caches frequently accessed entries of address mapping table, which serves address translation much faster than reading from slower flash memories every time. Standard caching algorithms can be adopted, such as Least Recently Used (LRU) and Least Frequently Used (LFU), to increase caching efficiency to shorten the critical I/O path. However, the overhead of maintaining the mapping table cache still exists. Address translation can just take as long as a page read, which is unfortunately unavoidable because it is on the critical I/O path. What is worse, if the request needs an cache entry update, it takes time of a page program to persist the cache entry replacement in flash memories. When the cache hit ratio is low due to poor I/O locality or cache capacity, excessive flash memory accesses are needed for write-back and replacement of cache entries.

The performance instability caused by the internal cache design has received little attention so far. The I/O patterns of workloads are typically different from others, which implies that a generic cache organization/replacement policy does not work well for all types of I/O workloads [288]. Additionally, cache starving and thrashing problem may

emerge in an I/O intensive scenario since multiple I/O flows may compete for limited cache space [167]. For instance, a workload with larger block sizes/queue depth tends to occupy more cache than other workloads, even if other workloads have similar write ratios and temporal patterns.

2.2 Motivations

As modern SSDs and their protocols evolve to meet the changing demands, the system community needs an SSD simulator that reliably models their new features and reflects the real performance factors. This is because SSD vendors keep their FTL designs as top trade secret and never disclose the details to public, users and researchers cannot further understand the performance metrics as they do not have further insights of device internals. Unfortunately, existing SSD simulators [19,48] are either outdated (do not have support for newer multi-queue protocols, at 2018), not scalable (hard-wired configurations and implementations), or have faulty/incomplete assumptions: (1) they stop all of the events when GC is triggered in the background (2) they do not have an accurate modeling of write cache access, mapping table entry access and replacement, or non-trivial I/O command scheduling) (3) they do not take data transfer delay into account or consider realistic restrictions on flash commands. These inaccurate modeling techniques result in huge deviations between simulation results and commodity products. The engineering effort to extend existing simulators clearly exceeds that of building a new one. Therefore, we develop our own SSD simulator instead of extending an existing one (SSDSim, SSD extension for DiskSim [48]).

2.3 Performance Factors

Compared to mechanical disks, the performance specification of an SSD on paper looks perfect. However, its realistic performance on real-world workloads is subject to vary by many factors [75, 139, 278]. To evaluate how SSDs with different configurations and firmware designs perform in various environments, we must take their major performance factors into account and model them in faithful details. In this section, we analyze the key determinants to the performance of an SSD.

2.3.1 Flash Layout and Internal Parallelism

SSDs deploying different types of NAND flash have different performance as the cell access time varies. For example, Samsung's ultra-low latency Z-NAND SSD provides 5.5 times lower latency (around 10us for random read/write) than conventional SSDs with a new tier of NAND technology. 2D MLC and 3D TLC flash dies have different access speed and lifetime. Meanwhile, SSD layout decides the count of independent resources that can be accessed concurrently, the end-to-end bandwidth is a variable of them as a result. There are four levels of parallelism in SSD as shown in Table 2.1: channel-level, chip-level, die-level and plane-level. When serving an I/O request, the many related flash requests can be scattered across multiple flash memories. Once the physical address is determined by FTL, the related flash accesses can be parallelized over multiple flash channels, called channel striping. Further, each channel can pipeline I/O

commands and data transfer to maximize utilization of multiple flash chips within the channel. This process is called channel pipelining. Inside a flash chip, flash commands and data movement can be further interleaved and the multiple dies can work simultaneously, which in turn improves the bandwidth of a single chip by m times, where m is the number of flash dies within a chip. Within a flash die, there are multiple planes that can be accessed at the same time when certain conditions are met, which improves the bandwidth of a single die by n times, where n is the number of planes within a die. Ideally, all these four levels of parallelism can be exploited at the same time. The rich internal parallelism provides two benefits:

- Accessing data from multiple flash memories simultaneously produces high bandwidth in aggregate.
- The high latency of erase or program operations can be hidden by other operations as long as the I/O bus is not occupied. However, more parallelism does not directly translate into shorter latency.

Despite the rich hardware parallelism, studies [133] have shown that internal resource utilization decreases and flash resource idleness increases as the number of dies increases due to dependencies caused by I/O access patterns and bus contention. How to efficiently exploit all four levels of SSD parallelism remains an open research topic.

2.3.2 Page Allocation Scheme

A page allocation scheme determines which physical pages (physical page address) are selected to accommodate an I/O write request. A physical page is uniquely addressed

by its channel address, chip address, die address, plane address, block address and page address, as shown in Figure 2.13. Generally, there are two types of allocation schemes: one is static allocation [115], which is based on fixed striping across multiple flash memories. Depending on the priority order of striping, there can be a total of 24 static allocation schemes, Figure 2.14 illustrates one of the static page allocation scheme: Channel-Way-Die-Plane (CWDP). The other scheme is dynamic allocation [242], which assigns page to the range of an entire SSD, according to the idle/busy status of channel/chip, the erasure count of blocks, page offset and priority order of striping. Despite dynamic allocation provides more flexibility in exploiting parallelism, static allocation has advantage in implementation and anti-fragmentation. Generally, the priority order of striping has an impact on the overall performance for most workloads. Moreover, page allocation scheme decides the data placement, in many cases, partitioning hot and cold data is needed to achieve better GC and WL efficiency.

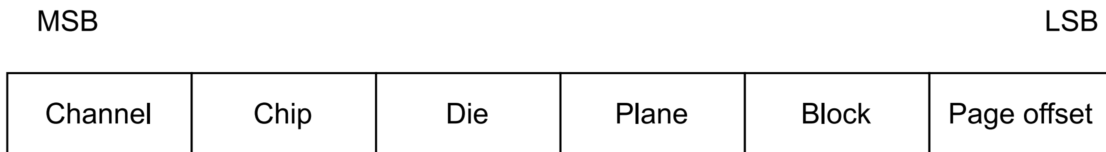


Figure 2.13: Page address encoding.

2.3.3 Host Interface

The SSD performance bottleneck has shifted from the storage media to the host interface as more channels and more sophisticated data buffering are available in the

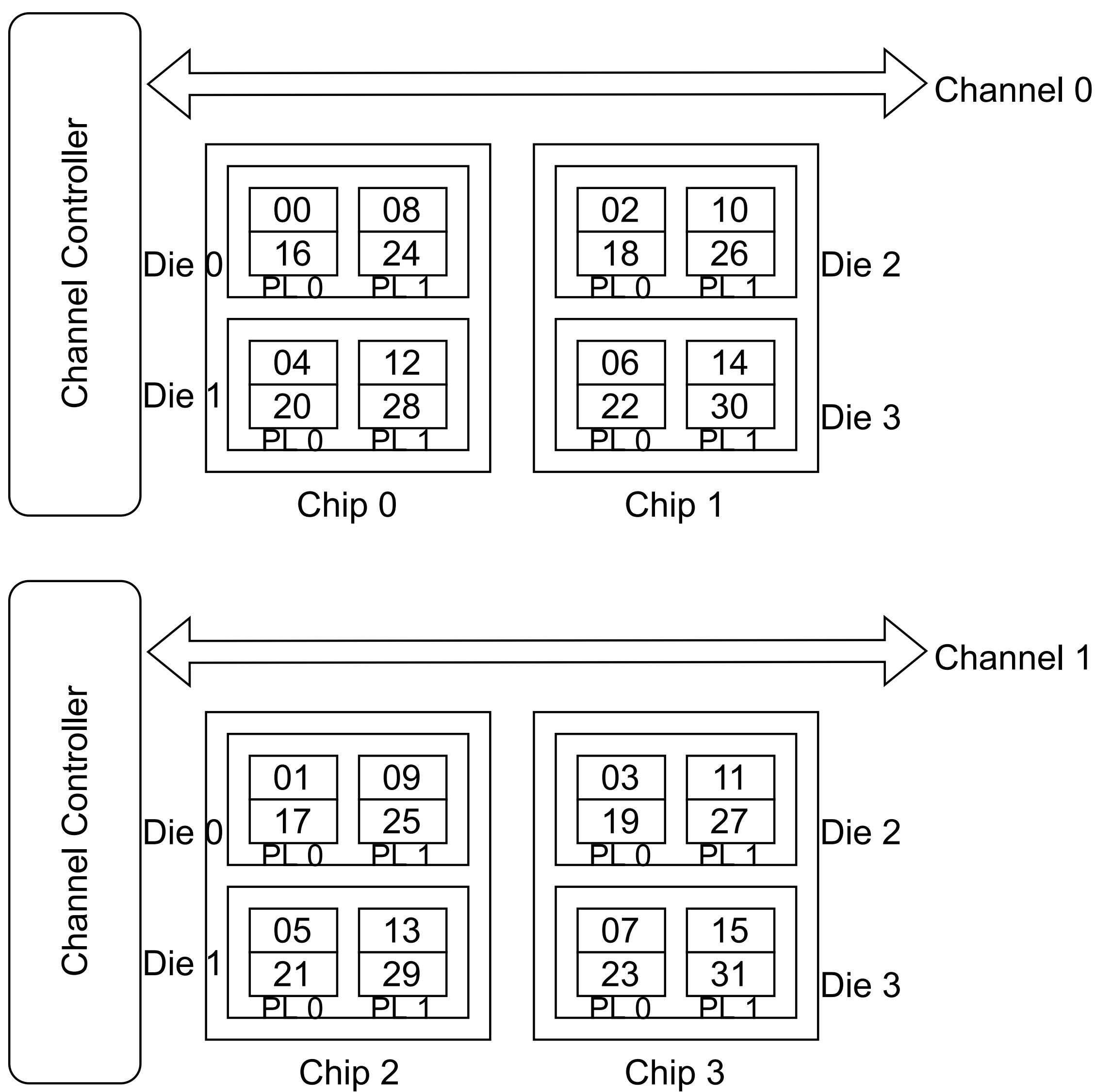


Figure 2.14: A static page allocation scheme: channel-way-die-plane.

SSD [85]. The PCIe host interface can provide much higher theoretical bandwidth by attaching the multi-channel SSD directly to the PCIe host bus. For example, a 4-lane (x4) PCIe Gen4 link can deliver up to 8GB/s data rates [233]. Recent commercial products [4, 15] have already been able to achieve this number. To exploit the massive internal parallelism, modern SSDs introduce NVMe, the interface designed to access fast non-

volatile memories, which supports up to 64K submission and completion queues capable of queuing up to 64K commands. Moreover, Linux kernel introduces a multi-queue block I/O layer to efficiently support NVMe SSDs [111]. This layer has two layers to facilitate scalability of host multi-core and multi-queue SSDs: the first level is the software queues (SWQs) to alleviate the lock contention problem in multi-core environments, and the second level is the hardware queues (HWQs) to dispatch I/O requests to submission queues in the multi-queue SSD, as shown in Figure 2.15.

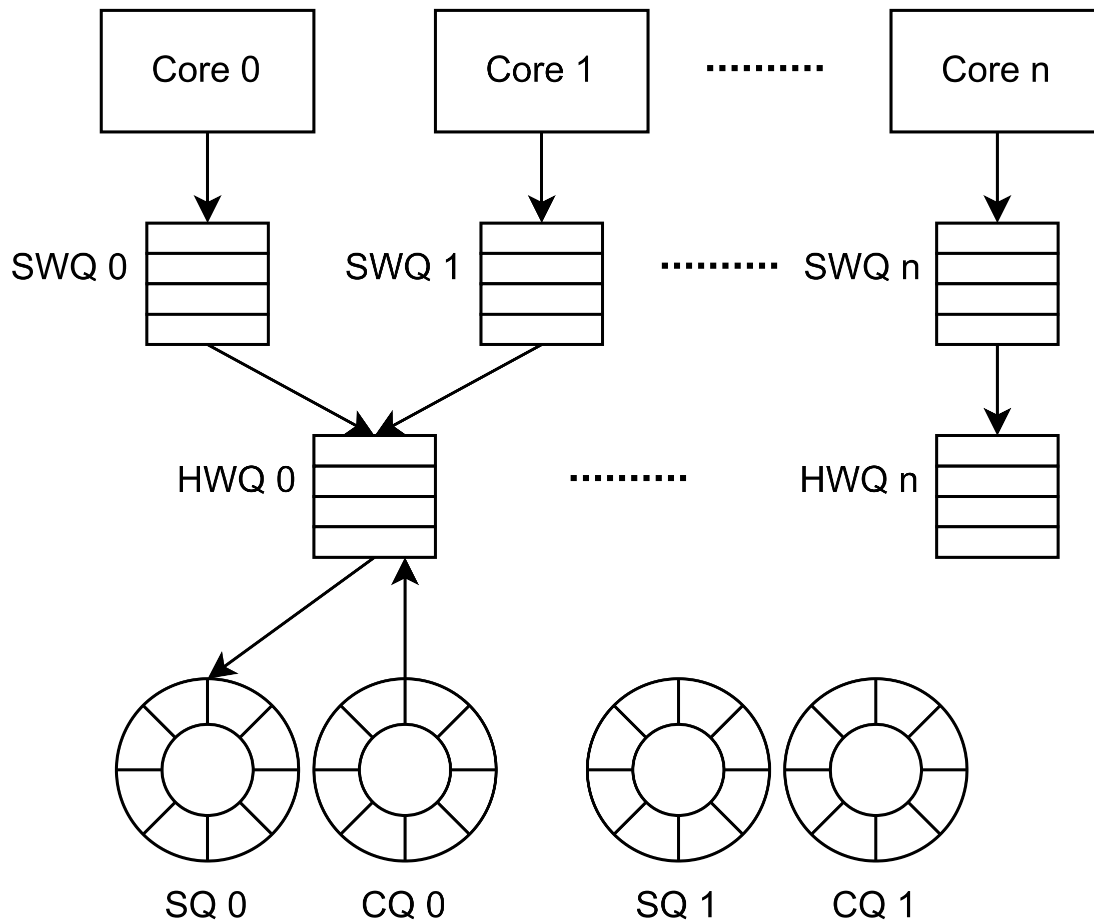


Figure 2.15: Multi-queue SSD interface and mqblk I/O layer.

2.3.4 I/O Transaction Scheduling

By adopting a multi-channel and multi-way architecture that allows many flash chips work concurrently, SSDs can quickly serve host I/O requests. It is essential to decide the optimal order of executing multiple I/O hosts to achieve higher performance. Since the NAND flash memory have severe read/write performance asymmetry, with millisecond-level erase and program latency and microsecond-level read latency, an intuitive way is to prioritize cheaper reads over high-latency erase and program operations. However, writes should not be starved in case I/O flows were paused by data dependency. There are two different approaches proposed in literature or used in commercial products [59, 83, 89, 188]: in-order scheduling dispatches incoming I/O transactions to idle flash chips by their arrival times. Out-of-order scheduling aggressively rearranges I/O transactions in the device queue as long as data dependency is not violated, in order to improve the resource utilization and overall throughput. Figure 2.16 presents a comparison between in-order scheduling and out-of-order scheduling. Suppose there are five flash operations from tail to head in the hardware queue: Read for chip 2, read after write for chip 1, erase then write for chip 0. In-order scheduling dispatches I/O transactions to the target chip by their order in the transaction queue. If the I/O bus or chip is busy, it just stalls processing the next request. Out-of-order scheduling can aggressively reorder and dispatch the queued transactions as long as the data dependency is not disturbed and the I/O bus and target chip is idle. As a result, out-of-order scheduling can drastically reduce the toal time to complete all I/O transactions. When SSD experiences intensive background activities such as GC, FTL should give priority to user I/Os by preempting background I/O or

maintaining two separate command queues with different priorities.

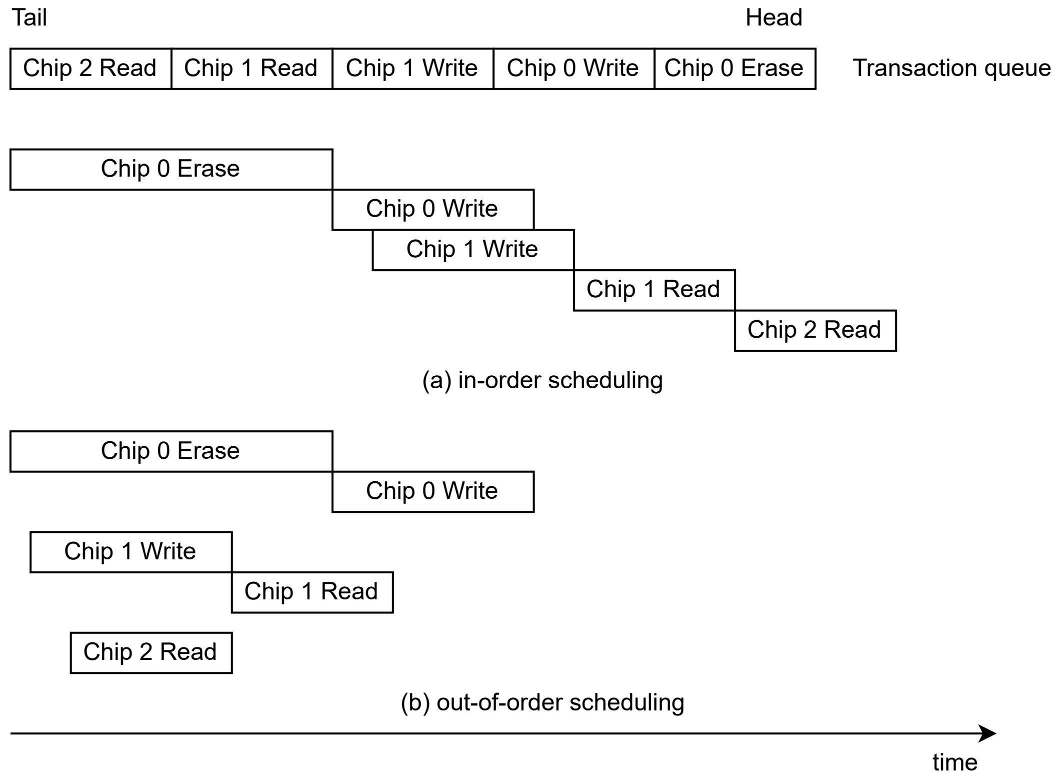


Figure 2.16: Comparison between in-order scheduling and out-of-order scheduling.

2.3.5 FTL Algorithms

The FTL is a thick software layer inside SSD. It is obvious that many FTL algorithms are the key determinants of SSD performance. First, flash management algorithms, including garbage collection and wear leveling incur a large amount of background traffic. For user I/O, GC and WL induced traffic is background noise, which results in huge fluctuation of user I/O latency and throughput. Therefore, the background traffic and user I/O flows should be properly dispatched to achieve optimal user I/O performance. When to perform

GC and which blocks to select as victim blocks are two major strategies in a GC algorithm. Selecting proper blocks to GC can minimize the data pages to move around. For example, the so-called greedy GC algorithm [50] always selects the block with the least number of valid pages, the cost is moving pages to a new block and updating the mapping entries. A possible optimization is to pick a new block from the same plane and use the copy-back command to move pages internally from one block to another. This can help reduce potential resource conflicts with other requests on shared channels and chips. Some studies [142, 258] show that some flash suspension commands (program/erase suspension) can suspend the on-going blocking program and erase operation to service pending reads and resume the suspended operation afterwards. This can achieve a near-optimal performance gain on servicing user I/O requests by reducing the interference of GC operations. The page allocation algorithm decides the data layout inside the flash memories, which indirectly affects the amount of data moved during the GC process and determines the background traffic. The address mapping algorithm decides the overhead of address translation, as discussed in Section 2.1.6, which is on the critical I/O path. Various FTL designs favor different I/O patterns. The sequential I/O pattern is most favored in most FTLs, its write amplification can get close to 1 and the SSD achieves optimal performance at the moment. When an SSD services random segmented (block size or even smaller) I/Os, more data blocks are moved during GC and performance can be significantly reduced.

2.3.6 DRAM Data Caching

Ideally, the internal parallelism of SSD allows concurrent accesses to address mapping table entries and actual data. Unfortunately, the mapping table query is on the critical I/O path, which makes the SSD unable to serve data accesses at the same time. Therefore, modern SSDs adopt complicated internal cache management policies that serve the following purposes: (1) data cache improves the response time of SSD, and increased the lifetime of SSD by servicing the write traffic with DRAM. (2) mapping cache improves SSD performance by caching mappings of frequently accessed pages. The cache allocation, capacity and eviction policies can be substantially different from each other, resulting in much different cache-hit rate and overall performance. For example, the cache capacity decides the amount of data that can be buffered in the DRAM. When the cache buffer does not have enough space to buffer the request, it must evict some data items from the cache buffer and then process the request. The SSD would have to experience some kind of stall due to the evictions. Eventually all dirty data must be written back to backend flash memories because data must be persisted into the flash in case of potential data loss caused by power outage. The more dirty data is held in the cache buffer, the longer it takes to finish the flush operation, which has a negative performance impact on subsequent I/O requests. Therefore, a larger data cache does not always imply better performance. The data cache can be configured as a write-cache only or a read-write cache. In the case of write cache, a write request is directly written to the cache, a read request can be served from cache if the data is found in cache, otherwise it is served from flash memories; In the case of read-write cache, a read miss would move the data from flash memories to cache.

The unit of data caching can be a block or a page. The eviction policy also decides the transaction flow to underlying flash chips.

2.4 Simulator Overview

Figure 2.17 gives an overview of the SSD simulator. The host interface implements the protocol that communicates with the host system. The FTL manages flash resources and processes I/O requests. The backend storage simulates data transfer and flash commands with detailed timings. We also model a functional internal DRAM that can be used to store mapping tables and buffer I/O requests. This integrated design allows our simulator to model the key performance factors of modern NVMe SSDs in reliable details. A series of internal statistics such as cache hit rate, block states and queue usage are available for performance evaluation. In this section, we describe the implementation details and the interaction between each module.

2.4.1 Flash Complex

We model a multi-channel, multi-chip hierarchical storage backend as shown in Figure 2.17, including flash memories, control units and interconnection I/O channels. Each channel is shared by a set of flash chips. Each channel has its own channel controller, which is responsible for dispatching I/O transactions to target chips and resolve resource contention. For simplicity, the scheduling policy is configured as First Read-First Come First Served (FR-FCFS) by default. For the flash model, we adopt a simple state machine model that captures major timings and states, such as read/write/erase timings, and page

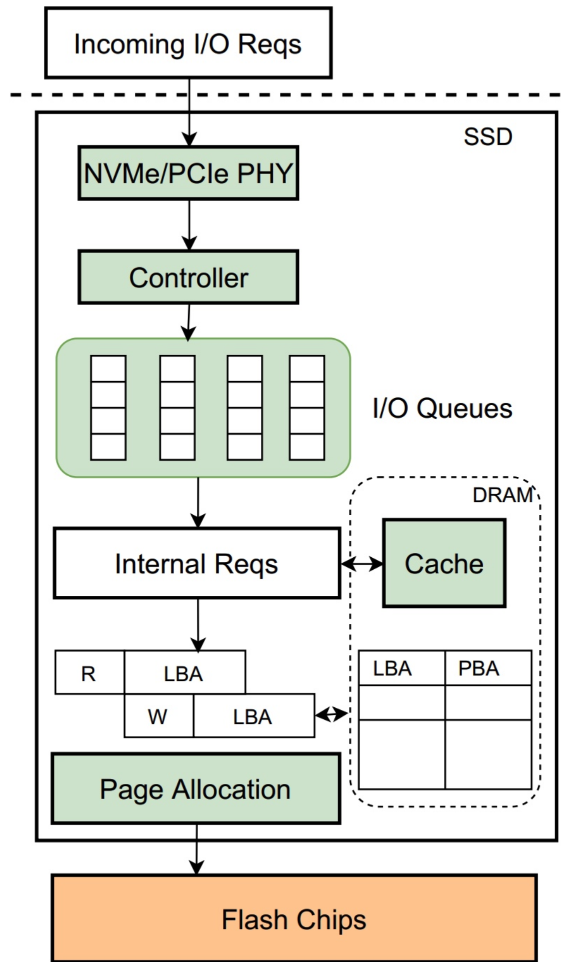


Figure 2.17: Overview of the SSD simulator.

states (blank/valid/stale). We fix a common problem in existed simulators that pages are not always written to the current offset within the block. To simulate the internal parallelism in faithful details, we carefully consider constraints of die-level and plane-level parallelism and advanced flash commands. All these geometries of the SSD is configurable, for example, the number of channels and the number of chips per channel. This allows researchers to study different flash configuration and its implication on parallelism without modifying other components. We also model a functional DRAM and its memory

controller to simulate a realistic memory model that caches data and mapping entries. The DRAM is dynamically updated by the FTL. As for data transfer, we only capture the latency based on flash and I/O bus timings.

2.4.2 FTL Modules

We implement all the major performance-related FTL modules discussed in Section 2.1.5, including the Greedy GC algorithm [51], configurable static page allocation schemes [132] and page-level mapping scheme [197]. By default, our FTL performs GC and WL based on the number of valid pages to migrate. Intra-plane data copy-back is prioritized to move valid pages from one physical block to another to reduce performance penalty of GC operations. FTL strips I/O requests across multiple channels and ways based on the page allocation scheme and dispatches flash-level transactions to flash chips. We also model a functional DRAM that can be used to cache I/O requests and mapping table entries. We implement demand-based page-level mapping as the default mapping policy. A portion of DRAM is used to cache frequently accessed mapping entries. The entire mapping table is stored in flash memories. If the requested mapping gets a cache miss, the cache manager promotes the mapping entry into the mapping cache. The caching layer can be configured with different associativity, caching and replacement policy as discussed in Section 2.3.6.

2.4.3 Host Interface

The host interface module is responsible for fetching and scheduling I/O requests from input traces. The request fetch rate is configurable to model how real NVMe SSDs

process incoming requests from host submission queues. The host interface models a functional NVMe protocol/queue management that performs request dispatching in the device-side queues using a First-In-First-Out (FIFO) policy. The three different priority hardware request queues and the I/O arbitration mechanism (Weighted-Round-Robin, WRR) specified in the NVMe whitepaper [6] are modeled to provide close to native NVMe I/O scheduling.

2.4.4 I/O Flow within the Simulator

When an I/O request arrives at the SSD, the host interface inserts the request in a device queue and parses it into several page-sized transactions, each of which has a specific LPA, because the block size of an I/O request can be as large as multiple pages. Next, FTL checks if the transaction is a write, if yes and the write caching is enabled, the DRAM caching manager immediately caches the request and its metadata, and returns a response to FTL. Otherwise, the FTL should translate the LPA of that transaction into a PPA. A LPA-PPA mapping table is maintained to keep track of the physical location of an LPA. Translating an LPA into a PPA is accomplished in two steps: First, FTL uses a static mapping scheme to determine the channel, chip, die and plane by modulo calculations, which users can configure with their preference. By default, the priority of striping LPA is from channel, chip, die and plane (CWDP); Second, FTL uses the page-level mapping table to redirect the page to a PPA within that plane. After the address translation, FTL issues the transactions of related PPAs to the underlying flash controllers and memories. When there is no available page for write, FTL performs GC to reclaim new blocks by

searching for and cleaning victim with Greedy GC algorithm. During GC, FTL copies all valid pages of the victim blocks, writes them into new blocks, and updates the mapping table for the moved pages. The flash scheduling unit schedules flash transactions for each chip using the simple yet effective First Read-First Come First Served (FR-FCFS) scheduling policy. FR-FCFS can alleviate the impact of high write/erase latency on reads. To prioritize reads over writes, the flash scheduling unit maintains a separate read/write First-In-First-Out (FIFO) queue for each chip, where the queue depth of the write queue is larger than the read queue. The flash scheduling unit is also responsible for guaranteeing timing correctness for flash chips and I/O bus. After processing the I/O request, the flash scheduling units return the consumed time for each sub-request to FTL. FTL then concludes the overall latency for the I/O request.

2.4.5 Key Features

- **Detailed:** Our simulator models major performance factors of NVMe SSDs, supporting complete FTL functionalities. It allows us to study the internal nuances and investigate performance implications.

- **Modular:** Each module in the simulator is designed to decouple from the rest of the system for flexibility and extensibility. Therefore, researchers can modify/insert their own page mapping algorithms, GC strategies and other parts for design space exploration.

- **Accurate:** The accuracy of the simulator is verified by the correctness of the internal states, such as the block states, bus rate, queue usage and cache hit ratio.

We validated against these internal states by multiple unit tests.

2.5 Evaluation

Due to the black-box nature of SSDs and limitations of simulating them as full-fledged embedded systems, it is hard to extract absolute performance metrics and compare them to commodity products. However, by analyzing internal metrics, an accurate SSD simulator can still serve as a good research platform to understand the performance factors, explore the design space and provide system-level insights for both firmware designers and programmers. For example, when performance change is observed for various workloads, users can analyze the inconsistency and pinpoint the bottleneck by reading the internal states of the SSD simulator. Users can also propose device-level innovations and benchmark through real workloads. We present various types of studies enabled by our simulator in this section, including device-level exploration, impact of I/O patterns on the SSD performance, implications for getting better and sustainable performance.

Table 2.2 lists the specification of the simulated SSD. The layout and flash timings are configured to make the performance comparable to an enterprise-class SSD [13]. We run synthetic benchmarks and database experimental data is collected after 75% of the SSD capacity is filled up with data, to ensure that GC is frequently performed. The major performance metrics are IOPs (throughput), average request latency, and tail latency. We evaluate the SSD internal behaviors and sensitivity to different configurations, and discuss design trade-offs of various internal mechanisms.

Table 2.2: Configuration of SSD simulation.

SSD Layout	8 channels, 4 chips/channel, 4 dies/chip 2 planes/die, 2048 blocks/plane 512 pages/block, 4KB page
NAND Flash Spec	tREAD = 80us tPROG (tW) = 1000us tERASE = 3ms
SSD System	PCIe gen3, NVMe 1.4 2GB internal DRAM, 1 channel, 1 rank 8 bank, access latency (tRAM) = 50ns Page-level mapping, greedy GC 1920MB cached mapping table, 128MB LRU write cache
Workloads	Synthetic FIO: 100% write (W), 70% write 30% read (HW) 50% write 50% read (RW), 70% read 30% write (HR), 100% read (R) Application: YCSB-C on RocksDB, TPC-C varmail, fileserver, oltp (filebench), fin, web, sql

2.5.1 IOPs and Latency Under Varying I/O Queue Depth

Figure 2.18 show the IOPS delivered by our simulator with varying number of I/O queue depth (from 1 to 32), when one of the five synthetic workloads (traces) is executed. The five synthetic I/O workloads are 100% write (W), 70% write-intensive

(HW), 50% read and 50% write (RW), 70% read-intensive (HR), 100% read (R). We configure the simulators with an educated guess of the internal structures of a commodity enterprise-level SSD, to perform the performance validation. The closer the simulator IOPs compares to the realistic measurement, the more accurately it can model real devices (despite it is very unlikely since we do not know their exact physical specifications (e.g., the number/size of dies/blocks/pages, read/write latency) and many unrevealed FTL details). As the number of queue depth increases, the IOPs curve from the simulator scale in a similar trend as that of the commodity SSD. The real SSD exhibits steady and saturated throughput once the queue depth reaches around 16. However, the simulator IOPs tends to grow linearly with queue depth in all workloads. The difference in inflection points is probably because commodity SSDs adopts some secret prefetching algorithm that can quickly saturate the throughput of flash backend by predicting the I/O pattern. In terms of access latency, our simulator scales in an appreciable manner compared to the off-the-shelf SSD, the absolute error ranges from 16.2% to 34.2% for random read requests. Figure 2.5.1 shows the average access latency of sequential read, random read, sequential write and random write accesses with varying queue depth.

Compared to the realistic number when queue depth is 1, the error rate in terms of absolute IOPs ranges from 24% - 185%. This suggests that it is extremely difficult to build an SSD simulator for an apple-to-apple comparison in terms of absolute performance metrics, even the architecture and timing configurations are configured to match commodity products (yet we do not know). However, we can simulate on different set of flash configurations to approach the realistic performance curve of the off-the-shelf SSD, as shown in Figure 2.18.

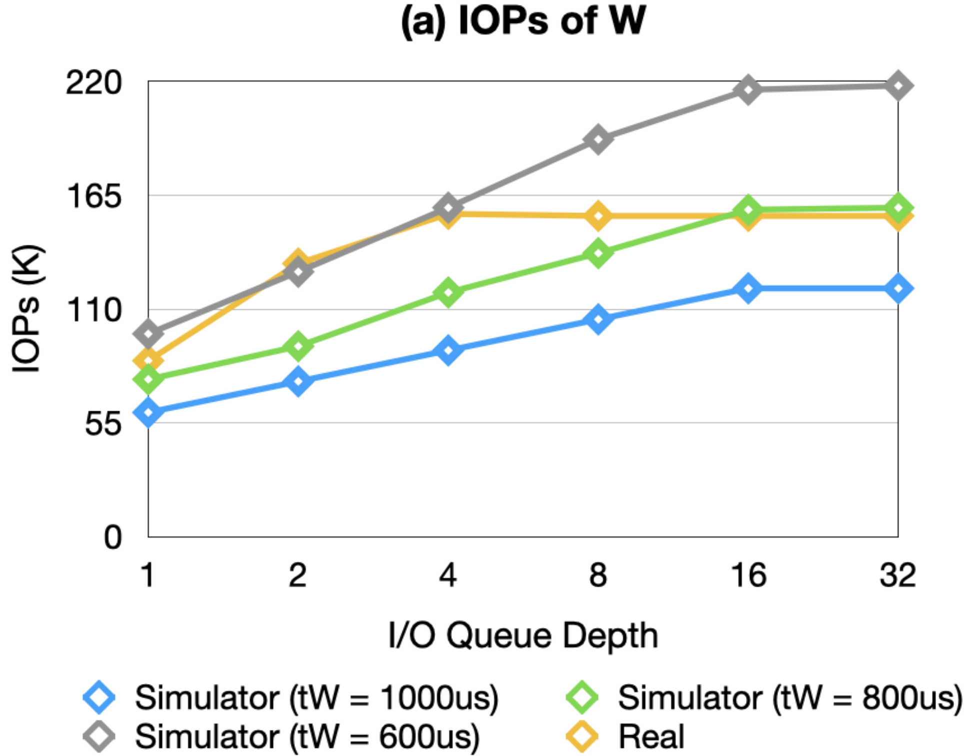


Figure 2.18: Write (W) IOPs comparison between multiple simulation configs and a real SSD with various queue depth.

2.5.2 IOPs with Varying Number of Flash Channels

Figure 2.23 show the IOPS trend with varying number of channels, from 4 to 32. The closer the IOPS is linearly scaled, the more accurately it can emulate an ideal SSD. Ideally, the total throughput of the SSD scales linearly with number of channels until the aggregated internal bandwidth reaches the external PCIe limit.

Insight: Channel-level parallelism is in the easiest form to leverage as each channel operates independently. Modern enterprise SSDs adopt tens of channels, wide internal

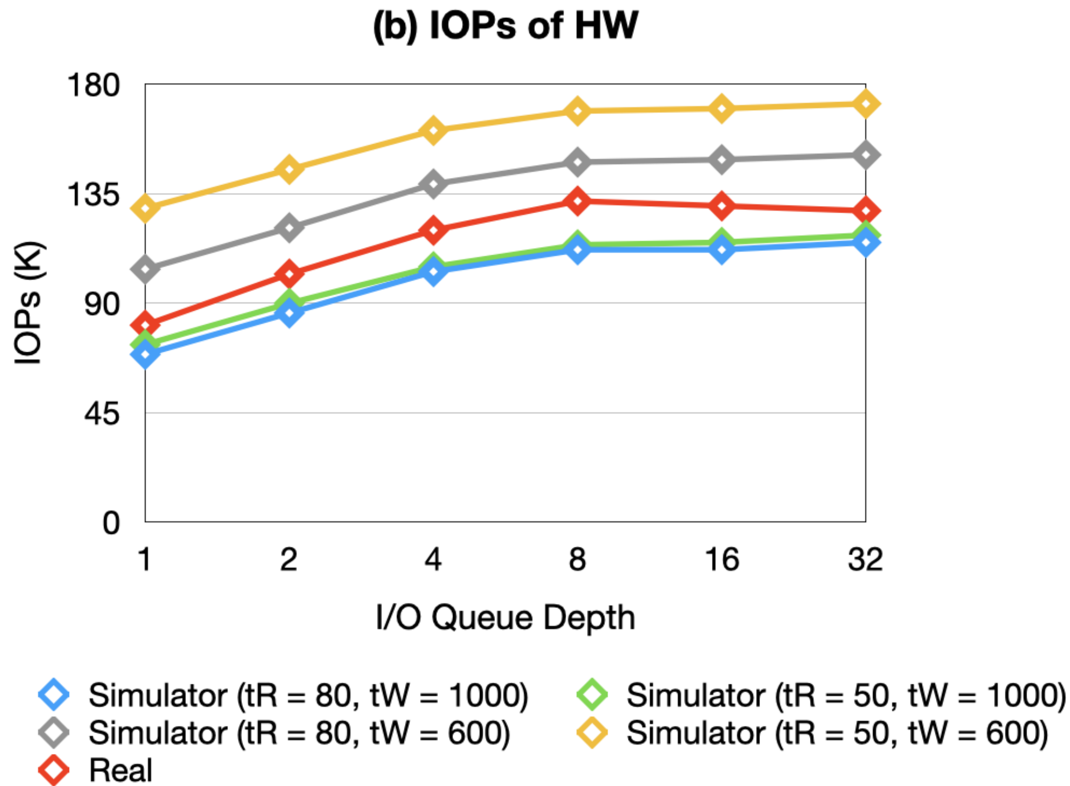


Figure 2.19: Heavy-write (HW) IOPs comparison between multiple simulation configs and a real SSD with various queue depth.

data bus and sufficiently large DRAM for write buffer and placing the entire mapping table to resolve internal bottlenecks. Besides these storage-side measures, storage system architects may also need to pay attention to potential bottleneck and contentions in the device-side queue and physical interface. It remains a challenge to expose the multi-channel architecture to OS and data-intensive applications.

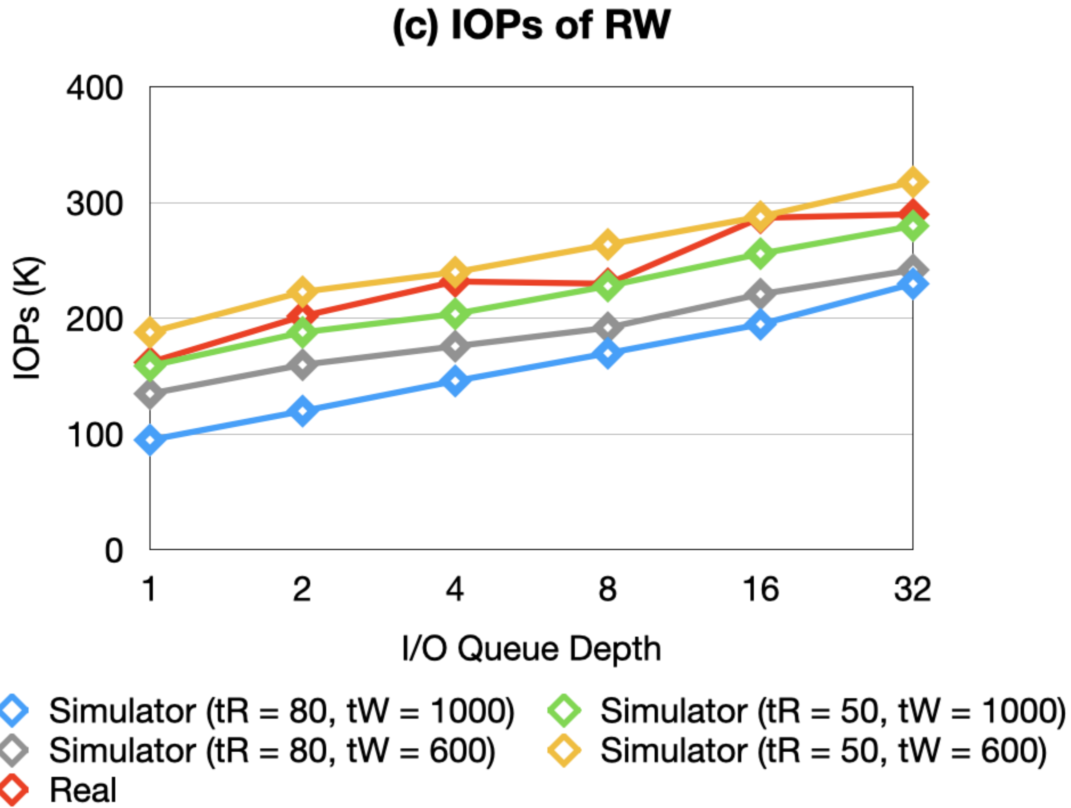


Figure 2.20: Read-write (RW) IOPs comparison between multiple simulation configs and a real SSD with various queue depth.

2.5.3 Impact of Page Size

Modern SSDs are adopting larger flash pages to enable higher capacity. This can be a double-sword to performance: larger page size increases the efficiency of serving page-size requests; however, for an SSD adopting large flash pages, it occurs more frequent that small write requests only update part of a page, the overhead of updating partial page is significantly higher: First, the entire page data including the sector to be updates need to be read to the register, then the controller modifies the corresponding sector of data,

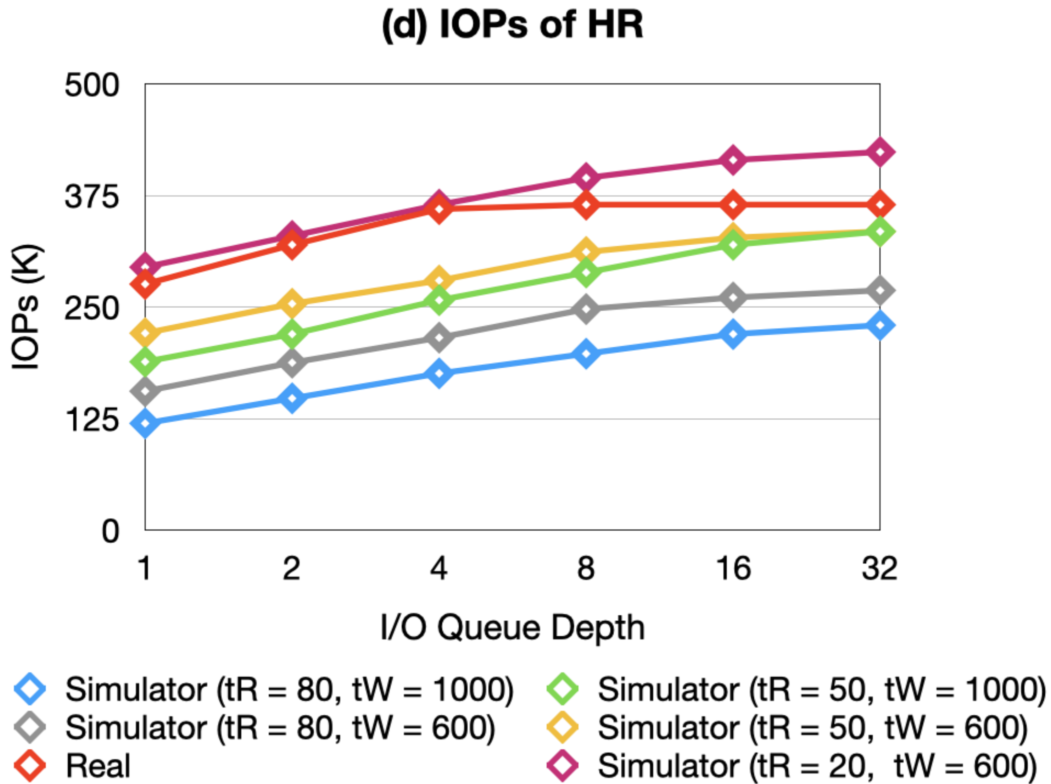


Figure 2.21: Heavy-read (HR) IOPs comparison between multiple simulation configs and a real SSD with various queue depth.

finally the entire updated page data is written back. this is the so-called read-modify-write operation. In this subsection, we evaluate the performance impact of varying page size. The workload for evaluation is a 100% synthetic random write (queue depth = 16) with varying request size from 4KB to 16KB. Figure 2.24, Figure 2.25 show the average response time and throughput, respectively. One can observe the closer request size gets to the page size, the better average response time is. This is because less partial updates are incurred when request size are multiples of page size. However, it is clear that the response time sees a sharp uptick when the request size is sub-page sized. In terms of

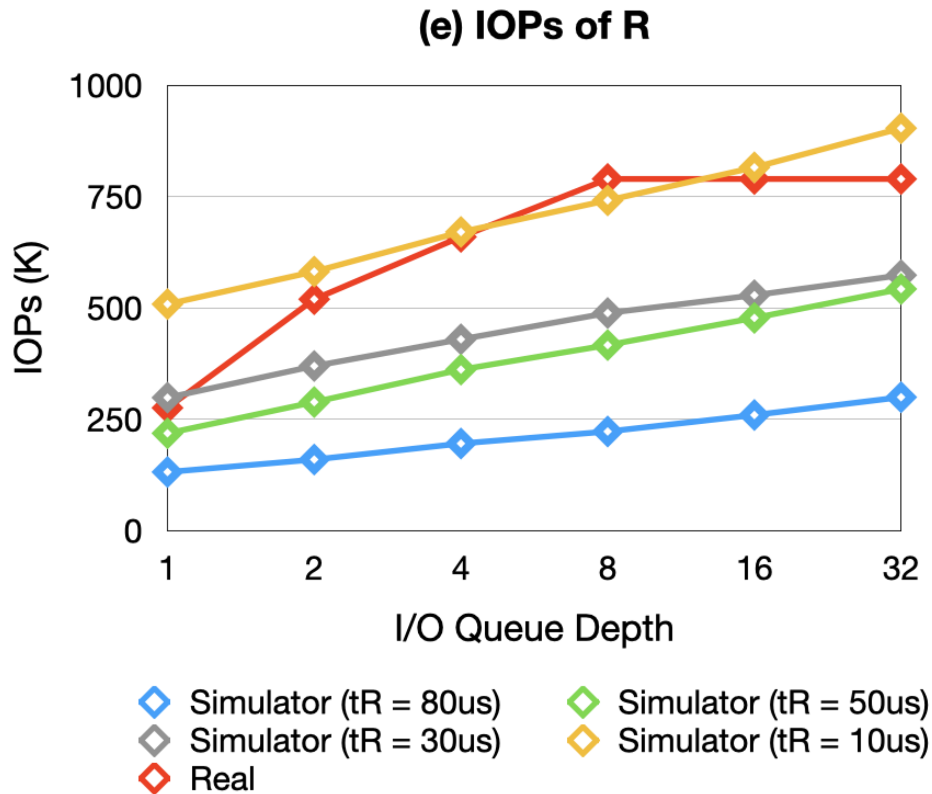
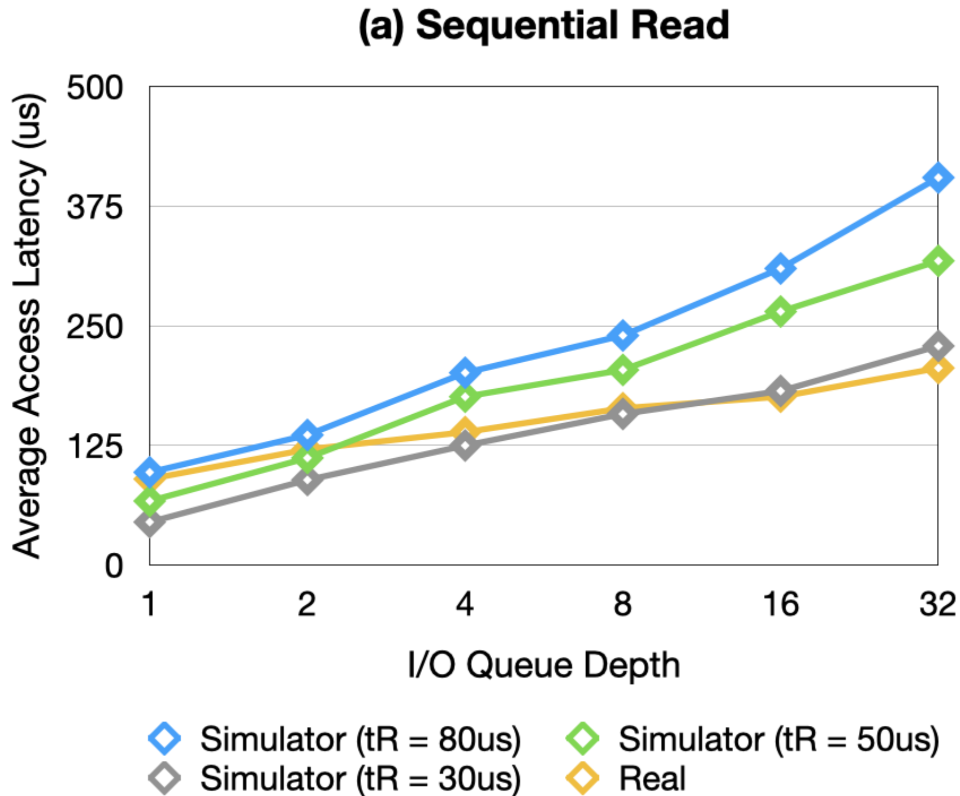


Figure 2.22: Read (R) IOPs comparison between multiple simulation configs and a real SSD with various queue depth.

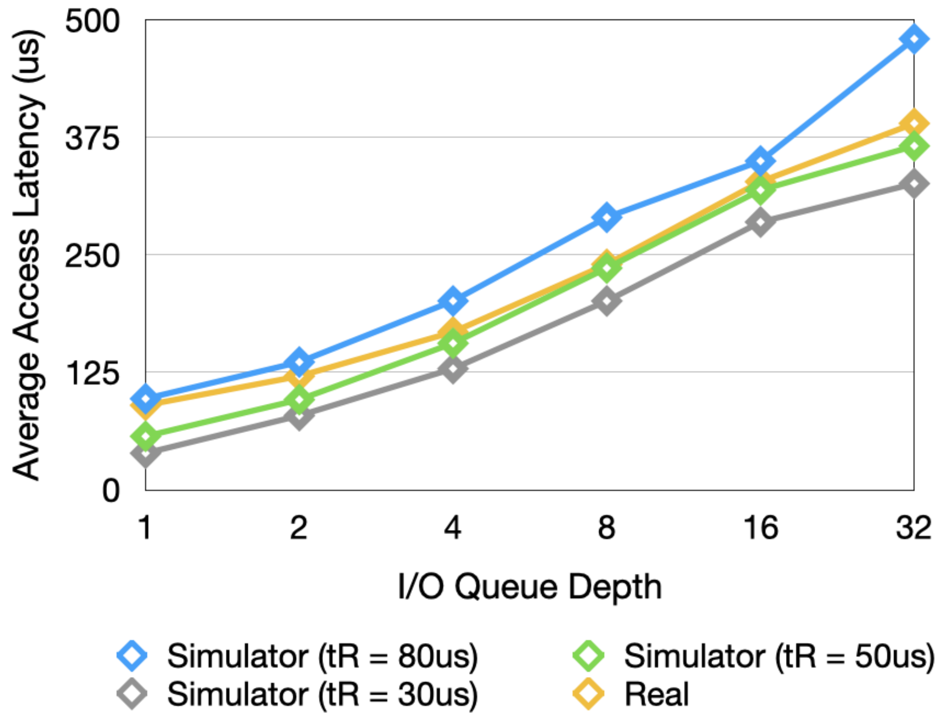
throughput, it is minimally impacted by page size because FTL parses the request into multiple page-sized transactions before dispatching them into flash chips, thus the rich parallelism remains nearly the same.

Insight: As most OS requests are sized around 4KB, increasing the capacity by adopting larger page flash faces latency challenges. SSD designers should look for efficient partial update strategies or finer granularity address mapping scheme (sub-page based) to offset the response time penalty when handling sub-page I/O requests.



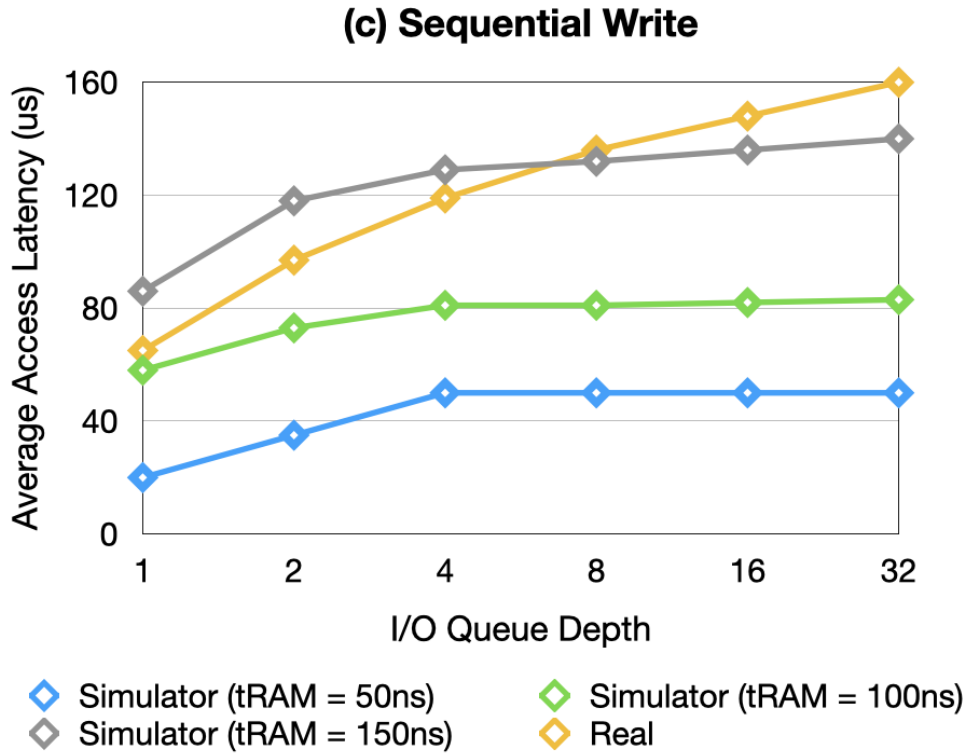
2.5.4 Impact of Read and Write Interference

SSDs are asymmetric in terms of read and write operations. If a read request arrives at a flash channel that is busy serving a write or erase operation, its completion time will be significantly delayed. Table 2.3 shows the impact of concurrent writes on read operations. We use FIO benchmarks to produce 4K random read and sequential write requests. The random reads achieves an average of 69.2MB/s bandwidth after stabilizing. The read bandwidth drops to a mere 11.8MB/s when random reads and sequential writes are mixed as 1: 1. Note we disable write buffer to prevent its alleviation on the write contention. We measured the average stalling time of read requests per channel, it is



7.9x longer when background write I/O presents. Further, we modify the write trace to redirect it into a specific channel. The read bandwidth receives a significant relief, reaching 60.8MB/s. The reasons lie in several layers: (1) I/O scheduling. The read and write interference exacerbates when the I/O scheduler naively follows a First-Come-First-Serve (FCFS) policy because write/erase operations consumes 10x to 50x time than reads. To counter the write interference, SSD firmware designer can either introduce request reordering (prioritize reads over writes) in the controller queue, or suspension command that can stop erase/write operation to prioritize reads. We model the latter approach to test its capability to combat write interference.

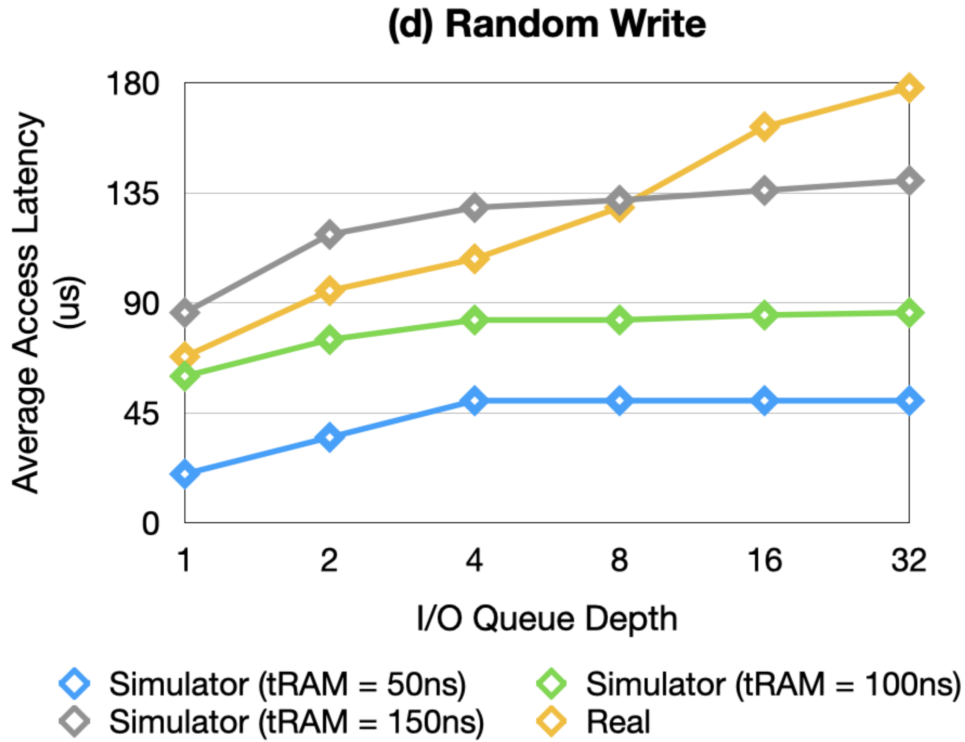
The write-disturb problem is a major factor that impacts QoS of user I/O. Although



it can be mitigated by optimizing the SSD I/O controller, we do not regard that as a perfect solution as long as the read and write requests flow into the same channel. It needs cooperation from the storage stack and upper applications for a more efficient data layout scheme, depending on I/O patterns of applications, such as lifetime, hotness and sequentiality. OpenChannel [42] is a popular emerging interface that enables a seamless integration between storage software and underlying SSD.

2.5.5 Impact of Data Buffer

In this subsection, we evaluate the performance impact of a data buffer. We use the average response time per request as the performance metric and vary cache buffer size



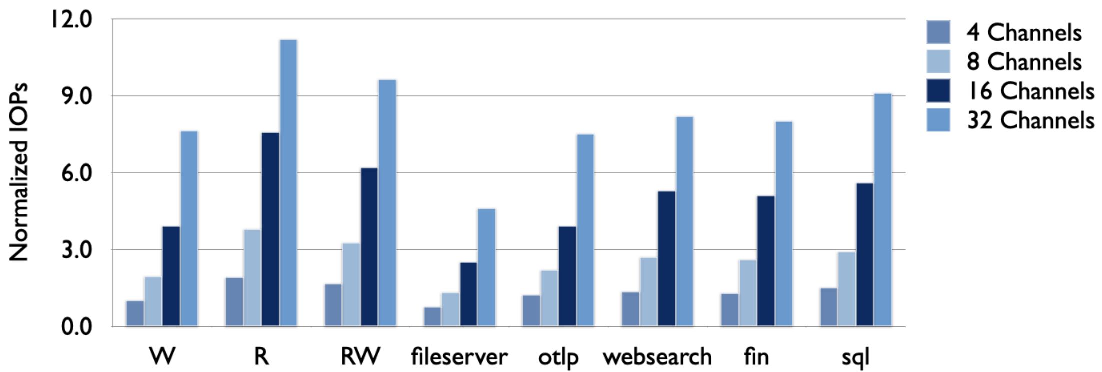


Figure 2.23: IOPs trend with varying number of channels.

from 0 to 128MB. In this experiment, we adopt fully associative write page-based cache, caching both recently-written at page granularity. Figure 2.26 shows the average latency per request of real-world workloads.

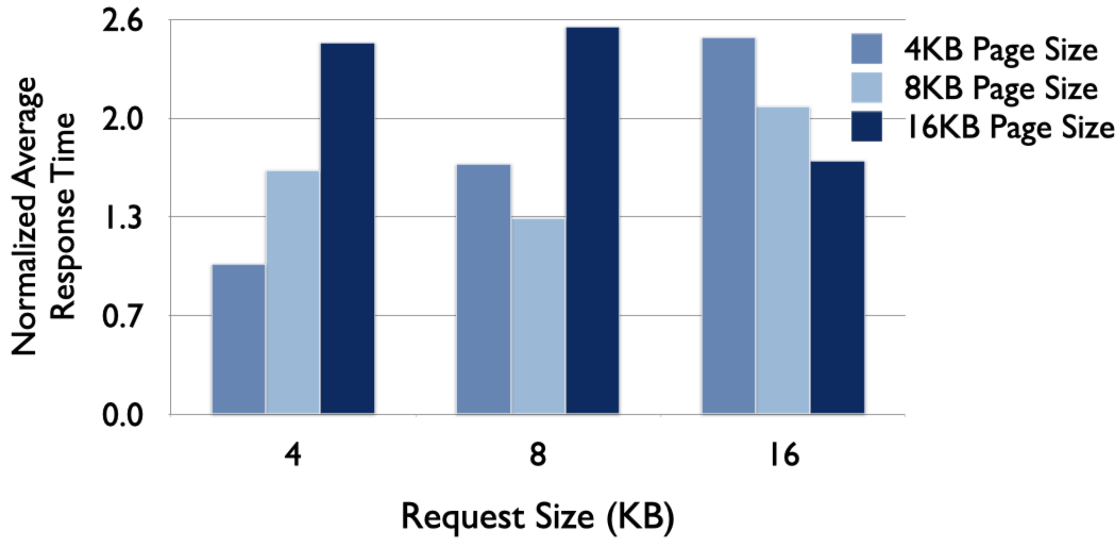


Figure 2.24: Normalized average response time with varying page size.

Table 2.3: Average bandwidth and queuing time of random reads (RR), RR + sequential writes (SW), RR + channel-fixed SW, RR + SW with write suspension, bandwidth (MB/s), Queue waiting time (us)

	RR	**RR + SW**	**RR + channel-SW**	**RR + SW w/ suspension**
Bandwidth (MB/s)	69.2	11.8	60.8	42.4
Queuing Time (us)	90	852	147	219

2.5.6 Impact of Caching Mapping Table

As discussed in Section 2.1.6 and Section 2.3.6, our cached mapping table (CMT) works as follows: for a read request, if the cached mapping table is full, a victim entry is selected and written back to the mapping table stored in flash memories if the victim entry is dirty. If a cache miss, the mapping entry of that page is promoted into the cached mapping table. For a write request, after the FTL determines its PPA based on page

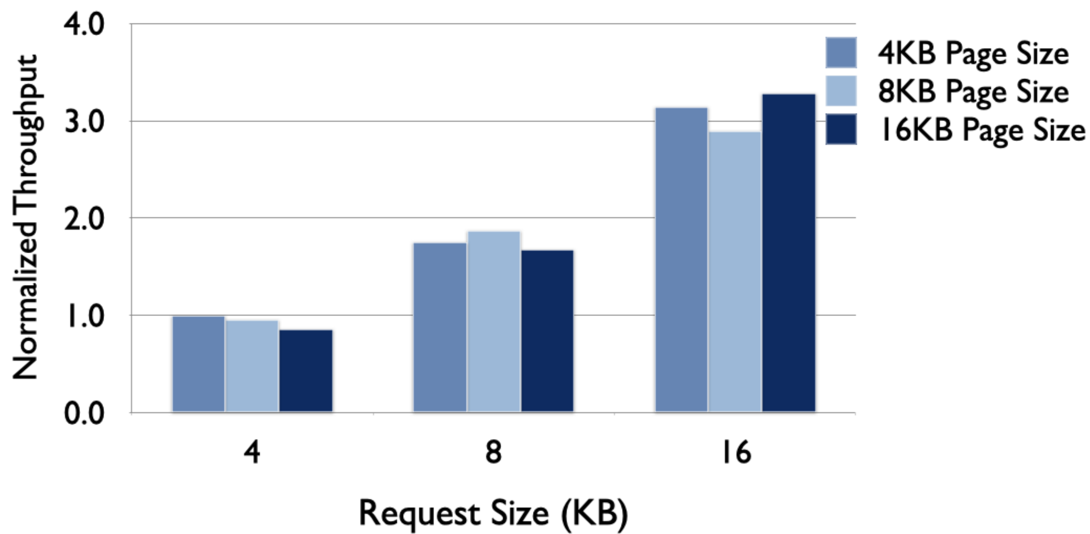


Figure 2.25: Normalized throughput with varying page size.

allocation scheme, the mapping info is updated/persisted into the mapping table in flash memories. Whether the mapping entry is cached in DRAM depends on the caching policy. It is obvious that cached mapping table could alleviate the address translation overhead. However, it comes with a price - the write-back operation is costly because it incurs a write operation on the flash memory. For workloads with poor locality, caching mapping entries may incur twice as many flash writes if cache miss in the mapping table occurs frequently. To evaluate the performance impact of caching mapping table, we compares IOPs and the average response time between no_cmt and vary CMT size from 512MB to 2058MB. Figure 2.27 and Figure 2.28 show IOPs and average latency per request of six real-world workloads.

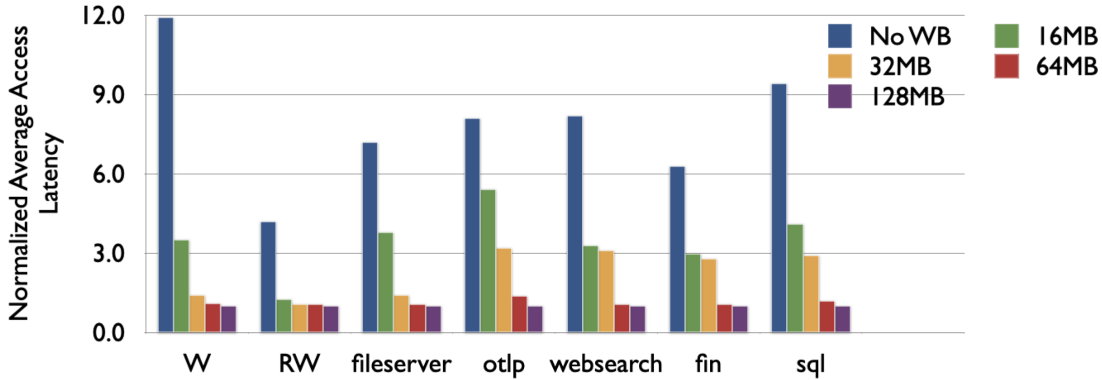


Figure 2.26: Average access latency per request of real-world workloads, less is better.

2.5.7 Impact of Data Cache Contention

Write buffer is an essential design to hide the response time of write requests, especially for applications with high locality. If the reserved capacity for write cache is not large enough, the entire cache can be filled up quickly by write-intensive applications, causing significant cache thrashing and spikes in response time. What is worse, the data cache is blindly shared by all user I/Os. Even if a random write flow doesn't enjoy much benefit of write caching, it might be cached in the data buffer, starving other flows that may benefit more from the write caching. We illustrate the performance impacts of data cache contention in Figure 2.29. We select FIO 16K random write (RW) and 16K sequential write (SW) as the workloads. To make sure that flash chips and buses are not the contending sources, we modify the address mapping scheme to dispatch RW and SW I/O flows into different channels. The intensity of the I/O workload are varied by its queue depth. By varying the queue depth of RW, we can observe that the bandwidth of SW, which heavily relies on write caching, suffered drops from 34.5% to 7.8% compared

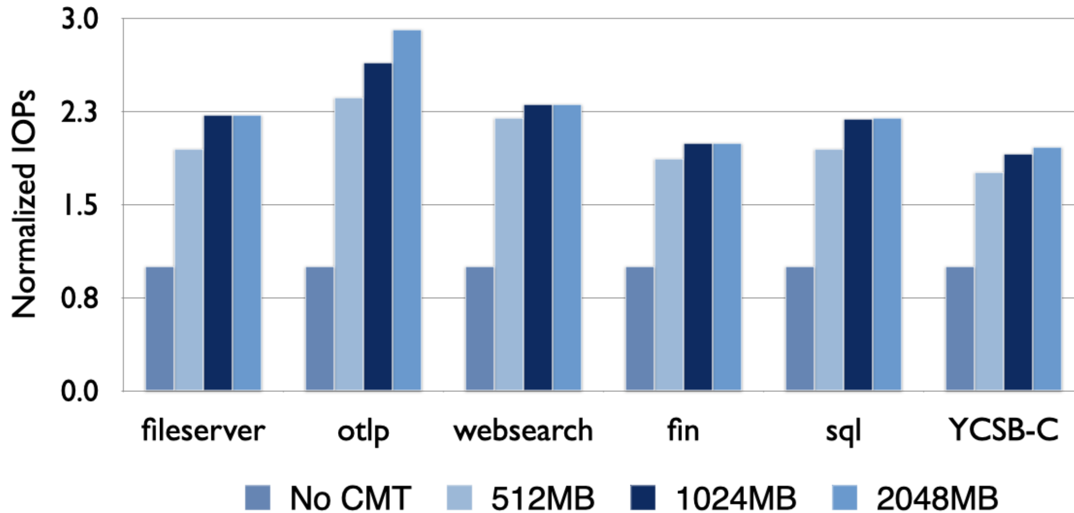


Figure 2.27: Normalized IOPs of real-world workloads, higher is better.

to its bandwidth running alone. Figure 2.30 shows the same setup scaling with various cache size. Besides increasing the write cache size, implementing efficient and fair share of write cache is crucial to guarantee QoS for multiple I/O flows.

Insight: I/O requests from multiple flows can interfere with each other when accessing the data cache. Hence, data cache policies should be designed to consider the potential interference which may result in unfairness. Flash-level parallelism should also be leveraged to minimize the write-back overhead from the data cache.

2.5.8 Impact of Page Allocation Scheme

In order to improve SSD performance, there are four different level of parallelism to be exploited, as discussed in Section. As for possible static page allocation schemes, there may exist an optimal order of them to maximize the SSD performance as suggested in [115]. In this subsection, we compare the performance of several representative static

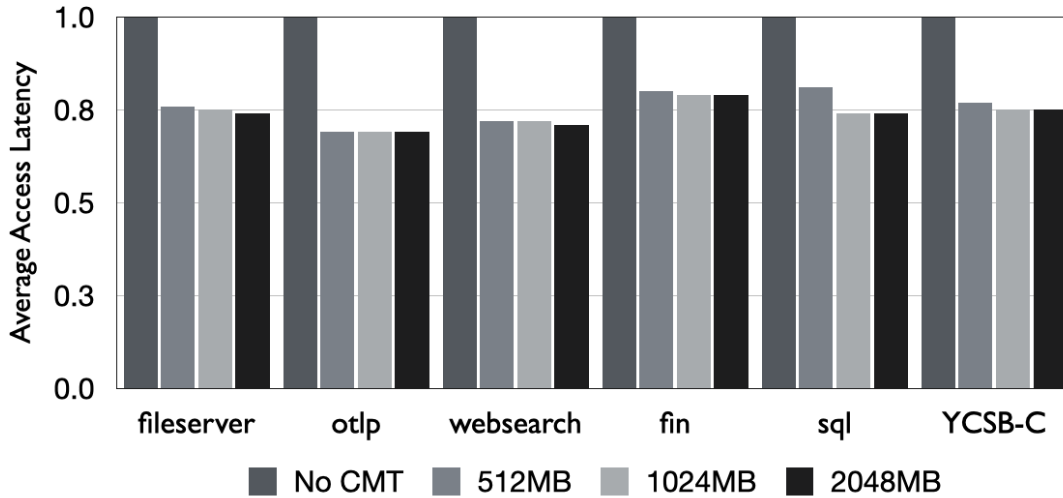


Figure 2.28: Average access latency per request of real-world workloads, less is better.

page allocation schemes. Figure 2.31 shows the normalized IOPs of various static page allocation schemes on multiple real-world applications.

2.5.9 Impact of Multi-Plane Operations

To achieve higher performance, flash chips support several advanced commands such as multi-plane and copy-back operations. Multi-plane operations allow simultaneous access to the planes in the same die. Intuitively, using multi-plane commands can double the theoretical bandwidth. However, they have extremely tight prerequisites: the operations should be the same type, and operate at the same plane offset (aligned). Our experiments show that aggressively enabling advanced flash commands may not bring extra performance benefit for many workloads, especially in an aged SSD. We evaluate three policies over several real applications: (1) does not apply multi-plane operations (2) aggressively issue multi-plane operations that write pages to the same offset at dual planes (3) only wisely

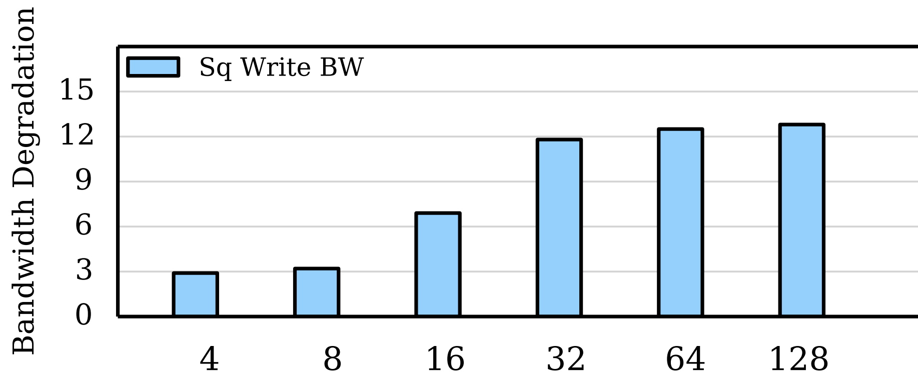


Figure 2.29: Bandwidth degradation factor (Y-axis) scaling with queue depth (X-axis) of concurrent Random Writes.

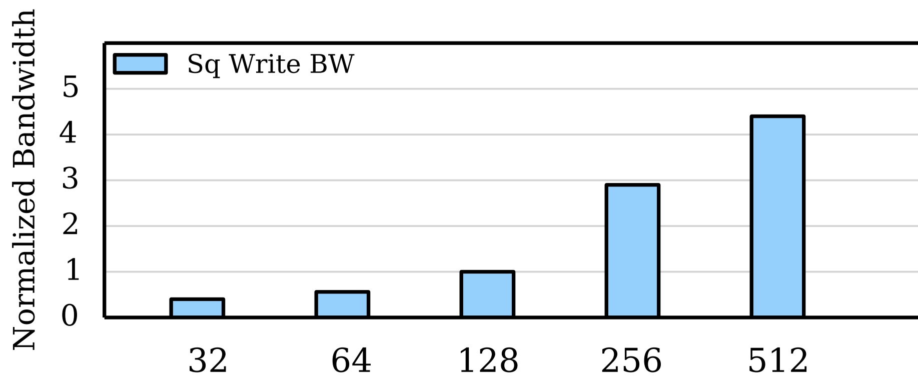


Figure 2.30: Bandwidth degradation factor (Y-axis) scaling with write cache size (X-axis, MB).

apply multi-plane operations when the constraint is met. The SSD is warmed up to steady state before running the application traces. Thee I/O patterns include sequential, random insertions and queries. Figure 2.32 shows the comparison of average request latency for these three policies. We observe that for all applications, especially those having larger request size, the performance with multi-plane operations is generally higher from 2.4% to 11.9%. Surprisingly, the performance improvement with wise multi-plane issuing is slightly better than the aggressive scheme. This is because other levels of parallelism is

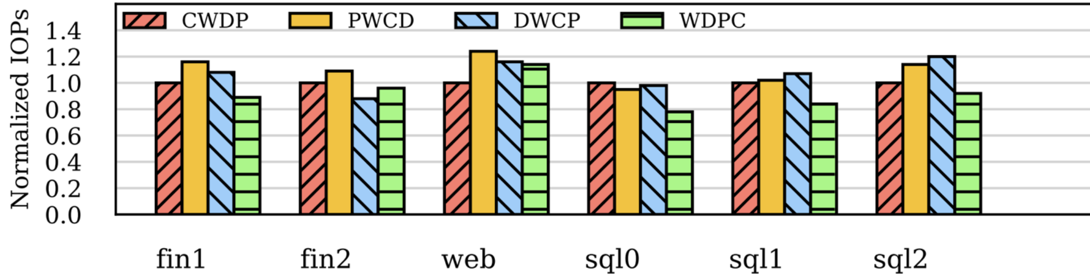


Figure 2.31: Normalized IOPS of various static page allocation schemes.

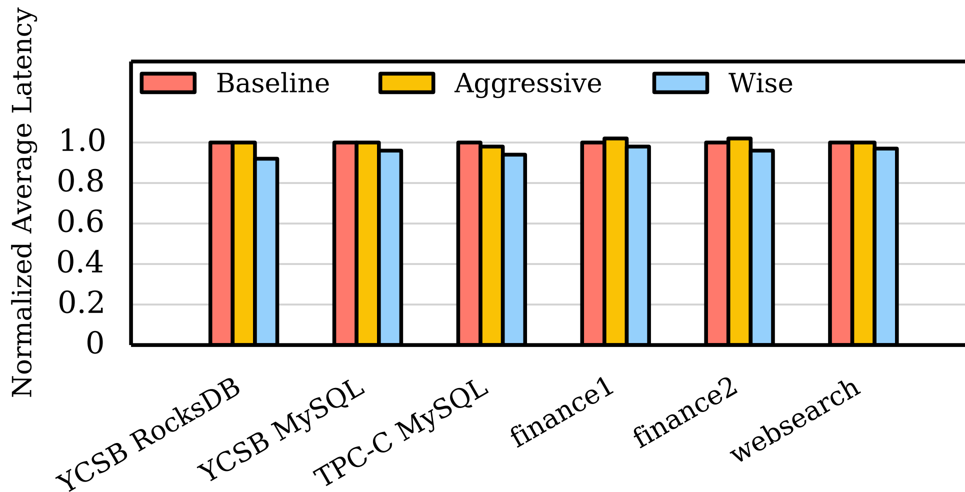


Figure 2.32: Normalized average latency (Y-axis) for various workloads (X-axis, MB) under three plane policies.

sacrificed in order to take advantage of plane-level parallelism.

2.5.10 Impact of Copy-back Operations

Exploiting copy-back operations can improve SSD performance by reducing GC overhead, as discussed in Section 2.1.2. We evaluate the performance benefit of enabling copy-back operations by inspecting WAF (Write Amplification Factor = total_write/host_write,

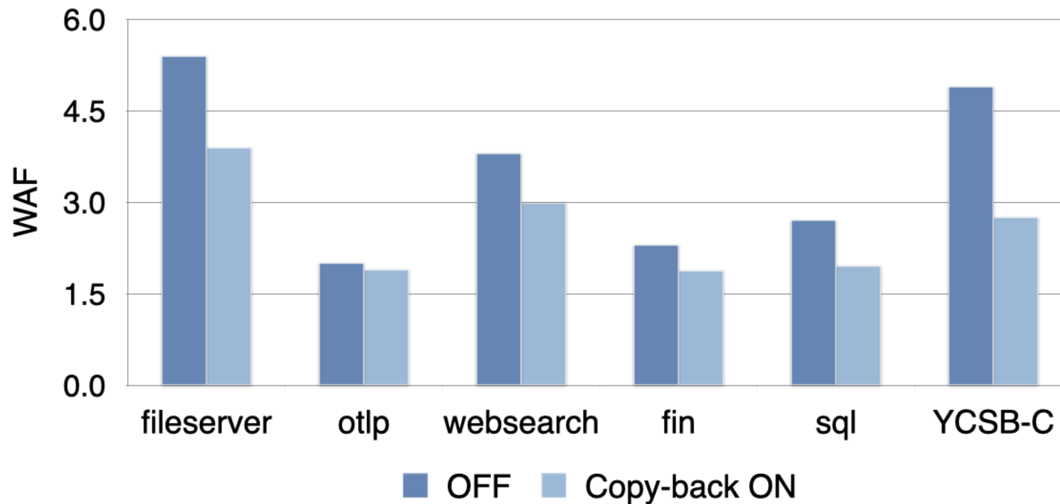


Figure 2.33: Impact on Write Amplification Factor by enabling copy-back operation.

lower is better) as the performance metric (intra-plane copy-back is not counted towards an extra write. Figure 2.33 shows the WAF across six real-world applications with copy-back enabled and disabled. Note lower WAF implies better device lifetime and efficiency.

2.6 Summary

In this chapter, we present an accurate, configurable and detailed SSD simulator. We describe our modeling considerations that take major performance factors into account and provide modern host interface such as NVMe. We hope that this tool can enable studies of device-level designs on application-level performance.

Chapter 3: IceClave: A Trusted Execution Environment for In-Storage Computing

3.1 Background and Motivations

3.1.1 In-storage Computing

In-storage computing has been a promising technique for accelerating data-intensive applications, especially for large-scale data processing and analytics [34, 53, 77, 98, 129, 146, 178, 181, 225, 243]. It moves computation closer to the data stored in the storage devices like flash-based solid-state drives (SSDs), such that it can overcome the I/O bottleneck by significantly reducing the amount of data transferred between the host machine and storage devices. As modern SSDs are employing multiple general-purpose embedded processors and large DRAM in their controllers, it becomes feasible to enable in-storage computing in reality today.

To facilitate the wide adoption of in-storage computing, a variety of frameworks have been proposed. For instance, Willow [225] enables developers to offload code from the host machine to the SSD via RPC protocols, Biscuit [98] develops an in-storage runtime system for supporting multiple in-storage computing tasks following the MapReduce computing model, Summarizer [146] proposed a task scheduler for enabling the computing

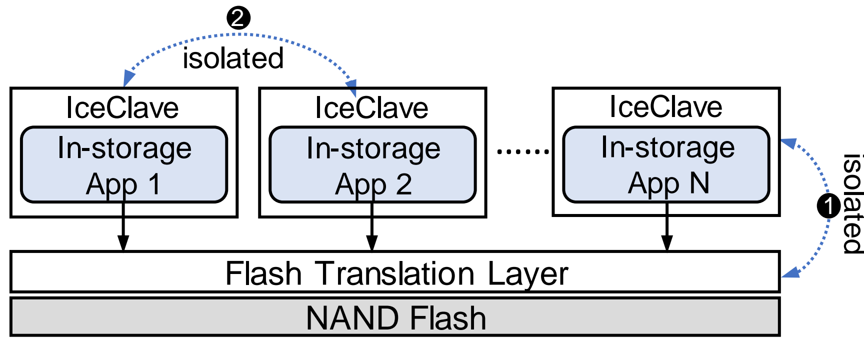


Figure 3.1: IceClave enables in-storage trusted execution environments to achieve security isolation between in-storage programs, FTL, and flash chips.

coordination between the host machine and in-storage processors. All these prior works demonstrate the great potential of in-storage computing for accelerating data processing in data centers. However, most of these studies [45,77,78,146,181,225,244] focus on the performance and programmability of in-storage computing, few of them treat the security as the first citizen in their design and implementation, which imposes great threat to the user data and SSD devices, and further hinders its wide adoption.

3.1.2 In-storage Vulnerabilities

As in-storage processors operate independently from the host machine, and modern SSD controller does not provide a trusted execution environment (TEE) for programs running inside the SSD, they pose severe security threats to user data and flash chips. To be specific, a piece of offloaded (malicious) codes could (1) manipulate the mapping table in the flash translation layer (FTL) to mangle the data management of flash chips, (2) access and destroy data belonging to other applications, and (3) steal and modify the memory of co-located in-storage programs at runtime.

To overcome these security challenges, as developed in state-of-the-art in-storage computing frameworks [98, 225], we can simplify the runtime system by maintaining a copy of the privilege information in the DRAM of SSD controllers (SSD DRAM) and enforcing permission checks for in-storage programs. However, such a solution still suffers from many security vulnerabilities. For example, a malicious offloaded program can exploit memory vulnerabilities such as buffer overflow [70, 238] to enable privilege escalation to access and modify the cached mapping table of FTL in the SSD DRAM; adversaries can steal and modify intermediate data and results generated by the co-located in-storage programs via physical attacks such as cold-boot attack, bus snooping attack, and replay attack [221, 245, 256]. An alternative approach is to adopt Intel's Software Guard Extension (SGX) as a drop-in solution. Unfortunately, modern in-storage processor architectures do not support SGX techniques. And the SGX approach still suffers from significant performance overhead [17, 143, 206, 223, 239], which cannot be afforded by in-storage computing and SSD controllers today.

Therefore, providing a secure, lightweight, and trusted execution environment for in-storage computing is an essential step towards its widespread adoption. Ideally, we wish to enjoy the performance benefits of in-storage computing, while enforcing the security isolation between in-storage programs, the core FTL functions, and physical flash chips, as demonstrated in Figure 3.1.

When specific code is offloaded to in-storage processors, a copy of the privilege information is transferred and maintained in the DRAM of the SSD controller. Such a solution is developed under the assumption that the offloaded code has already known the address and size of the accessed data in advance. However, adversaries can exploit in-

storage software and firmware vulnerabilities such as buffer overflows, and bus-snooping attack to achieve privilege escalation. After that, they can conduct various further attacks. We present them in the following.

- A malicious user can manipulate the intermediate data and output generated by in-storage programs via both software and physical attacks, causing incorrect computing results.
- A malicious program can intercept FTL functions like GC and wear leveling in the SSD and mangle the flash management. This would cause data loss or device destroyed.
- malicious user can steal user data stored in flash chips via physical attacks like bus snooping attack, when in-storage programs load data from flash chips to SSD DRAM.

To defend against these attacks, an alternative solution is to develop an OS or hypervisor for in-storage computing. However, due to the limited resources in the SSD controller, running a full-fledged OS can introduce significant overheads to the SSD and increase the attack surface, due to its large codebase. Moreover, these techniques are not sufficient to defend against the aforementioned attacks such as board-level physical attacks. As we move compute closer to storage devices, it is highly desirable to have a lightweight execution environment for this non-traditional computing paradigm.

Since SSDs were designed with the assumption they are hardware isolated from the host and purely used as storage rather than computing, modern computing systems

do not provide secure runtime environment for in-storage computing. A straightforward approach is to adopt the SGX-like solutions [17, 73]. However, this requires significant hardware changes and even replacement of storage processors available in modern SSDs. As SGX was developed as a generic framework for host machines, it is hard to achieve optimal performance for in-storage programs.

ARM processors available in modern SSD controllers provide the TrustZone technique [28] to enable the creation of secure world and normal world, however, it is unclear whether and how it can be utilized for in-storage computing, and how it should work seamlessly together with the hardware components in SSD controllers. Therefore, as we move computation to the SSD controller, we have to rethink their system design with considering the security as the first citizen, and develop TEEs for in-storage programs.

3.1.3 Existing Security Framework

To defend applications from malicious systems software, a variety of trusted hardware devices have been developed. A typical example is Intel Software Guard Extensions (SGX) [36], which can create trusted execution environments for applications to run on untrusted operating systems, even if the OS, hypervisor or BIOS are compromised. Intel SGX offers hardware-based memory encryption that isolates specific application code and data in memory, and allows user-level code to allocate private regions of memory, called enclaves, which are designed to be protected from processes running at higher privilege levels. The protected application is loaded into an enclave where its code and data is measured, and once the application's code and data is loaded into an enclave, all external

software access is prevented. Enclaves are trusted execution environments provided by SGX to applications. Enclave code and data reside in a region of protected physical memory called the enclave page cache (EPC). While cache-resident, enclave code and data are guarded by CPU access controls. When moved to DRAM, data in EPC pages is protected at the granularity of cache lines. An on-chip memory encryption engine (MEE) encrypts and decrypts cache lines in the EPC written to and fetched from DRAM. Non-enclave code cannot access enclave memory, but enclave code can access untrusted DRAM outside the EPC directly, e.g., to pass function call parameters and results. It is the responsibility of the enclave code, however, to verify the integrity of all untrusted data. For Intel Skylake CPUs, the EPC size is between 64 MB and 128 MB, which may not host an application requiring more memory. To support enclave applications with more memory, SGX provides a paging mechanism for swapping pages between the EPC and untrusted DRAM: the system software uses privileged instructions to cause the hardware to copy a page into an encrypted buffer in DRAM outside of the EPC. Before reusing a shared EPC page, the system software must flush the TLB entry according to the hardware-enforced protocol. SGX incurs performance overhead when executing enclave code: (1) Privileged instructions cannot be executed within the enclave, so the thread must leave the enclave before a system call. Migrating such an enclave is costly. For security reasons, you need to perform a number of checks and updates, including sanitizing the TLB. Memory-based enclave arguments also need to be copied between trusted and untrusted memory. (2) Since the MEE needs to encrypt and decrypt the cache line, the enclave code also pays a penalty for memory writes and cache misses. (3) Applications whose memory requirements exceed the EPC size will need to swap pages between the

EPC and the unprotected DRAM. EPC pages are costly because they must be encrypted to protect their integrity before they can be copied to external DRAM. To prevent address translation attacks, the eviction protocol suspends all enclave threads and empties the TLB. Because of the enabled security isolation, the SGX technique is being extended or customized to support various computing platforms [20, 29, 69, 147, 198, 234, 249] and applications [33, 143, 206, 223]. The hardware devices with TPM [18] serve the similar purpose of security isolation by utilizing hardware support for attestation available in commodity processors from AMD and Intel [182, 183, 230]. As for ARM processors that are used in mobile computing platform and device controllers, they offer TrustZone that enables users to create secure world execution environment isolated from OS [47, 117, 222]. Trustzone established the concept of protection domains called secure world and normal world. Both secure world and normal world are completely hardware isolated and granted uneven privileges, with non-secure software prevented from directly accessing secure world resources [28, 117]. Specifically, by limiting the operating system to operate within the boundaries of the normal world, critical applications can reside inside the secure world without the need to depend on the OS for protection. Unfortunately, none of these trusted hardware devices can be directly applied to in-storage computing, and defend against physical attacks. For instance, Guardat [249] and Pesos [147] allow users to specify security policies in storage devices to protect data confidentiality and integrity, the most recent work ShieldStore [143] and Speicher [33] applied SGX to key-value stores. However, none of them can protect the execution runtime of in-storage programs. In this work, we utilize the ARM processors available in SSD controllers and develop specific trusted execution environments for in-storage applications for security isolation

with minimal hardware cost.

3.1.4 Secure Memory Design

Securing system memory from physical attacks is important for building trusted environments. One of the major security issue is that, the processor communicates with main memory in plaintext, the data stored in the main memory is also in plaintext. Attackers can easily steal sensitive data and modify memory contents. Memory encryption techniques can protect computation privacy from passive attacks by encrypting and decrypting code and data as it moves on and off the processor chip. Memory integrity verification protects code and data integrity, where attackers may attempt to modify memory contents in the main memory or communication channel, by computing and verifying Message Authentication Codes (MACs) as code and data moves on and off the processor chip.

Direct Encryption Early memory encryption schemes [235, 236] directly utilize a block cipher such as Advanced Encryption Standard (AES) to generate plaintext or ciphertext on a memory block when it is read from or written to memory. However, the overhead is high. If the block access experiences a cache miss, the block must first be fetched on chip before it can be decrypted. The latency of decryption adds up to the memory access latency, resulting in execution time overheads of 17% on average [235]. In addition, there is a security vulnerability that can be exploited by attackers because different blocks holding the same data will produce the same ciphertext. Attackers may be able to infer the data block distribution or use this leaked information to compare texts.

Counter Mode Encryption To address performance and security concerns of direct

encryption, researchers have proposed Counter Mode Encryption [165, 261]. The major difference is that counter mode encryption introduces an additional variable One Time Pad (OTP) into the encryption/decryption process. As shown in Figure 3.3, a plaintext cacheline P is encrypted through an XOR with an OTP, producing the encrypted string ciphertext C. Decryption is performed through an XOR of C with the OTP. If the length of plaintext differs from OTP length, P can be padded to facilitate the XOR operation. The OTP is a secret string generated by an AES block cipher with a per-line counter as input. The counter is incremented on each cacheline write to ensure temporal uniqueness. These counters are stored in the memory, and they are cached on-chip to avoid extra memory accesses for the counter. In this way, counter mode encryption allows the OTP to be pre-computed in parallel with the data access. To reduce memory traffic of counters, researchers propose a split counter design [221,237,261] that encapsulates a large number of counters in a cacheline. This is achieved by encoding the counter value as a concatenation of a large major counter and a smaller minor counter, as shown in Figure 3.4. To avoid counter duplication on a minor counter overflow, the major counter is incremented and all the minor counters in the cacheline are reset. This requires additional memory reads and writes to re-encrypt all the data cachelines associated with the minor counters.

Memory Integrity Verification To prevent data tampering by attackers, the signature of each cacheline content called MAC is stored in the memory. On every cacheline access, the MAC is accessed along with the data and the memory controller verifies its integrity by recomputing the MAC using the data and the counter, to ensure no tampering has occurred. To counter replay attacks (e.g., an adversary can replace a tuple of Data, MAC, Counter in memory with older values without detection), researchers propose integrity

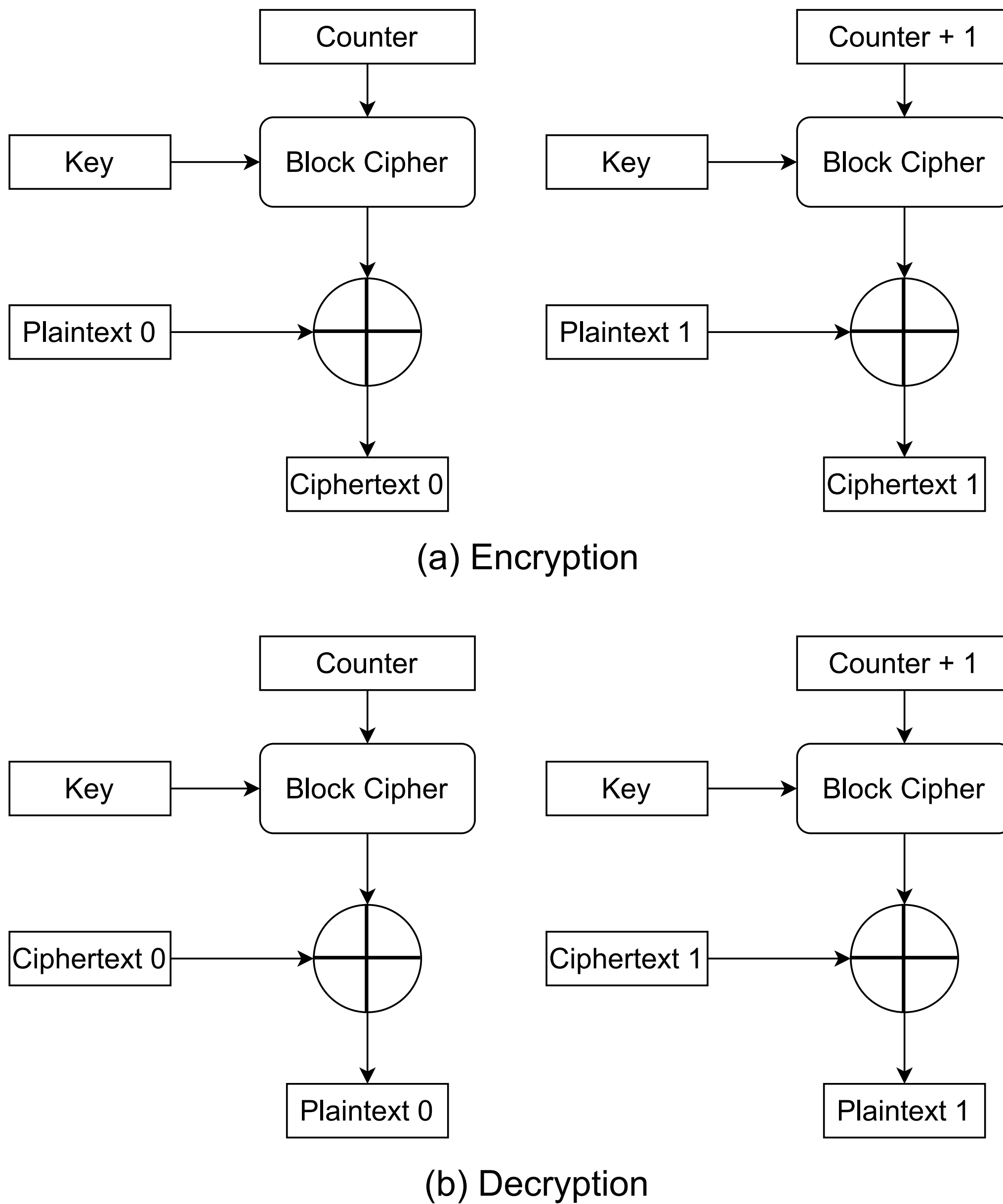


Figure 3.2: Process of encryption and decryption in counter mode encryption.

trees that store multiple levels of MACs in memory, with each level ensuring the integrity of the level below [235,261]. Each level is smaller than the level below, with the root small enough to be securely stored on-chip [221, 227, 261]. When a cache line is written back to memory, the merkle tree needs to update all the nodes on the path from the data block

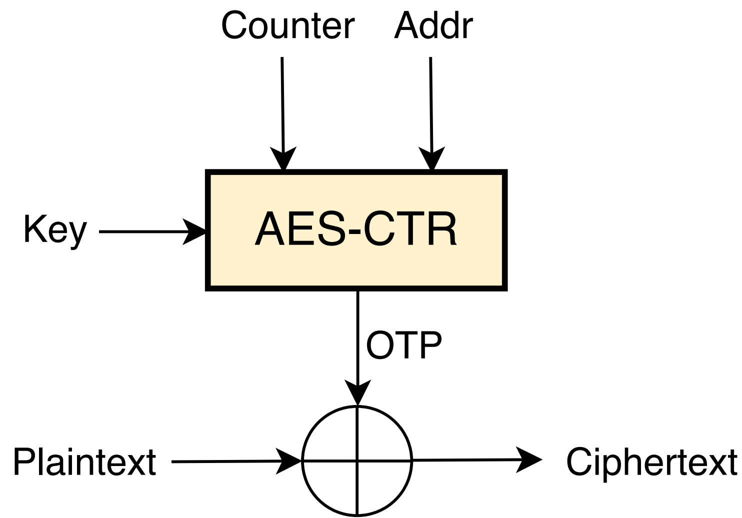


Figure 3.3: Process of encryption and decryption in counter mode encryption.

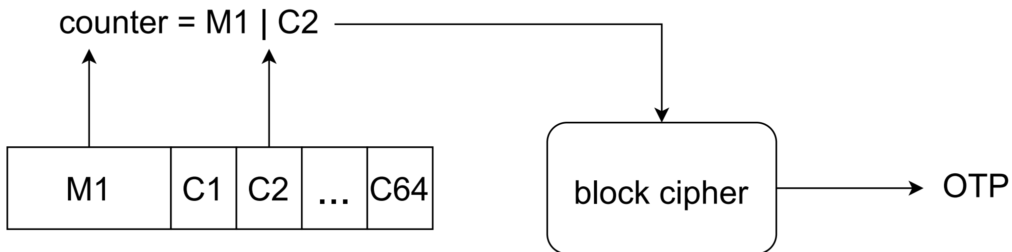


Figure 3.4: Diagram of Split Counter.

to the root. When a memory block is fetched, its integrity can be checked by verifying its chain of MAC values up to the root. Since the on-chip root MAC contains information about every block in the physical memory, an attacker cannot modify or replay any value in memory. Figure 3.5 illustrates a typical integrity tree Bonsai Merkle Tree for integrity verification.

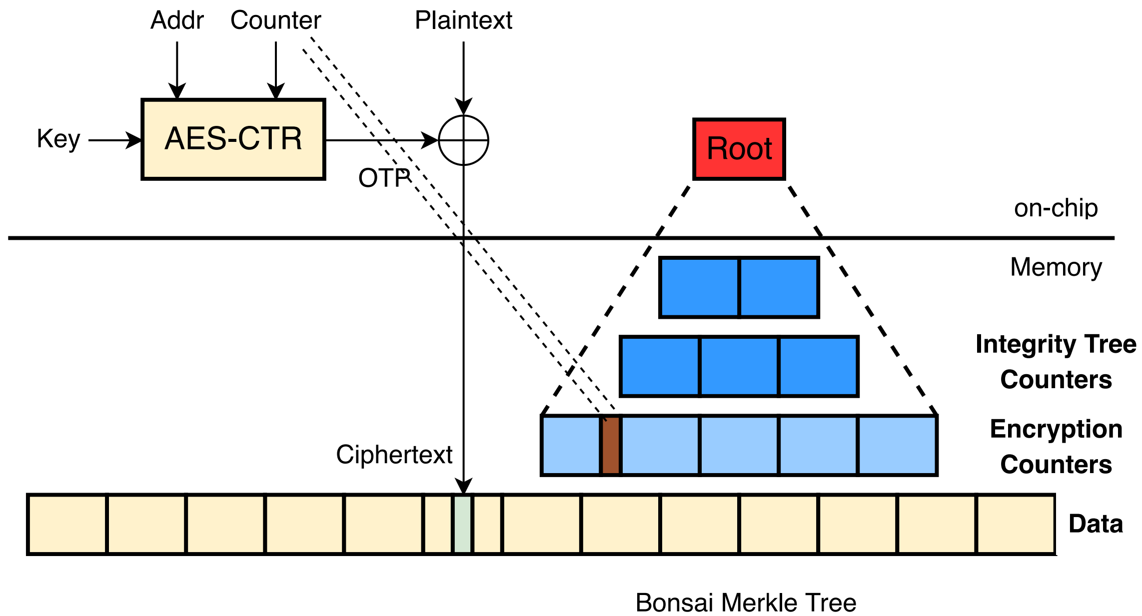


Figure 3.5: Using Bonsai Merkle Tree as integrity tree.

SSD Encryption and Stream Ciphers Some commodity SSD controllers include data encryption hardware. An SSD that contains data encryption hardware within its controller is known as a self-encrypting drive (SED). In the controller, data encryption hardware typically employs AES encryption, which performs multiple rounds of substitutions and permutations to the unencrypted data in order to encrypt it. In an SED, the controller contains hardware that generates the AES keys for each round, and performs the substitutions and permutations to encrypt or decrypt the data using dedicated hardware. The execution of modern stream ciphers usually consist of two stages: the initialization stage and the keystream generation stage. In the initialization phase, the stream cipher first loads the secret key and initialization vector (IV) into an internal state and then updates the

internal state by iteratively calling an updating function for sufficiently many rounds so that the secret key and IV are thoroughly mixed. In the keystream generation phase, the stream cipher generates the pseudo-random keystream digits from the highly diffused internal state by calling an output function while further updates the internal state with the updating function.

3.2 Design Goal

To this end, we present IceClave, a trusted execution environment for in-storage computing. Unlike generic TEE solutions such as SGX [17], IceClave is designed specifically for modern SSD controllers and in-storage programs, with considering the unique flash properties and in-storage workload characteristic. With ensuring the security isolation, IceClave includes (1) a new memory protection scheme to reduce the context switch overhead incurred by flash address translations; (2) an optimization technique for securing in-storage DRAM for in-storage programs by taking advantage of the fact that most in-storage applications are read intensive; (3) a stream cipher engine for securing data transfers between storage processors and flash chips, with low performance overhead and energy consumption; and (4) a runtime system for managing the life cycle of in-storage TEEs.

Specifically, to achieve the security isolation between the in-storage program and the FTL, we extend the TrustZone of ARM processors available in a majority of SSD controllers. IceClave executes core FTL functions such as garbage collection and wear leveling in the secure world, and runs the offloaded programs in the normal world, such

that offloaded programs cannot intervene the flash management. To protect the mapping table of FTL with low overhead, we introduce a protected memory region in the normal world, and place the address mapping table in it to avoid context switches for flash address translation. Therefore, only protected FTL functions can update the mapping table, and offloaded in-storage programs can only read it for address translation with enforced permission checks. In order to achieve the security isolation between in-storage programs, we build in-storage TEEs to host the offloaded programs, and enforce data encryption and memory integrity checks in both data communication and processing for in-storage programs. Since most in-storage computing workloads are read intensive, IceClave mainly needs to conduct the integrity check for the intermediate data and results generated by in-storage programs, which does not introduce much performance overhead to the in-storage program execution. To secure flash pages read by in-storage programs running in the in-storage TEE, we also integrate a lightweight stream cipher engine into the SSD controller with minimum hardware cost.

We implement IceClave with a full system simulator gem5 [91], and integrate an SSD simulator SimpleSSD [96] and memory simulator USIMM [57] into it for supporting secure in-storage computing. We also develop a system prototype to verify the core functions of IceClave with a real-world OpenSSD Cosmos+ FPGA board [281]. We use a variety of data-intensive applications that include transactional databases to evaluate the efficiency of IceClave. Compared to state-of-the-art in-storage computing approaches, IceClave introduces only 7.6% performance overhead to the in-storage runtime, while adding minimal area and energy overhead to the SSD controller. Our evaluation also demonstrates that IceClave can maintain the performance benefit of in-storage computing

by delivering 2.31× better performance on average than the conventional host-based computing approach.

3.3 Threat Model

In this work, we target the multitenancy where multiple application instances operate in the shared SSD. Following the threat models for cloud computing today [36, 69, 281], we assume the cloud computing platform has provided a secure channel for end users to offload their programs to the shared SSD. The related code-offloading techniques, such as secure RPC and libraries [58, 98, 225], have been deployed in cloud platforms [11, 137, 281]. However, an offloaded program can include (hidden) malicious code.

As for in-storage computing, we trust SSD vendors, such as Samsung SmartSSDs [14] and ScaleFlux Computational Storage [12, 53], who enable the execution of offloaded programs. We assume hardware vendors do not intentionally implant backdoor or malicious programs in their devices. However, as we deploy those computational SSDs in shared platforms (e.g., public cloud), we do not trust platform operators who could initiate board-level physical attacks such as bus-snooping and man-in-the-middle attacks, or exploit the host machine to steal or destroy data stored in SSDs. Similar to the threat model for Intel SGX, we exclude software side-channel attacks such as cache timing, page table sidechannel attacks [192], and speculative attacks [145] from the threat model for the TEE created by IceClave, since many of these attack approaches are cumbersome in reality [17].

We rely on the Error-Correction Code (ECC) available in flash controllers [99, 247] for ensuring the integrity of flash pages. To defend against attacks from cloud computing platforms or malicious host OS, users are usually encouraged to encrypt their data before storing them in the shared SSD. However, their data would still be leaked during the in-storage computing procedure. Therefore, we have a more conservative design for achieving the security goals of IceClave.

We believe our threat model is realistic. First, as system-wide shared resource, SSDs have been widely used by multiple applications. Existing in-storage computing frameworks have enabled end users to offload their programs into the SSD. Second, once program is offloaded to the SSD, the in-storage program will escape the control of the host OS and initiate attacks in the new execution environment. Third, our threat model considers the potential physical attacks initiated by untrusted platform operators.

To the best of our knowledge, this is the first work that develops a TEE for in-storage computing. It aims to defend against three attacks: (1) the attack against co-located in-storage programs; (2) the attack against the core FTL functions; (3) the potential physical attack against the data loaded from flash chips and generated by in-storage programs.

3.4 Design and Implementation

We present a TEE for in-storage computing with minimal performance and hardware cost. The TEE functionality is built upon ARM Trustzone technology enabled by the embedded ARM processor, on top of this, we propose to extend ARM Trustzone to create secure

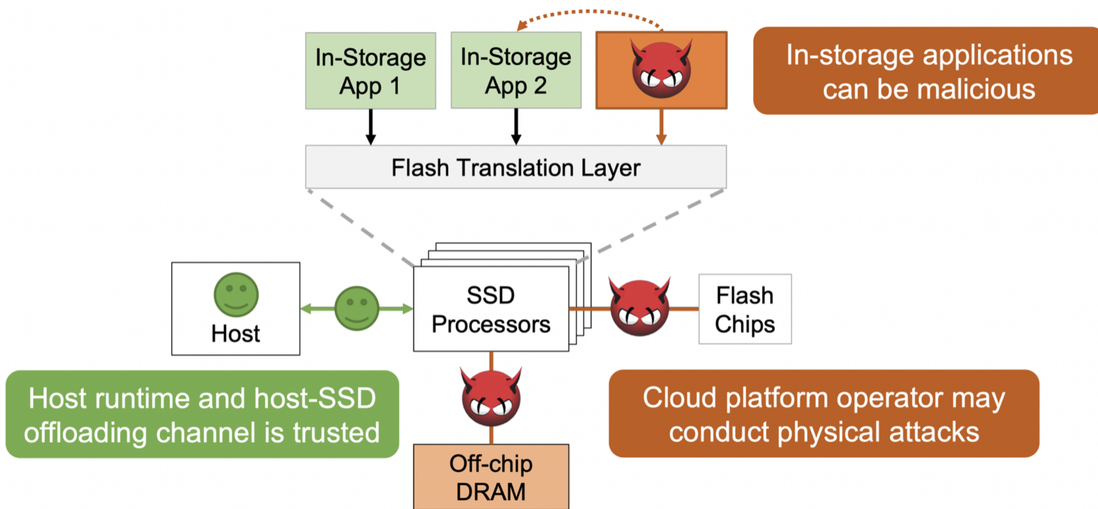


Figure 3.6: Threat model of in-storage computing.

and normal world for security isolation and protection of different entities in FTL, while enabling memory encryption and verification with memory encryption engine (MEE). We show the overview of IceClave architecture in Figure 4.7.

3.4.1 Challenges of Building IceClave

As discussed in Section 3.1, it is desirable to provide trusted execution environment for in-storage computing. However, we have to overcome three challenges.

- First, as SSD is shared by multiple applications, we need to ensure proper security isolation. Specifically, we need to not only enforce security isolation between in-storage applications and FTL functions, but also the isolation between applications and IceClave runtime. (Section 3.4.2).

- Second, to protect data of in-storage programs at runtime, IceClave needs to ensure

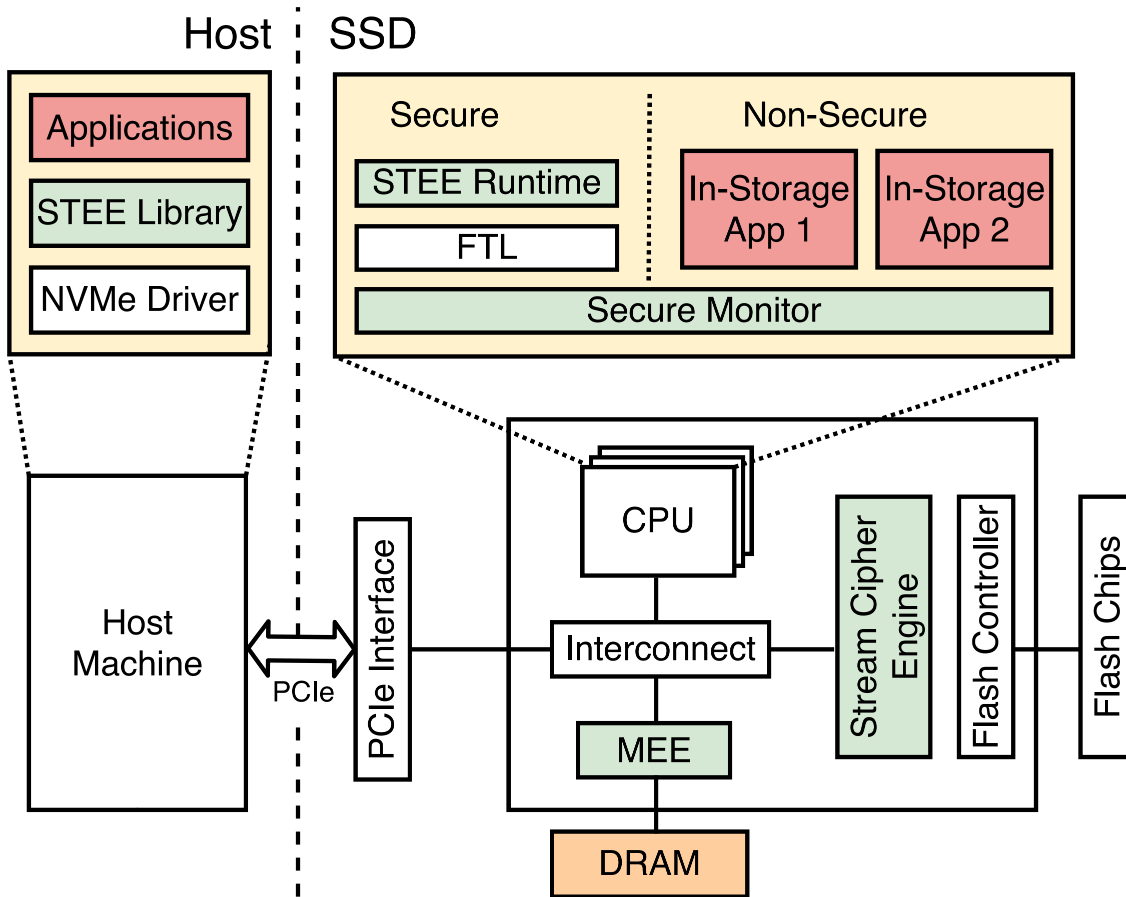


Figure 3.7: Overview of IceClave architecture.

the data security whenever the user data leaves the flash chips.

- Third, SSD controller has limited resource, such as DRAM capacity and processing capacity, therefore, IceClave should be lightweight and not significantly affect the performance of in-storage applications.

3.4.2 Protecting Flash Translation Layer

As FTL manages flash blocks and controls how user data is mapped to each flash block, its protection is crucial. If any malicious in-storage programs gain control over it, they can read, erase, or overwrite data from other users, which could cause severe consequences, such as data loss and leakage. And, IceClave runtime manages how each in-storage application is initialized inside SSD, and maintains their metadata, such as in-storage program identity. If any in-storage program gains access to the metadata, the adversary can easily compromise the security of other in-storage programs.

In order to protect FTL and IceClave runtime from malicious in-storage programs, we need to ensure proper memory protections for different entities in the SSD. Specifically, we have to guarantee offloaded applications cannot access the memory region used by the FTL and IceClave runtime. We also need to ensure offloaded applications cannot access each other's memory regions without proper permissions.

To achieve this, a straightforward way is to use TrustZone to create secure and normal worlds, and then place FTL functions and IceClave runtime in the secure world, and place all in-storage applications in the normal world. However, this will cause significant performance overhead for in-storage applications. This is because when an application accesses a flash page each time, it needs to context switch to the secure world which hosts the FTL and its address mapping table. Similarly, with the SGX-like approach, we can place FTL and in-storage programs in different enclaves, it will also generate significant performance overhead, as we have to switch frequently from one enclave to another.

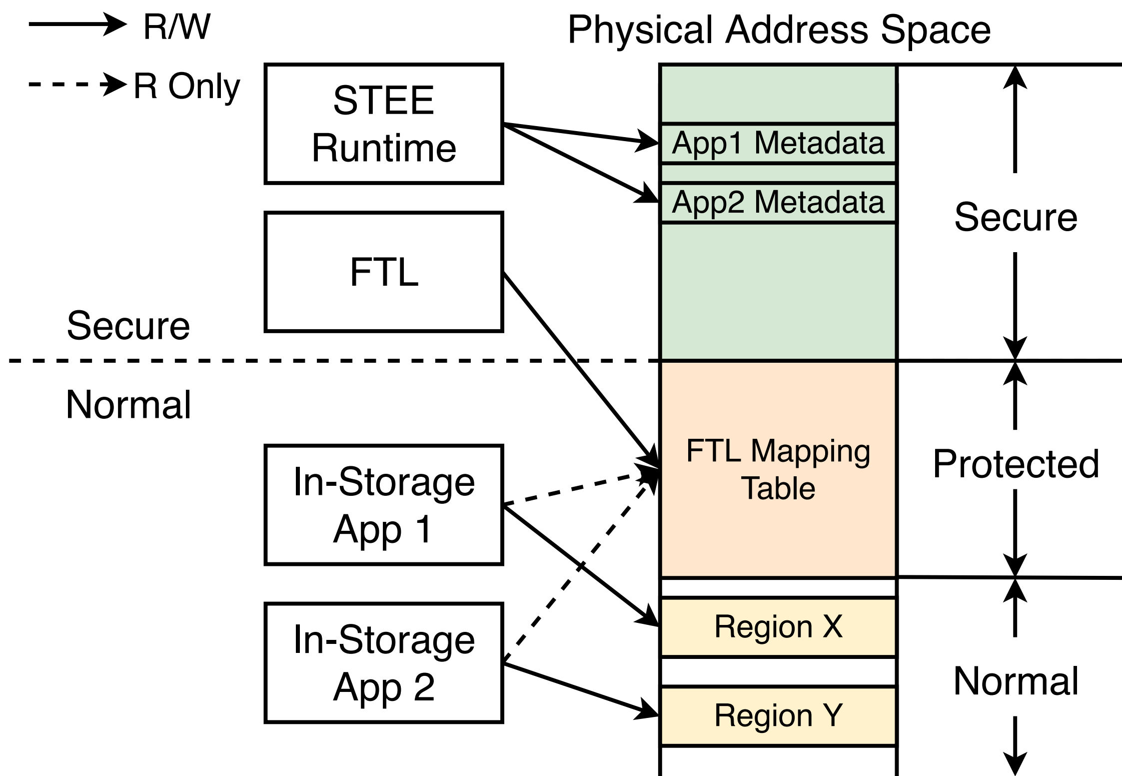


Figure 3.8: Memory protection regions in IceClave.

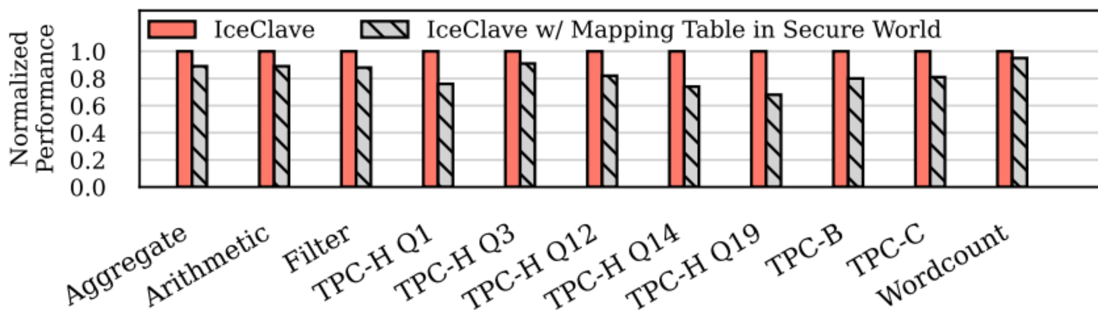
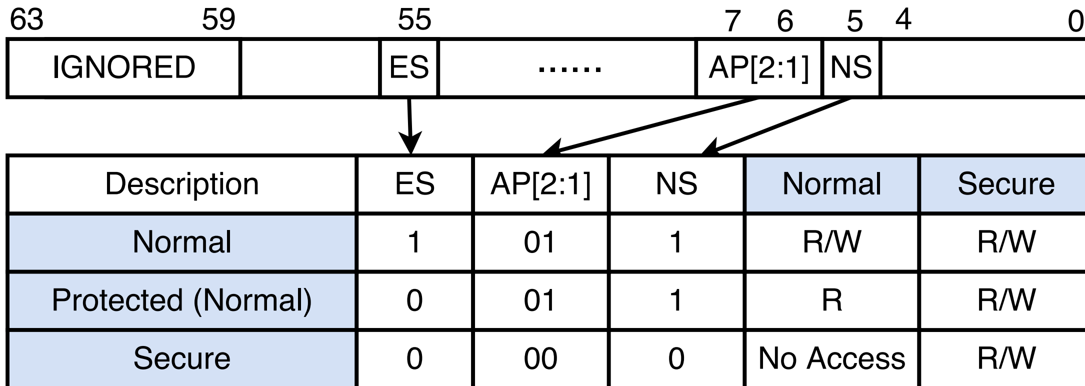


Figure 3.9: Performance comparison between IceClave and IceClave with FTL mapping table in secure world. Performance is normalized to IceClave.

To address this challenge, we partition the entire physical main memory space into three memory regions: normal, protected, and secure by extending TrustZone. We show these memory regions in Figure 3.8. Specifically, we allow FTL and IceClave runtime to execute in the secure world. And they have read/write permission to access the entire memory space. This is necessary, since the core functions of FTL need to manage the address mapping table, and IceClave runtime needs to manage each in-storage application, such as its TEE creation and deletion. We place in-storage applications in the normal world, therefore, they cannot access any code or data regions that belong to the FTL or IceClave runtime.

As for the protected memory region in the normal world, we use it to host the shared address mapping table, such that in-storage applications can only read the mapping table entries for address translation, without paying the context switch overhead. Evaluation shows that this optimization can improve the performance of in-storage applications by 21.6% on average Figure 3.9, compared to the scheme with the FTL mapping table in the secure world.

We demonstrate the details of the memory region attributes in Figure 3.10. Following the MMU specification of ARMv8 [26], we use the non-secure (NS) bit to indicate whether the memory access is performed with secure or normal right. We utilize the access control flags (AP[2:1]) and a reserved bit (ES bit in Figure 6) to create the protected region, in which IceClave gives read-only permission to the normal world and read/write permission to the secure world. It is worth noting that the in-storage memory protection can be easily implemented in an older version of ARM processors [174] by specifying the access control flags AP[2:0] as well as other processors such as RISC-V.



Description	ES	AP[2:1]	NS	Normal	Secure
Normal	1	01	1	R/W	R/W
Protected (Normal)	0	01	1	R	R/W
Secure	0	00	0	No Access	R/W

Figure 3.10: In-storage memory protection in IceClave.

3.4.3 Access Control for In-Storage Programs

Although each in-storage program only has the read access permission when accessing the address mapping table of the FTL, a malicious in-storage program could probe the mapping table entries (e.g., with a brute-force method) that are managing the address translation for the data belongs to other in-storage programs. Henceforth, adversaries can easily access the data of other in-storage programs.

To address this challenge, we extend the address mapping table of FTL. We use the ID bits in each entry (8 bytes per entry) to track the identification of each in-storage TEE, and use them to verify whether an in-storage TEE has the permission to access the mapping table entry or not. The permission checking of flash accesses is performed with a dedicated process. It receives flash access requests from in-storage applications and performs permission checks before issuing the requests to the flash chips. This process has exclusive access to the flash chips in the normal world, which prevents unauthorized

flash accesses from a malicious in-storage program. We use four bits for the ID by default, which introduces small (6.25%) storage cost to the mapping table. IceClave will reuse the ID for newly created TEEs within its runtime, and set the ID bits in the mapping table upon TEE creation (see the details in Section 3.4.6).

Each in-storage application has accesses to only the address mapping table of FTL and the allocated memory space. Accessing to any other memory locations will result in a translation fault in the memory management unit. To further enhance the memory protection for in-storage programs, we also enable memory encryption and verification.

3.4.4 Securing In-Storage DRAM

In-storage programs load data from flash chips to the SSD DRAM for data processing. To conceal the data read from flash chips, IceClave secures the data transfer procedure by encrypting the accessed data before it is transmitted on the internal bus. Modern SSDs have employed dedicated encryption engine [36], however, it is a cryptography co-processor mainly used for full-disk encryption. In this work, we develop a lightweight stream cipher engine in the SSD controller for securing the data transfers from flash chips to the storage processor (see the implementation details in Section 3.6).

Although we enable the data encryption as we transfer data between SSD DRAM and flash controllers, the user data that includes raw data, intermediate data, and produced results could still be leaked at runtime. To address this challenge, IceClave enables both memory encryption and integrity verification, with minimal performance overhead.

Memory Encryption The goal of memory encryption is to protect any data or

Table 3.1: In-storage workload characterization.

Workload	Write Ratio
Arithmetic	5.5e-4
Aggregate	2.1e-4
Filter	1.7e-4
TPC-H Query 1	6.42e-6
TPC-H Query 3	3.70e-3
TPC-H Query 12	2.31e-4
TPC-H Query 14	3.9e-6
TPC-H Query 19	9.15e-6
TPC-B	1.32e-1
TPC-C	9.05e-2

code in a memory access from being leaked. To achieve this, a common approach is to encrypt the cache lines in the processor, when they are being written to memory. The state-of-the-art work usually uses split-counter encryption [30, 239, 262]. It works by encrypting a cache line through an XOR with a pseudo one time pad (OTP), and OTP is generated from encrypting a counter through a block cipher such as AES. The counter is incremented after each write back to guarantee temporal uniqueness. It is encoded as a concatenation of a major counter and a minor counter. When a minor counter overflows, the major counter is incremented, and all other minor counters are reset. The associated memory blocks also need to be re-encrypted. Therefore, such an encryption scheme has significant performance overhead.

This is less of a concern for in-storage computing because of its read-intensive

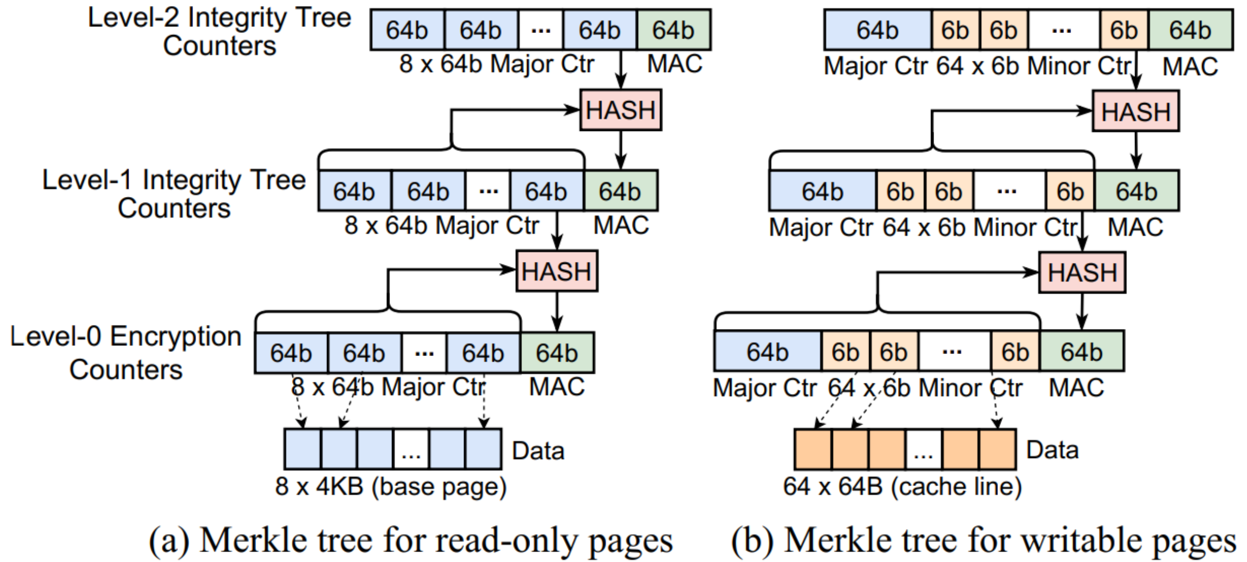


Figure 3.11: Memory encryption and integrity trees in IceClave.

workload characteristic. We conduct a study of typical in-storage applications (see Table 3.4, and profile their number of memory accesses when running them. Our observation is that most of these applications have a trivial portion of memory writes (see Table 3.1). These writes are usually caused by the produced intermediate data of in-storage programs at runtime.

Hybrid-Counter Scheme. Based on this observation, we design a hybrid-counter scheme. The key idea is that we only use major counters for read-only pages, and for writable pages, we apply the traditional split-counter scheme. This is because the minor counters will not change as long as the pages are read-only. In other words, we do not need minor counters for read-only pages. In this case, we can improve the caching performance for in-storage applications by packing more counters (eight for read-only pages) per cache line.

The hybrid-counter scheme maintains two types of counter blocks: split-counter blocks for writable pages, and major counter blocks for read-only pages, as shown in Figure 3.11. We use two integrity trees to store the two types of counter blocks respectively. Although this requires slightly more memory space (0.01% of 4GB DRAM capacity) to store the integrity trees, and needs two processor registers to store the two root message authentication codes (MACs) for memory integrity verification, the hybrid-counter scheme delivers improved performance by 43% on average (see Figure 3.12) for in-storage programs, compared to the current split-counter scheme.

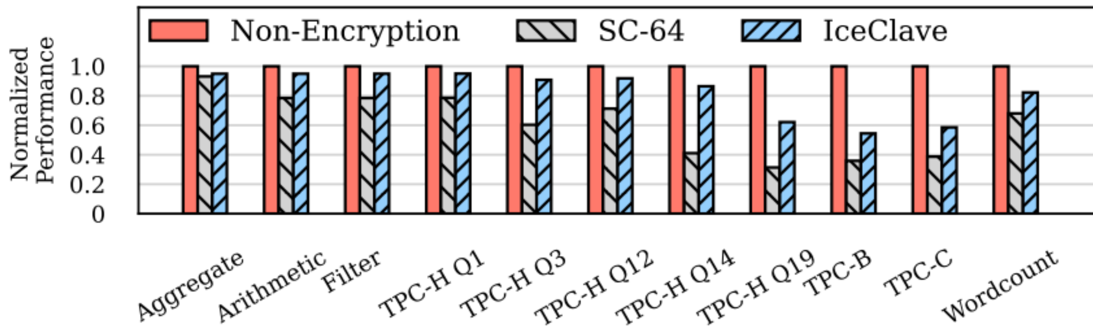


Figure 3.12: Performance comparison of Non-encryption, Split Counters (SC-64), and IceClave. It is normalized to the scheme without memory encryption.

As we use the hybrid-counter scheme in IceClave, we utilize the read/write permission bit in the page table entries to decide which counter blocks should be accessed. We also support dynamic permission changes of each memory page in IceClave. Specifically, for a read-only page, its corresponding counter is stored in the major-counter tree. When the page becomes writable and is updated, its corresponding major counter is incremented and copied to the corresponding entry in the split-counter tree, and also, the minor counters

in that entry are initialized. At the same time, the page is re-encrypted using the new split-counter entry, which will be used for later accesses. When a writable page becomes read-only, its corresponding major counter is incremented and copied back to the major-counter tree. In-storage programs can use the memory protection mechanisms offered by ARM processors (see Figure 3.10) to update the permissions of memory pages. For example, for the memory region used to store the input for in-storage programs, its pages are set to be read only; for the memory region allocated for storing intermediate data, its pages are set to be writable.

Memory Integrity Verification The goal of memory integrity verification is to ensure that the processor receives exactly the same content as it wrote in the memory most recently. To achieve this, a MAC is generated for each memory block by hashing its data and encryption counter. On each memory block access, the MAC is re-computed using the data and encryption counter, and it is compared against the stored MAC, such that any changes on the data or counter can be detected. The integrity tree can also prevent replay attack that could roll back both the data and MAC to their older versions. As shown in Figure 3.11, an integrity tree organizes MACs in a hierarchy, where the parent MAC ensures the integrity of its children MACs. The root of the tree is securely stored in the processor chip. When a cache line is written back to memory, the merkle tree needs to update all the nodes on the path from the data block to the root.

In IceClave, we employ Bonsai Merkle Tree (BMT) [210]. It generates its first-level MAC by hashing counter blocks instead of data blocks. As discussed, IceClave maintains two Merkle trees, but the extra memory cost is negligible, compared to the traditional BMT.

Table 3.2: IceClave API

API in IceClave library	Description
OffloadCode (char* bin, uint* lpa, void* args, uint tid)	Invoke an offloading procedure specified by tid.
GetResult (uint tid, uint64_t* res)	Retrieve results from the offloaded program specified by tid.
API in IceClave runtime	**Description**
CreateTEE (void* config, id_t &eid, tee_t* TEE)	Initiate an TEE and copies specified code into TEE.
SetIDBits (const id_t &eid, uint64_t* lpns)	Set ID bits of the corresponding address mapping table entries.
TerminateTEE (tee_t* TEE)	Terminate the specified TEE, and reclaim resources.
ThrowOutTEE (tee_t* TEE, TEE_MSG* sm)	Abort the execution of the TEE, and return an exception.
ReadMappingEntry (id_t &eid, uint64_t* lpa, uint64_t* ppa)	Request FTL to return the corresponding physical address.

3.4.5 IceClave Runtime

In this section, we discuss how IceClave runtime facilitates the execution of in-storage TEEs. It provides the essential functions for managing in-storage TEEs, such as TEE setup, TEE lifecycle and metadata management, and the interaction with the secure world. IceClave runtime also interacts with IceClave library deployed in the host machine. Note that IceClave library only exposes basic offloading interfaces (e.g., RPC) to end users. This not only reduces the trusted computing base but also simplifies the development of in-storage programs. We list the APIs of IceClave in Table 3.2.

IceClave allows an end user to interact with the SSD using two interfaces: `OffloadCode`

and `GetResult`. Once the program is offloaded to the SSD, IceClave runtime will execute `CreateTEE()` to create a new TEE. At the same time, it will call `SetIDBits()` to set the ID bits (access permission, see 3.4.3) of the corresponding address mapping table entries in FTL with the list of logical page addresses specified by the in-storage program. According to our study on the popular in-storage programs, their code size is 28–528KB. However, for an offloaded program whose size is larger than the available space of SSD DRAM, the TEE creation will fail. During the execution of an in-storage program, `ThrowOutTEE()` will be called to handle program exceptions. IceClave runtime will abort the TEE for these cases that include (1) access control is violated, (2) TEE memory or metadata is corrupted, and (3) in-storage program throws an exception. Once the in-storage program is finished, IceClave runtime will call the `TerminateTEE()` to terminate the TEE.

With the assistance of TrustZone, IceClave supports dynamic memory allocation within each TEE. To avoid memory fragmentation, IceClave will preallocate a large contiguous memory region (16MB by default). Upon TEE deletion, IceClave runtime will release the pre-allocated memory region.

3.4.6 Put It All Together

We illustrate the entire workflow of running an in-storage program with IceClave in Figure 3.13. Similar to existing in-storage computing frameworks [146, 225], IceClave library has a host-to-device communication layer based on PCIe, which allows users to transfer data between the host and SSD. We utilize the secure channel developed in

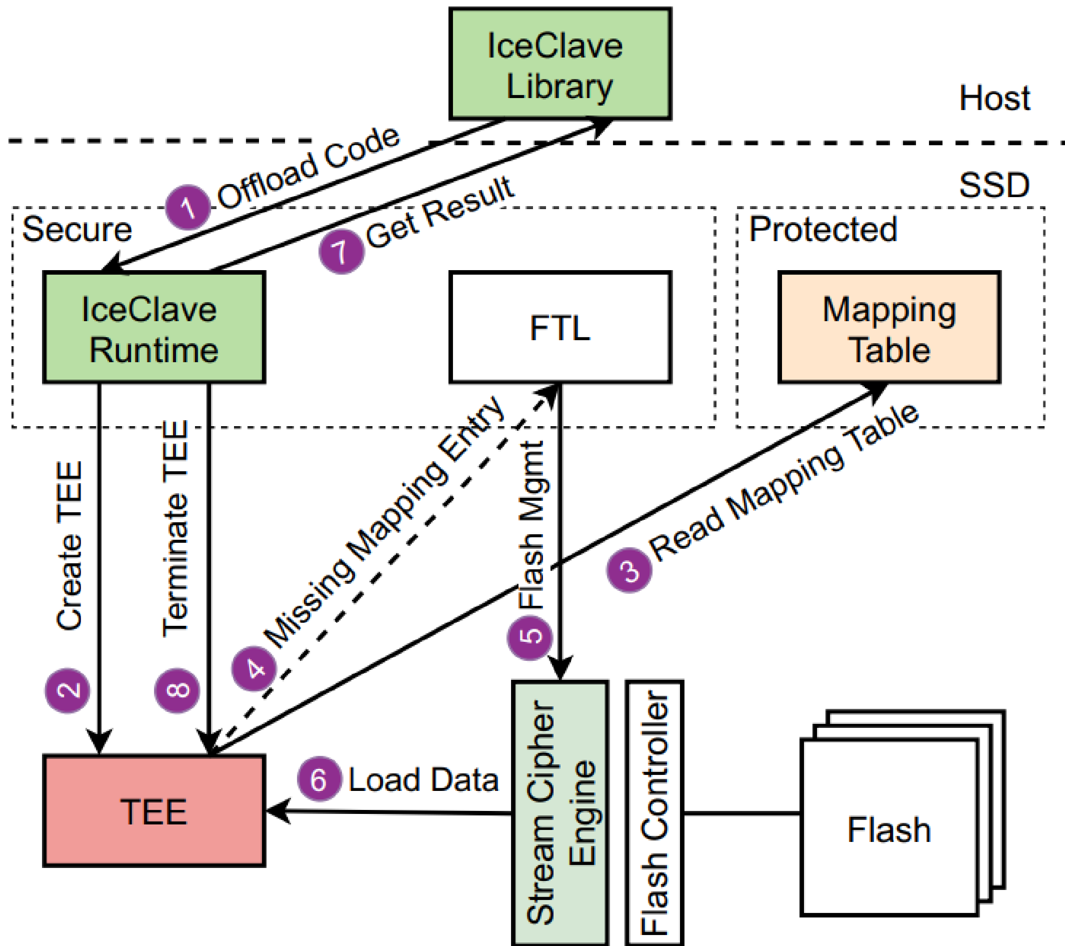


Figure 3.13: The workflow of running in-storage programs with IceClave.

modern cloud computing platforms for interactions between the host and shared SSD. The `OffloadCode` API described in 3.4.5 is called to offload programs. Its parameter bin represents the pre-compiled program in the form of machine code, and `lpa` is a list of Logical Page Addresses (LPAs) of data needed by the offloaded program. It uses task ID (`tid`) as an index for identifying the offloaded procedure.

IceClave runtime will create a new TEE for the offloaded program using `CreateTEE`(②). At creation, IceClave runtime will allocate memory pages from the normal memory

region to the TEE, while the TEE metadata will be initialized and maintained in the secure memory region. IceClave also executes `SetIDBits` to set access permissions in the address mapping table for LPAs. The TEE does not rely on FTL to get physical page addresses, as it can access the mapping table shared in the protected memory region(③).

However, the in-storage program may occasionally encounter cache misses, when a mapping entry for the accessed LPA is not cached in the SSD DRAM. In this case, the TEE has to redirect the address translation request to the FTL via `ReadMappingEntry`(④). This TEE will be paused and switched to the secure world, such that FTL will load the missed mapping table pages(⑤), update the cached mapping table in the protected memory region, and return the PPA to the TEE. To avoid that in-storage programs probing the entire physical space of the SSD, we enforce the access control. Any data on the data path to the TEE is encrypted with the stream cipher engine(⑥). Note that end users are encouraged to encrypt their data to defend against attacks from malicious host OS. In this case, they will send their decryption key to the TEE along with the offloaded program, and decrypt the data at runtime in the TEE.

The TEE will be invoked by IceClave runtime once the in-storage program is readily prepared inside the TEE. IceClave runtime constantly monitors the status of initiated TEEs, secures memory regions, and ensures the mapping table permission guard. Exceptions will be thrown out if any aforementioned integrity is compromised.

During the entire lifecycle of an TEE, the hardware protection mechanisms described in Section 3.4.4 are enforced to prevent physical memory attacks. IceClave will also enforce strong isolation between TEEs and the FTL. The memory protection described in Section 3.4.4 will also be enforced in existing TrustZone memory controller (TZASC) [25].

Once reaching the end of the TEE program, the results are copied into the TEE's metadata region before terminating the TEE and reclaiming used resources(⑧). IceClave will initiate a DMA transfer request to the host using NVMe interrupts, signaling the readiness of results. Results are returned to the host memory via `GetResult`(⑦) provided in IceClave library.

In summary, IceClave can protect in-storage computing from both software and physical attacks with low overhead: (1) it enables memory encryption and verification for SSD DRAM with low overhead; (2) it protects shared FTL without frequent context switches between normal and secure worlds; (3) it protects transferred data from flash chips to SSD DRAM with an efficient stream cipher engine in the SSD controller.

3.4.7 Discussion and Future Work

In this chapter, we exploit TrustZone technique in ARM processors to enable the memory protection between in-storage programs and FTL functions. This is driven by the fact that ARM processors are available in a majority of modern SSD controllers. As device vendors are also considering adopting the open-source RISC-V architecture in their controllers [76, 101, 229], the key idea of IceClave can also be implemented with new type of processors. To be specific, RISC-V defines three levels of privileges, including application level, supervisor level, and machine level [87]. We can map the normal, protected, and secure memory regions (see Section 3.4.2) to different memory regions in RISC-V respectively.

Beyond leveraging the ARM and RISC-V processors to conduct in-storage computing, recent works also deploy hardware accelerators in SSD controllers to speed up in-storage

applications [32, 63, 129, 130, 178, 181, 257]. They are also lacking the support of in-storage TEEs. We wish to extend IceClave to these in-storage hardware accelerators as future work.

As discussed in Section 3.1, similar to existing TEE solutions like Intel SGX for host machines, we exclude software side-channel attacks from our threat model. We wish to investigate the possibility of these attacks in IceClave for in-storage computing. Note our security scheme does not surpass the level provided by ARM TrustZone. If these attacks do exist in real-world in-storage computing scenarios, we would apply memory access obfuscation approaches [46], self-paging mechanism [193], hardware transaction memory [97], and even new specific approaches to defend against them as the future work.

3.5 Implementation Details

Full System Simulator. We implement IceClave with an in-storage computing simulator, which is developed based on the SimpleSSD [96], Gem5 [91], and USIMM [57] simulator. We use the SimpleSSD to simulate a modern SSD and its storage operations. We show the SSD configuration in Table 3.3. To enable in-storage computing in the simulator, we utilize Gem5 to model the out-of-order ARM processor in the SSD controller. We also implement the stream cipher in the integrated simulator to enable the data encryption/decryption as in-storage applications load data from flash chips. We use CACTI 6.5 [187] to estimate

Table 3.3: In-storage computing simulator.

In-Storage Processor	ARM Cortex-A72 1.6GHz [27, 78]
Decoder Width	3 ops
Dispatch/Retire Width	5 ops
L1 I/D Cache	48KB/32KB
L2 Cache	1MB
DRAM in SSD	DDR3 1600 MHz
Capacity	4GB
Organization	1 Channel, 2 Ranks/Channel, 8 Banks/Rank
Timing	t_{RCD}-t_{RAS}-t_{RP}-t_{CL}-t_{WR} = 11-28-11-11-12
Encryption Delay	AES-128: 60ns
SSD	1TB Flash-based SSD
Organization	8 channels, 4 chips/channel, 4 dies/chip
	2 planes/die, 2048 blocks/plane
	512 pages/block, 4KB page
	t_{RD}/t_{WR}=50/300μs
Bandwidth	600 MB/s per channel

its hardware cost, and find that the cipher engine introduces only 1.6% area overhead to a modern SSD controller used for a 1TB Intel DC P4500 SSD.

As IceClave will enable the memory verification in SSD DRAM, we leverage USIMM to simulate the DRAM in the SSD. We implement the Bonsai Merkle Tree (BMT) in the USIMM simulator, and use the split-counter mode [262] as the memory encryption scheme. As discussed in Section 3.4.4, the root of the integrity tree is stored in a secure on-chip register. To enable memory encryption and integrity verification, we enforce that

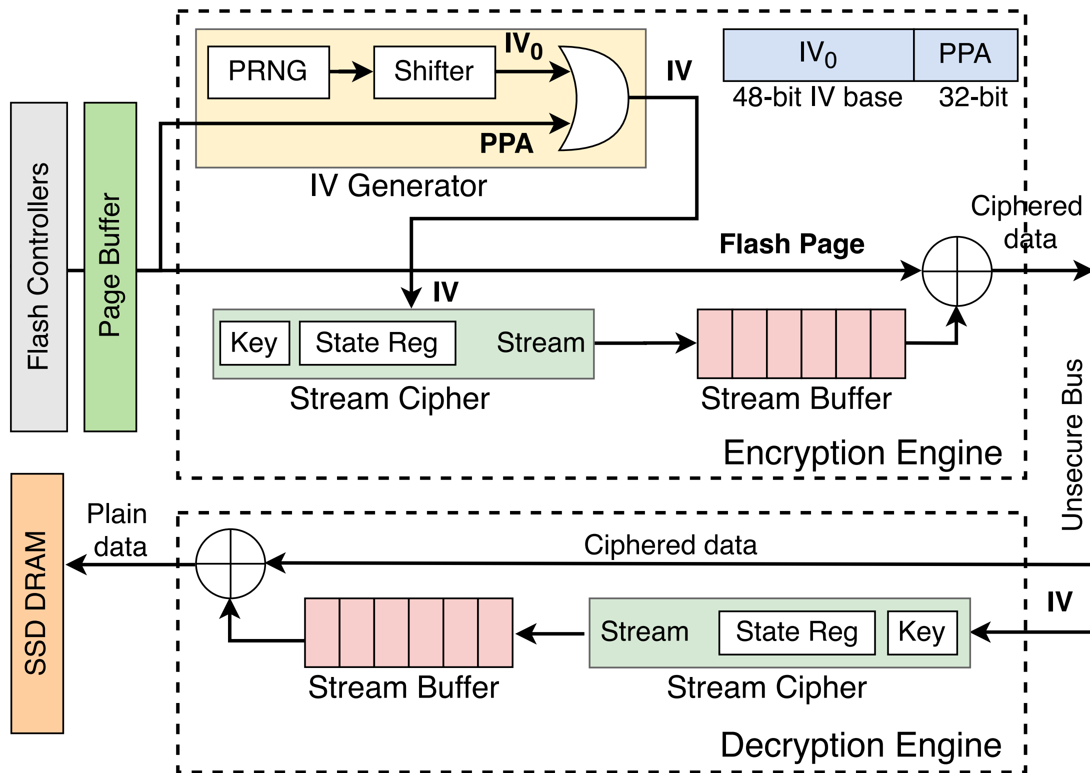


Figure 3.14: Stream cipher engine design in IceClave.

each memory access will trigger the verification and updates of the MAC and integrity tree.

Real System Prototype. To verify the core functions of IceClave, including TEE creation/deletion, FTL, and stream cipher engine, we also implement IceClave with an OpenSSD Cosmos+ FPGA board that has a Dual ARM Cortex-A9 processor. We measure their overheads and show them in Table 3.5. We demonstrate the architecture of the stream cipher engine in Section 3.14. Its key initialization block takes a symmetric key and an arbitrary initialization vector (IV) as the input to initialize the cipher. IceClave keeps the key in a secure register, while the IV can be public. Once initialized, the stream cipher generates

64 keystream bits per cycle. The generated keystream is XORed bitwise with the data read from flash chips to produce the ciphered data. The decipher uses the same key and IV to decode the ciphered data. The IV is constructed with temporally unique random numbers and spatially unique address bits. The orthogonal uniqueness enforces a strong guarantee that the same IV value will not be used twice during a certain period of time. The stream cipher algorithm we used refers to the Trivium [74]. To provide the uniqueness for different flash pages, we compose the IV by concatenating its physical page address (PPA) and the output of a pseudo-random number generator (PRNG).

3.6 Evaluation

To evaluate the performance of our design, we conduct cycle-accurate simulations with workloads from four essential database operations and benchmark suites: TPC-H. These workloads are described in Table 3.4, along with their write ratio. We generate the memory traces for these workloads with PIN [122]: the traces are generated for 1 billion memory accesses after fast-forwarding to the region of interest. These traces are then fed into cycle-accurate memory system simulator USIMM [57]. USIMM parameters are listed in 3.3, simulating an Out-Of-Order ARM processor. Our evaluation demonstrates that: (1) IceClave introduces minimal performance overhead to in-storage workloads while enforcing security isolation in SSD controllers; (2) It has minimal negative impact on the energy efficiency of in-storage computing; (3) IceClave scales in-storage application performance as we increase the internal bandwidth of SSDs; (4) It can benefit various SSD

Table 3.4: In-storage workloads used in our evaluation.

Workload	Description
Arithmetic	Mathematical operations against data records
Aggregation	Aggregate a set of data values with average operation
Filter	Filter a set of data that matches a certain feature
TPC-H Q1	Query pricing summary involving scan
TPC-H Q3	Query shipping priority involving join
TPC-H Q12	Query shipping modes and order priority involving join
TPC-H Q14	Query market response to promotion involving join
TPC-H Q19	Query discounted revenue involving join and aggregate
TPC-B	Queries in a large bank with multiple branches
TPC-C	On-line transactional queries in a warehouse center

devices with different access latencies; (5) IceClave still outperforms conventional host-based computing significantly while offering security isolations, as we vary the in-storage computing capability.

3.6.1 Experimental Setup

We evaluate IceClave with a set of synthetic workloads and real-world applications as shown in Table 4.2. In the synthetic workloads, we use several essential operators in database system, including arithmetic, aggregation, and filter operations. As for the real-world applications, we run real queries from the TPC-H, TPC-B, and TPC-C benchmarks. Specifically, we use TPC-H Query 1, 3, 12, 14, and 19 that include the combination of multiple join and aggregate operations. In all these workloads, we populate their dataset

(tables) to the size of 32GB, and place them across the channels in the SSD.

We compare IceClave with several state-of-the-art solutions. Particularly, we compare IceClave with the Intel SGX available in the host machine, in which we load the data from the SSD to the host memory and conduct the queries in the SGX. For this setting, we use a real server, which has an Intel i7-7700K processor running at 4.2GHz, 16GB DDR4-3600 DRAM, and 1TB Intel DC P4500 SSD. For fair comparison, we follow the specification of the Intel DC P4500 SSD to configure our SSD simulator. We also compare IceClave with state-of-the-art in-storage computing approach that does not provide TEEs for offloaded programs. To summarize, we list them as follows:

- **Host**: in which we load data from the SSD to the host memory, and execute the data queries using host processors. The host machine and SSD setups are described above.
- **Host+SGX**: in which we run data queries within the Intel SGX after loading data from the SSD. The version of the SGX SDK we use in our experiments is 2.5.101.
- **In-Storage Computing (ISC):** in which we run data queries with the ARM processors in the SSD controller, such that we can exploit the high internal bandwidth of the SSD.

We use CACTI 6.5 [187] to estimate energy utilization of the SRAM (including the one used for stream cipher engine) in the 32nm technology node. We use the `itrs-hp` model for SRAMs of the SSD. We assume the DRAM energy is 20-pJ/bit [268]. We use the power consumption of flash page access in the Intel DC P4500 SSD to compute the

energy consumed by flash accesses. We calculate the energy consumption of processor and network-on-chip using McPAT [160].

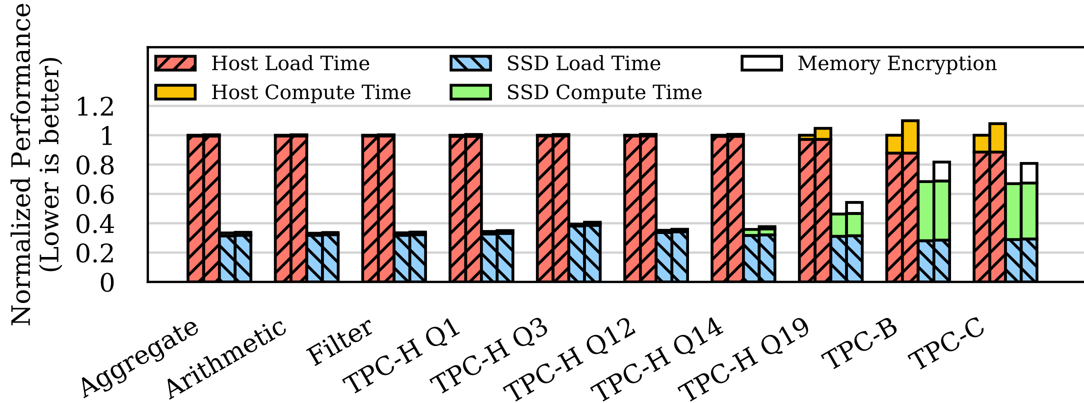


Figure 3.15: Performance comparison of Host, Host+SGX, ISC, and IceClave (from left to right). We show the performance breakdown of each scheme.

3.6.2 Performance of IceClave

We show the normalized performance of running each query benchmark in Figure 3.15. We use the Host as the baseline, in which we run the query workload with the host machine while loading the dataset from the SSD. As shown in Figure 3.15, IceClave outperforms Host and Host+SGX by 2.39× and 2.43× on average, respectively. This shows that IceClave will not compromise the performance benefits of in-storage computing, while ensuring the security isolation inside SSDs. As those data query workloads are bottlenecked by the storage I/O, the Intel SGX on the host machine (Host+SGX) slightly decreases the workload performance. Compared to in-storage computing without security isolation enabled (ISC), IceClave introduces 6.2% performance overhead, due to the

security techniques used in the in-storage TEE.

To further understand the performance behaviors of IceClave, we also demonstrate the performance breakdown in Figure 3.15. As for the Host and Host+SGX schemes, we partition their workflow into two major parts: data load and computing time. As we can see from Figure 3.15, Host+SGX incurs 88.9% extra computing time on average, caused by the SGX running in the host machine. As for the ISC and IceClave schemes, we profile the data load time from flash chips, the computing time with in-storage processors, and the overhead caused by the memory encryption and verification for IceClave. As shown in Figure 3.15, IceClave and ISC take much less time on loading time, as the internal bandwidth of the SSD is higher than its external bandwidth. And they require more time (2.6× on average) to execute the data queries. Compared to ISC, IceClave needs memory encryption and verification for defending against physical attacks. However, IceClave still outperforms Host-based approaches significantly.

Table 3.5: Overhead Source of IceClave.

Overhead Source	**Average Time**
TEE creation	95μs
TEE deletion	58μs
Context switch	3.8μs
Memory encryption	102.6ns
Memory verification	151.2ns

Table 3.6: Extra memory traffic caused by memory encryption and verification when running in-storage workloads.

Workload	Encryption	Integrity Verification
Arithmetic	10.3%	17.3%
Aggregate	9.5%	16.6%
Filter	9.3%	16.4%
TPC-H Query 1	9.9%	15.4%
TPC-H Query 3	22.4%	34.4%
TPC-H Query 12	20.9%	28.5%
TPC-H Query 14	46.7%	64.5%
TPC-H Query 19	69.7%	81.0%
TPC-B	39.6%	53.4%
TPC-C	37.4%	49.6%

3.6.3 Overhead Source in IceClave

We also profile the entire workflow of running an in-storage program with IceClave. We show the critical components of IceClave and their overhead in Table 3.5. As we can see, IceClave takes 95μs and 58μs (measured in real SSD FPGA board) to create and delete an TEE inside SSD, respectively. The overhead of switching the context between secure world and normal world is 3.8μs. As discussed in Section 3.4.2, IceClave has infrequent context switches at runtime, as it places the frequently accessed address mapping table in the protected memory region. The context switch happens mostly because the mapping table is missing in the protected memory region, and IceClave needs to switch to the secure world to fetch the flash pages of the mapping table from flash chips,

and update them in the protected memory region.

Moreover, IceClave incurs much less memory encryption and verification operations, because most in-storage workloads are read intensive (see Section 3.4.4 and Table 3.1). The average execution times of each memory encryption and verification take 102.6 ns and 151.2 ns, respectively. To understand the memory overhead of IceClave, we profile the additional memory accesses (see Table 3.6) incurred by fetching and overflowing counters in the memory encryption and integrity verification. We show the extra memory traffic in percentage in Table 3.6, when comparing to the regular memory traffic without enforcing memory security. Memory encryption and verification increase the memory traffic by 27.6% and 37.7% on average, respectively. However, as in-storage applications are usually I/O bottlenecked, the increased memory traffic does not hurt the performance of in-storage computing significantly. These incurred memory traffic is the source of overhead by our IceClave encryption and verification scheme. As suggested in Section 3.4.4, the OTP encryption mechanism requires re-encrypting the associated memory when the counter overflows, this happens more frequently if an application has a higher write ratio. We also profile the number of flash address translations requested from an TEE, and find that only 0.17% of these address translations are missed in the cached mapping table in the protected memory region. This indicates that in-storage applications do not incur the context switch from the normal world to the secure world frequently, providing the evidence that IceClave is lightweight for in-storage workloads.

3.6.4 Impact on Power Consumption

We now evaluate the energy efficiency of IceClave against Host, Host+SGX, and ISC. We measure the power consumption of the host processors using an open-source tool CPU Energy Meter [232] and *perf* [121]. With the measurement methods described in § 3.6.1, we collect the energy consumption of the entire SSD, including the energy consumed by embedded processors, SSD DRAM, and flash chips. We present the comparison in Figure 3.16. Compared to ISC without security isolation, IceClave introduces 11.8% energy overhead averagely as we run various query workloads.

To further understand the energy consumption of IceClave, we also demonstrate the energy breakdown of all these schemes in Figure 3.16. It includes the energy consumption of ARM processor, SSD DRAM, and flash access. Compared to ISC, the energy overhead of IceClave is mainly caused by the extra operations generated by the memory encryption and verification. The stream cipher engine of IceClave adds trivial energy overhead (0.01% of the energy consumption of flash access).

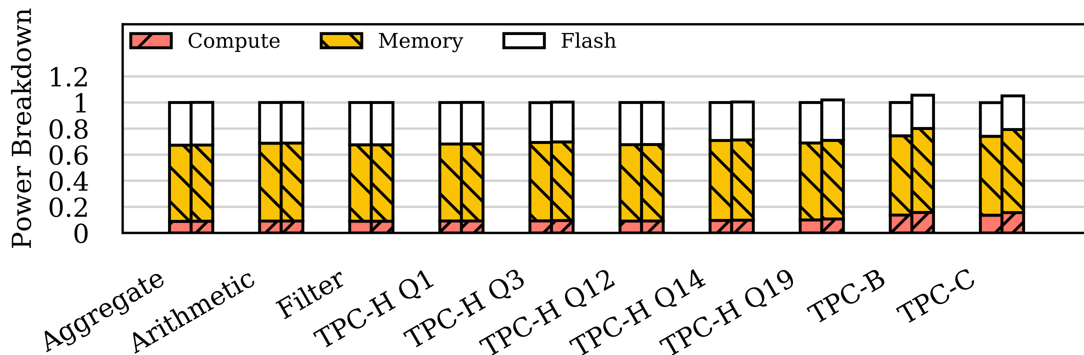


Figure 3.16: Energy overhead of IceClave.

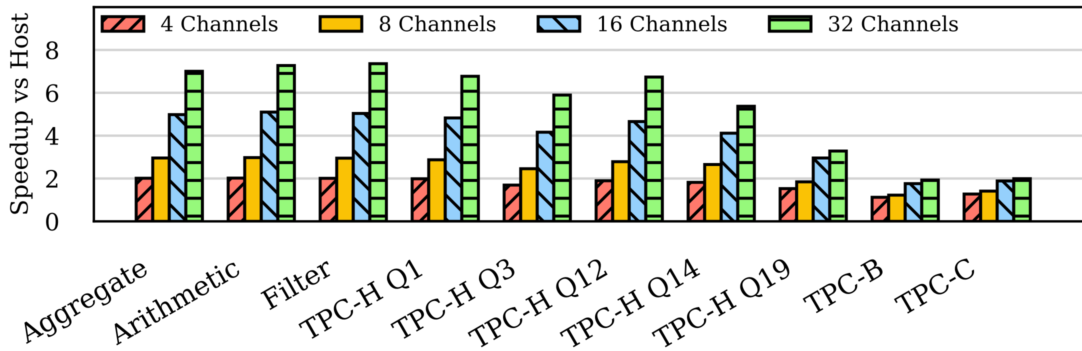


Figure 3.17: IceClave performance (normalized to Host), as we vary the internal SSD bandwidth by using different number of channels.

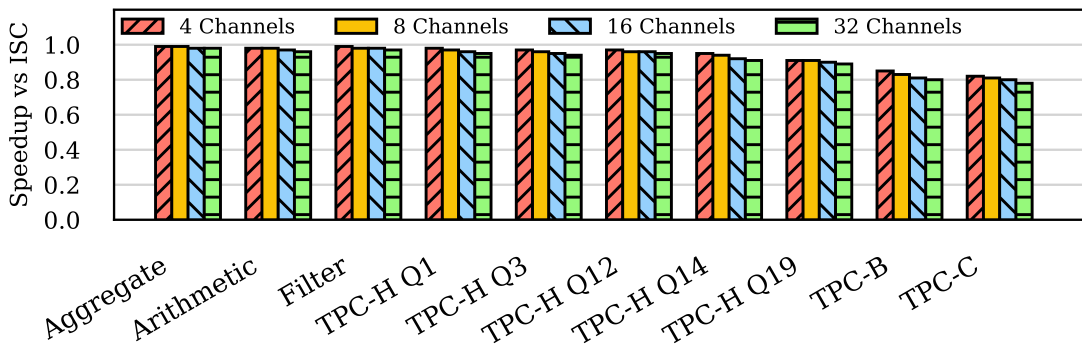


Figure 3.18: IceClave performance (normalzied to ISC), as we vary the internal SSD bandwidth by using different number of channels.

3.6.5 Impact of SSD Bandwidth

In this section, we evaluate the performance sensitivity of IceClave, as we change different SSD parameters. We first vary the internal bandwidth of the SSD by changing

the number of flash channels from 4 to 32. With this, the aggregated internal I/O bandwidth grows linearly as we have more channels, while the external bandwidth is capped by the PCIe bandwidth [43, 146]. We compare IceClave with the host-based computing approach (Host), and present the normalized speedup in Figure 3.17. As we can see, for each data query workload, the performance benefit of IceClave scales significantly as we increase the number of channels. To be specific, IceClave speeds up the performance by 1.7–5.4× over Host, demonstrating that IceClave has negligible negative impact on the performance scalability of in-storage computing. As for in-storage workloads that involve more complicated query operations such as TPC-B and TPC-C, increasing the internal bandwidth brings 1.2–2.1× performance speedup, and for others such as the synthetic workloads and TPC-H, we obtain more performance benefits (1.9-6.2×). It is worth noting that IceClave achieves even more performance benefits than Host+SGX, because SGX introduces extra overhead (see Figure 3.15 and Section 3.6.2). And most importantly, IceClave enables trusted execution environments for in-storage programs.

As we vary the internal SSD bandwidth, we also compare IceClave with ISC. As shown in Figure 3.18, IceClave decreases the application performance by up to 20.3% (6.6% on average), compared to ISC. And the addiontal overhead is slightly increased as we increase the number of channels for complicated data queries like TPC-C. This is mainly due to the increased overhead of memory encryption and integrity verification. However, IceClave offers a trusted execution envrionment for offloaded programs, making us believe it is worth the effort.

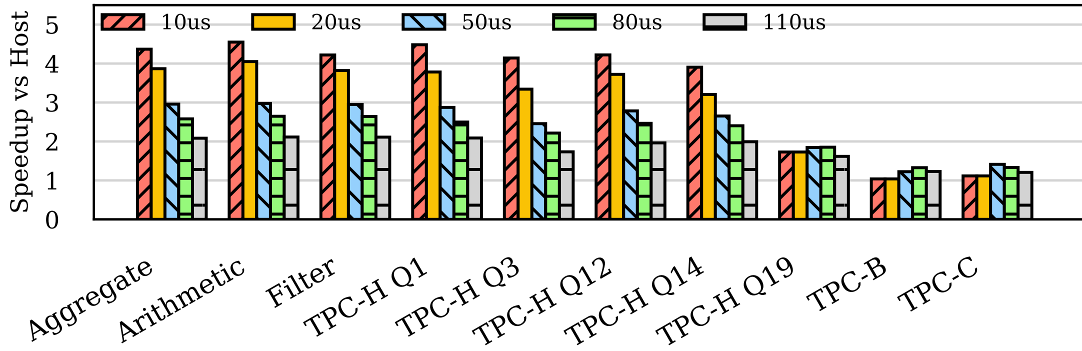


Figure 3.19: IceClave performance as we vary the latency of accessing flash pages.

3.6.6 Impact of Data Access Latency

To understand how the data access latency affects the performance of IceClave, we vary the read latency of accessing a flash page from 10μs, modeling an ultra-low latency NVMe SSD [8], to 110μs, modeling a commodity TLC-based SSD [184]. We keep the write latency as 300μs, this is because most in-storage workloads are read-intensive, which involve few write operations to the dataset stored in the SSD. We use 8 channels in the SSD. We present the experimental results in Figure 3.19. As shown in Figure 3.19, compared to the host-based computing approach that is bottlenecked by the external PCIe bandwidth, IceClave delivers performance benefit (1.8–3.4$\times$) for various SSD devices with different access latencies. For TPC-B, TPC-C, and TPC-H Q19 query workloads that require more computing resource for hash join operations, IceClave offers less performance benefit for the SSD with ultra-low latency, because the processors in the host machine provides more powerful computing resource.

3.6.7 Impact of Computing Capability

As we exploit the on-board embedded processors to run in-storage applications, it will be interesting to understand how the in-storage computing capability affects the efficiency of IceClave. We vary this SSD parameter by using various model of embedded processor. We use our in-storage computing simulator to simulate the representative out-of-order (OoO) ARM processor A72, and the in-order processor A53 with varying frequencies. We compare IceClave with the baseline Host that has an Intel i7-7770K processor running at 4.2GHz.

We show the normalized speedup in Figure 3.20. As expected, the performance of IceClave drops 13.6–33.4% as we decrease the CPU frequency of ARM processors. And an OoO processor A72 performs slightly better than the in-order processor A53 with the same CPU frequency. This demonstrates that IceClave can work with different type of ARM processors and deliver reasonable performance benefits.

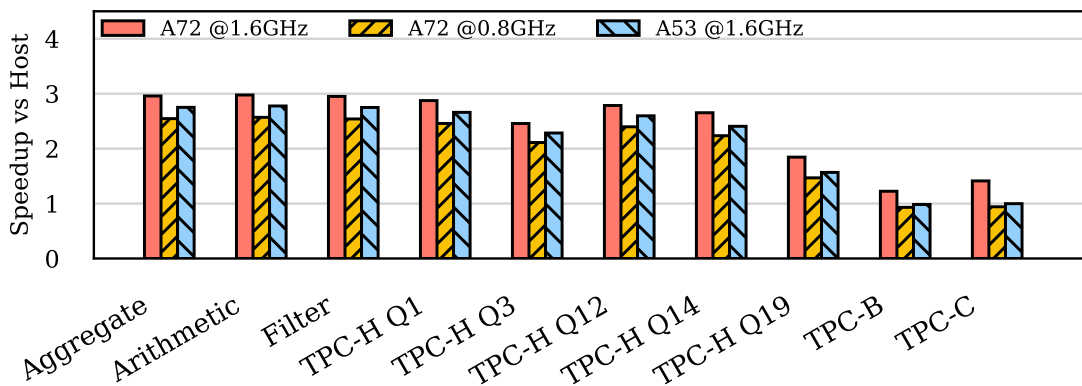


Figure 3.20: IceClave performance as we vary the in-storage computing capability.

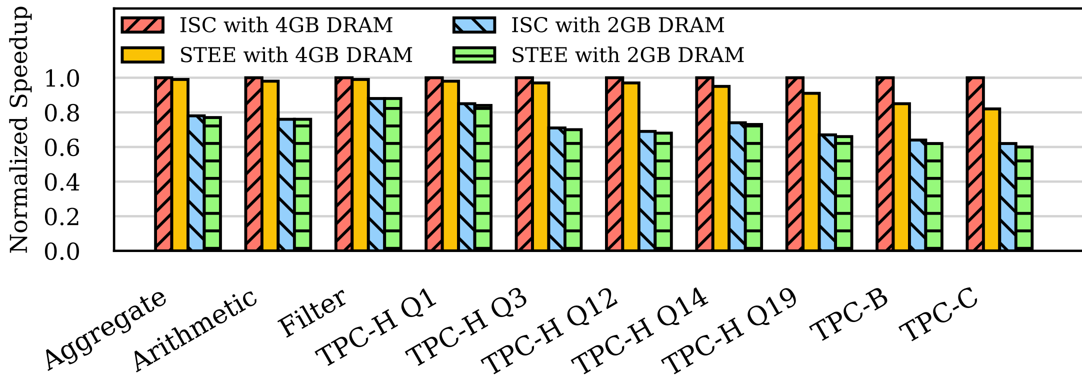


Figure 3.21: IceClave performance as we vary the SSD DRAM size.

3.6.8 Impact of DRAM Capacity in SSD

To evaluate the impact of the SSD DRAM capacity on the IceClave performance, we change the SSD DRAM size from 4GB to 2GB while using the same configurations as described in Table 3.3. We present the experimental results in Figure 3.21. As we decrease the SSD DRAM capacity, the performance of ISC drops by 13.6%–61.3%, as it has limited memory space to store its data set. The performance of IceClave follows the same trend. However, compared to ISC, IceClave still introduces minimal performance overhead.

3.6.9 Performance of Multi-tenant IceClave

To further evaluate the efficiency of IceClave, we run multiple IceClave instances concurrently, and each instance hosts one of the in-storage workloads as described in

Table 4. We compare the application performance with the case of running each in-storage application independently without co-locating with other instances. As shown in Figure 3.22, when we collocate the TPC-C instance with other workloads, the performance of in-storage applications is decreased by 6.1–15.7%. As we increase the number of collocated instances (see Figure 3.23), their performance drops by 21.4% on average. This is mainly caused by (1) the computational interference between the collocated IceClave instances, and (2) the increased cache misses (up to 8.7%) of the cached mapping table in the protected memory region. However, these in-storage programs still perform better than host-based approaches that are constrained by the external I/O bandwidth of SSDs.

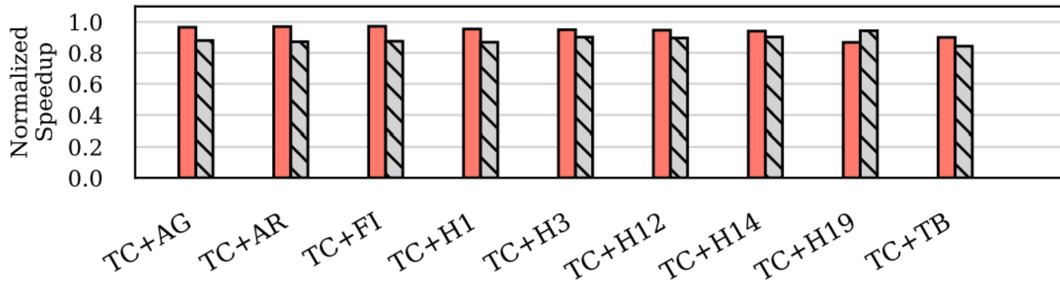


Figure 3.22: IceClave performance as we run two in-storage applications concurrently.

3.7 Conclusion

Due to the lack of TEE support in SSD controllers, adversaries can intervene offloaded programs, manage flash management, steal and destroy user data. To this end, we develop IceClave, a lightweight TEE which enables security isolation between in-storage programs

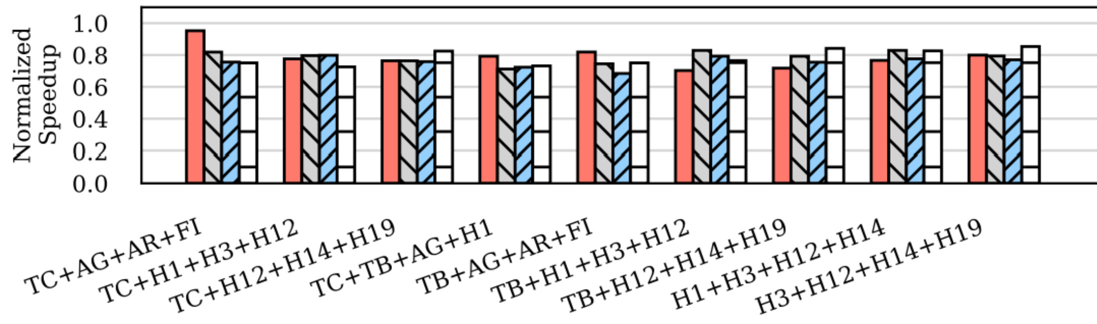


Figure 3.23: IceClave performance as we run four in-storage applications concurrently.

and flash management. IceClave can also defend against physical attacks with minimal hardware cost.

Chapter 4: Zoned FTL: Achieve Resource Isolation via Hardware Virtualization

Aiming at a higher service rate per unit cost, cloud service providers are deploying multiple tenants in the form of Virtual Machines (VMs) and lightweight containers [68, 179, 194] on the same physical server. All these tenants are conceptually isolated but still share underlying hardware resources such as processors, memory, storage, and network resources [52,191]. Though state-of-the-art virtualization technologies can manage hardware resources to provide performance isolation to some extent [92,157,280], providing consistent I/O performance with SLA compliance remains a challenging task [35, 105, 138]. This is because either the cloud providers or the users can communicate the performance objectives efficiently between applications, the storage software stack and underlying devices. While enjoying the simplicity of using a simple read and write device, users often experience varying I/O performance in a naturally shared cloud environment [81, 275]. The wide gap between the decades-old block interface and growingly complicated NVMe devices lead to wide performance fluctuations and SLA violation from time to time, and fade many software-level attempts to achieve predictable performance [44, 82, 161, 273, 286].

In cloud environments, users expect as much as they pay, they expect a certain level of performance stated in the contract with cloud providers [23, 67, 176]. This tier-based

pricing applies to storage resources as well. Some cloud users may choose a high-capacity and low-performance storage tier, while others are pursuing fast and predictable I/O response time for their I/O intensive applications, such as Online Transaction Processing (OLTP), non-relational database and other big-data workloads [71, 102, 248]. Current Cloud providers, like Amazon EC2 [23], have already deployed a resource management framework to enable the pre-defined levels of hardware resources and performance that users require. Compared to processor, memory and networking, I/O SLA, often evaluated in IOPS (throughput) and end-to-end latency (in the chapter, our main concern is the device access latency, which often is the most significant piece of the overall latency.), is the most challenging to achieve due to the thick storage software stack and dynamic sharing nature of a black-block storage device. Resource contention includes but not limited to: (1) The necessary internal management activities performed by the SSD firmware such as garbage collection can block user I/Os, leading to a delay in an order of milliseconds; (2) shared storage resource such as flash chips, data bus and internal buffer tend to incur contentions on the critical I/O path, blocked I/Os suffer significant delays. These tail-latency events slow down overall system performance and amounts to unreliable service. This is because applications are unaware of data layout in the underlying device and the device is unaware of the SLA of I/O requests. The conventional storage system stack lacks full isolation between applications sharing the storage device.

Many software-based isolation techniques implements I/O schedulers to slice SSD's throughput to tenants, such as the cgroup module and multiple block I/O schedulers in Linux kernel. However, they cannot manage resource conflicts at device-level, as data from different tenants are usually striped across the entire SSD to achieve high

utilization and throughput [263]. In this scenario, the overall I/O throughput and latency can fluctuate violently depending on the type (e.g. read, write, or sync) and pattern (e.g., sequential or random) of I/O requests. This means that the I/O pattern of a tenant can possibly disrupt the SLA of other tenants. Device-level isolation techniques such as multi-streamed SSD [65, 264, 284] can separate tenants to different SSD sections, which alleviate performance interference inside the device to some extent. However, different tenants still compete many resources on the critical I/O path, performance interference can still be severe under data-intensive scenarios. New interfaces have emerged to give people more control on SSD resources [10, 39, 216, 253]. The Open-Channel [42] interface exposes raw parallelism to the upper layers – the host is aware of the device layout, and can manipulate data placement and I/O scheduling [202], enabling a new opportunity for implementing performance isolation. However, it does not gain much commercial popularity as it requires exclusive hardware and significant changes in applications and interfaces for satisfying deployment. Therefore, providing a holistic, lightweight yet effective performance isolation scheme that works with I/O virtualization is crucial in a cloud computing environment. Ideally, we wish to deliver maximum hardware throughput while enforcing resource isolation in shared flash memories, management protocols, and I/O queues.

In this chapter, we quantify the impact of workload and storage device characteristics on I/O latency predictability. Not surprisingly, we find widespread and severe unpredictability across the board. By analyzing the latency breakdown, We point out that the overwhelming source of unpredictability is the low-latency SSD. SSD is inherently unpredictable due to the interference of other I/O flows and background activities such as garbage collection

(GC) and internal buffer flush. To address its unpredictable nature, we propose Zoned FTL (ZFTL), which combines host-level modifications and device-level designs to eliminate interference from co-located tenants by enforcing isolation in I/O queues and internal SSD resources. Specifically, we expose an NVMe SSD as multiple instances, and attach each virtualized service to a physical instance through existed virtualization techniques and slightly modified NVMe semantics. We implement the device-level design following these principles:

- I/O requests from each instance should not experience major resource conflicts, especially in performance-critical shared resources.
- Each instance should manage its flash resources, I/O scheduling and caching policy independently, depending on its I/O pattern.
- Users should still observe a conventional block interface to use whereas the heavy-lifting is handled by the storage device.

We implement and evaluate ZFTL designs on the full-system emulator FEMU [159]. We investigate the effectiveness, and tradeoff between predictability and performance of ZFTL on various configurations of data-intensive workloads. Our evaluation demonstrates that ZFTL barely impacts the raw performance while delivering up to 1.51x better throughput and reduce the 99th percentile latency by up to 79.4% in a multi-tenancy environment.

4.1 Background and Motivations

4.1.1 Multi-queue NVMe SSD and NVMe Semantics

Emerging NVMe SSDs are driving the evolution of cloud computing systems. This ongoing replacement has enabled much higher parallelism and a lower end-to-end latency, bridging the widening gap between storage and memory. Before the emergence of NVMe, SSDs are using the AHCI and SATA protocol, which was originally designed for conventional Hard Disk Drives (HDD). Compared with the HDD, SSD has a lower delay and higher performance, AHCI has been unable to keep pace with the development of SSD performance, thus has become a bottleneck restricting SSD performance. SSD needs PCIe and NVMe to replace SATA and ACHI to unleash its performance potential. For example, the I/O throughput has rocketed from thousands of IOPS several years ago, to several millions of IOPS [4, 24] today. This sharp increase is primarily due to a large fleet of flash chips packed in a single SSD, the multi-queue I/O architecture and wider data path (PCIe) between the SSD and the host. The multi-queue I/O architecture is a high-performance and scalable design that can handle millions of IOPS on multi-core systems. As shown in Figure 4.1. The processor core issues an I/O request to its associated software dispatch queue through the multi-queue block device driver, known as blk-mq in Linux [111]. A dispatch queue is associated with a pair of NVMe submission and completion queue. Note the submission and completion queue are located in the host memory. When issuing an I/O request to the NVMe SSD, the block driver dispatches the request into the corresponding NVMe submission queue. The host does not send commands directly to SSD, but notifies

the SSD by writing the doorbell register located in the NVMe device. There are two doorbell registers for each NVMe queue, recording the head and tail of submission and completion queue, respectively. When the NVMe device finished processing the command, it writes the completion result into the corresponding completion queue and raises an interrupt to the host. In the current Linux kernel, the NVMe driver creates a submission queue and completion queue per core to avoid locking and ensure process integrity.

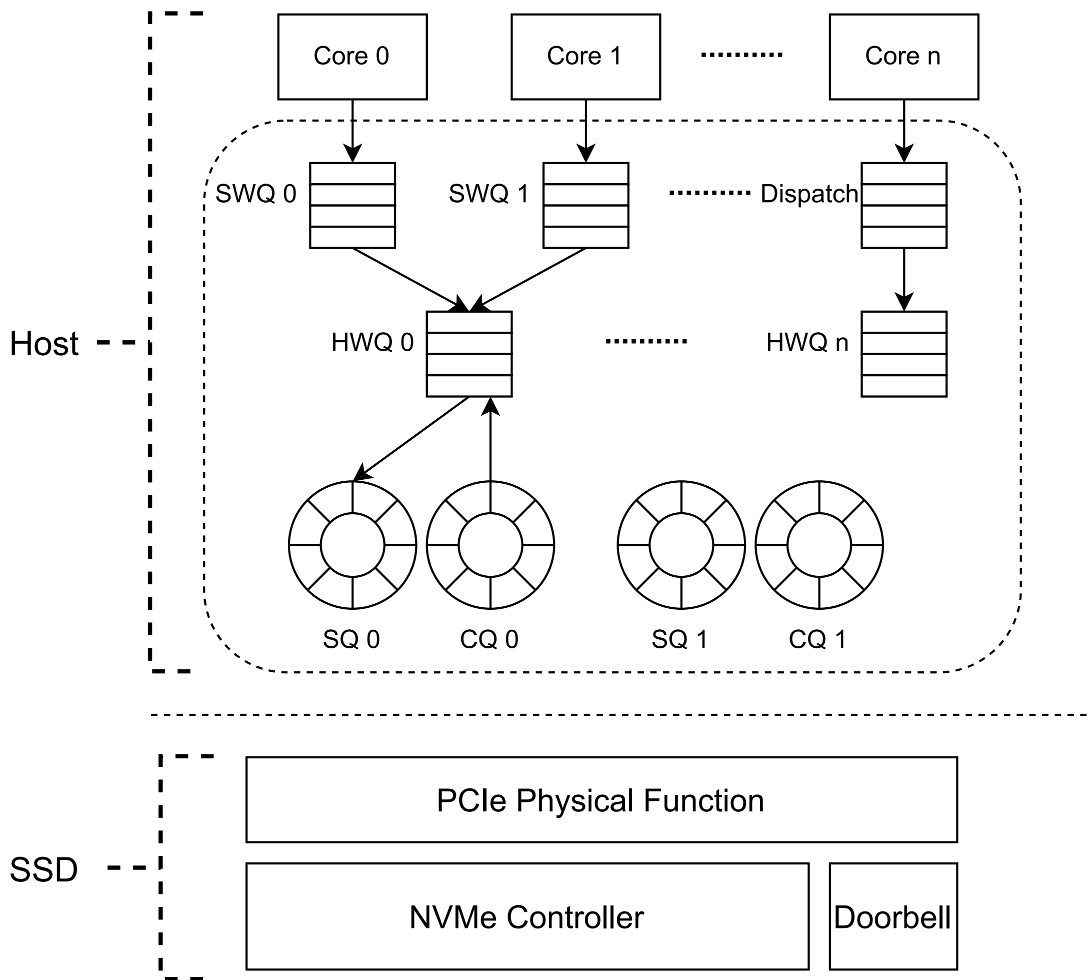


Figure 4.1: The Multiqueue I/O architecture.

Figure 4.2 shows that a NVMe SSD can be virtualized into multiple virtual instances

(functions) through Single-Root I/O Virtualization (SR-IOV) [136]. A virtual function (VF) is assigned to a VM. A VM access a VF through a multi-queue block device driver as if it accesses a real device. We leverage this existed feature to bridge The semantic gap between virtualized services and the multi-queue SSD with queue pairing. We discuss virtualization methods for NVMe devices in more details in later sections.

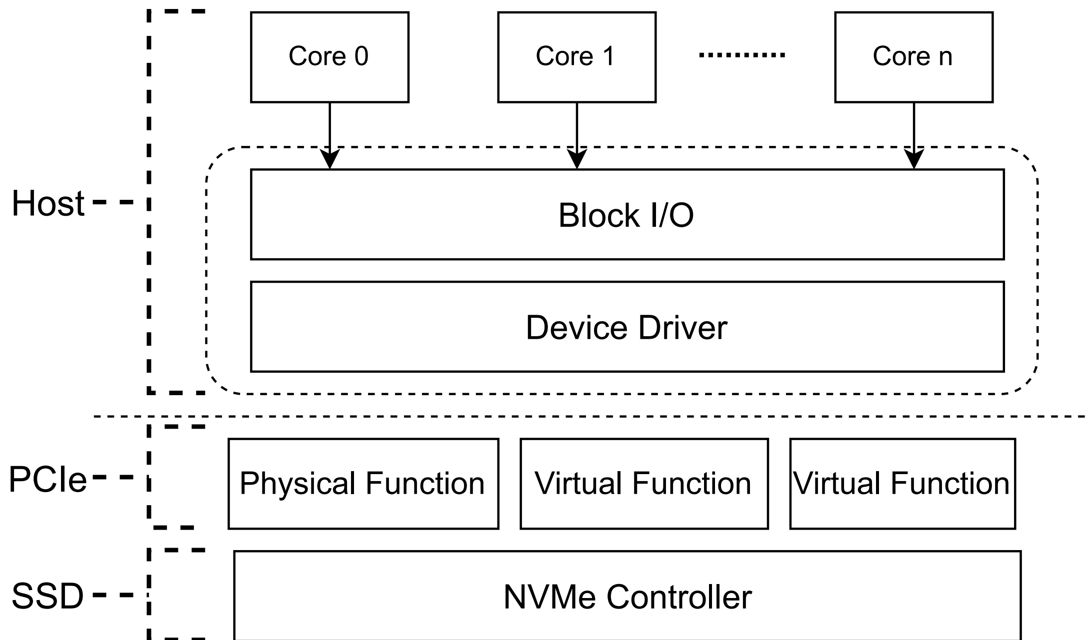


Figure 4.2: An overview of NVMe SSD virtualization.

In order to read/write data from/into the SSD, the host should inform SSD the source address, the destination address, and the amount of data. For example, a write command specifies the data to be written with logical block address, the SSD parses the command, initiates data transfer by setting source address (host memory) and destination address of DMA engine via the PCIe physical link, maintains a mapping table between logical address and physical address, and finally writes the data to the flash memories.

For NVMe SSDs, an NVMe command is a 64 Bytes long data structure that encapsulates information about command type, the DMA address of the data to be processed, the logical block address, command-specific fields and reserved fields, as shown in Figure 4.3. The reserved fields can be used to enable context exchange between the host and internal flash firmware without re-designing the entire protocol. Specifically, the NVMe driver and device-side controller should be extended to enable such message exchange across the storage stack.

Bytes	Description
63:40	Command Specific
39:24	PRP Entry
23:16	Metadata Buffer
15:8	Reserved
7:4	Namespace Identifier
3:0	Command Type

Figure 4.3: NVMe Command Format.

The NVMe driver is a software layer between block I/O layer and the device-side NVMe controller, which is responsible for generating actual NVMe commands for I/O requests from upper layers and dispatching them to the device. The device-side NVMe controller is responsible for executing received commands, managing hardware queues

and sending response back to the host.

4.1.2 Unpredictable Performance of SSD

Predictable stable performance is hard to achieve in NAND-based SSDs due to unpredictable internal activities. The SSD firmware rarely regulates background activities with little regard to on-going user I/O. What is worse, the existence of co-located I/O flows could exacerbate the problem by introducing more background activities. In this section, we discuss several SSD mechanisms that may cause performance instability. We quantify which internal mechanism results in high latency variations.

Garbage Collection. NAND flash does not support in-place update or page-level erase, erasure operations must be performed at a block granularity while read/write I/O operates at the page granularity. The asymmetry of operation granularity forces the FTL to reclaim old blocks that are filled with invalid pages occasionally. A GC operation is costly since it needs to copy valid pages of that block into new spare blocks. The induced GC traffic can block user I/Os in a unpredictable manner, thereby causing severe tail latencies. There is a huge latency gap between a normal case and a GC-blocking case, the difference of an ideal latency without GC and a realistic latency can be as large as 80x, suggested by our in-house simulator described in Chapter 2. Modern enterprise-level SSDs are reserving a generous amount of Over-Provisioning capacity to smooth out performance penalty incurred by GC.

Data Placement. Rich hardware-level parallelism is commonly available in modern SSDs. A common way to increase throughput is via data striping, the black-box FTL

determines the data placement in a way that can provide most parallelism. However, this could likely lead to interference between different workloads as data pages from different workloads may be placed in the same physical block. The data mixture at the block level can introduce several negative performance impacts. First, it naturally results in resource contention among all workloads residing on the block since a block's page buffer can only serve one read/write operation at a time. Second, if the co-located pages have different life time, it intensifies the GC cost because GC can be triggered more frequently. Lastly, there are conflicts at other levels such as shared channel data bus, and chip-level flash transaction queue, plane-level address buffer, and so forth that result in resource contention. Data placement inside SSD has a profound impact on predictable performance, especially under a shared I/O intensive environment. A natural idea of achieving deterministic performance is to partition the flash channels and assign physically partitioned resources to different workloads [118, 150, 164], however, these solutions sacrifice the aggregate bandwidth.

Caching. Resource sharing on an SSD is not limited to flash chips but also on-board DRAM. To increase the controller bandwidth and approach theoretical bandwidth of aggregated flash memories, modern SSDs adopt the on-board DRAM to buffer I/O requests and mapping tables. However, it does not provide cache space partitioning across I/O flows, which results in poor cache hit rate. Second, the fixed mechanism of buffering hot data and mapping table entries are not working for every workload because they do not consider any workload characteristics, and the I/O patterns of workloads are typically different from others. As all I/O flows and mapping entries share the cache space indifferently, they contend precious cache blocks, resulting in widely fluctuated

performance. Third, during buffer flush, corresponding flash chips cannot process other read requests, resulting in high read latency.

Prefetching. Prefetching preloads data speculatively from flash memories into on-board DRAM, to serve read requests faster [190]. Prefetching is essential for read performance of I/O workloads that present good temporal locality. If the data is prefetched earlier than needed, it might not available when it is actuall needed due to eviction. If the data is prefecthed later than needed, this prefetch is invalid. Since the prefetching capacity is very limited, multiple I/O flows may result in polluted or thrashed prefetching cache. Ideally, the prefetcher should adapt to I/O patterns dynamically so that the data fetched served exactly when it is needed. It is desirable to separate prefetching of multiple I/O flows to prevent cache pollution.

To figure out how each feature causes high latency variations, we use our SSD simulator to analyze the source of tail latency. First, we extract the ideal tail latency by disabling GC functionality and simulating a sufficiently wide and large internal buffer. We use the ideal latency as the threshold number to distinguish high latency request from normal latency request. The high latency cause is credited to the module where the request stalls for the longest time. It is worth noting that most of the requests fall into the normal latency category. On average, GC-induced stall and buffer-induced stall account for 30.1% and 49.2% of the high latency request. The evaluation results across various types of workloads are shown in Figure 4.4. We observe that I/O requests are frequently blocked by (1) writing data from write buffer to the SSD backend (2) GC operations. The results suggest that it is crucial to reconcile GC and write buffer designs to achieve tiny tail latencies.

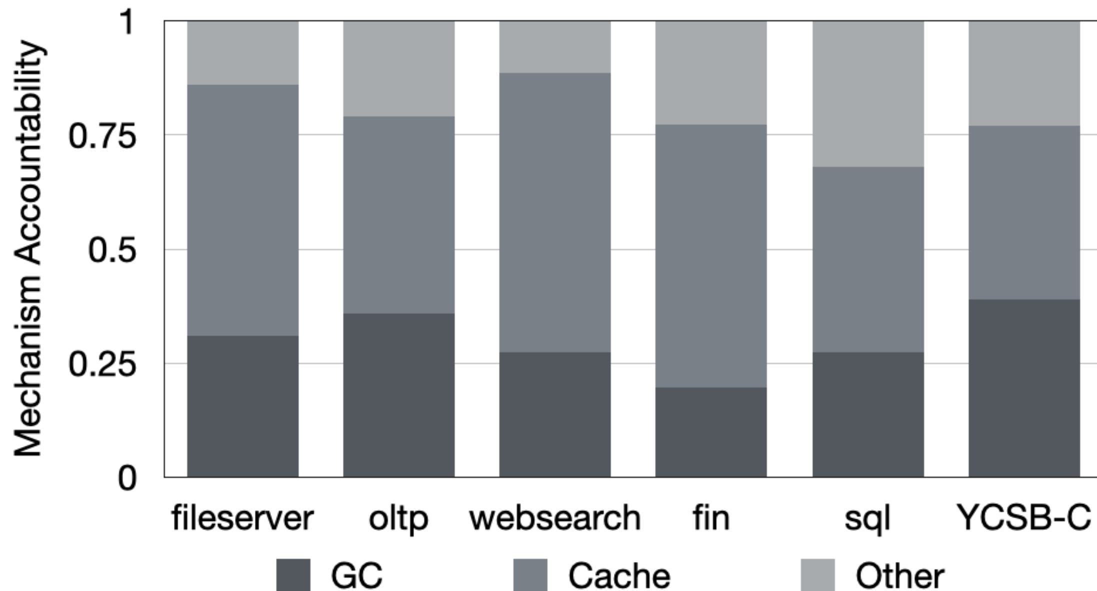


Figure 4.4: Breakdown of feature accountability for high latency.

With the evolution of flash chips from SLC to MLC, and TLC, the performance asymmetry between the read and write becomes wider significantly. Thus, the read performance will be more unpredictable by the on-going write requests as the performance penalty caused by write disturbance becomes more severe. This motivates us to eliminate the contention between the on-going write requests (either user I/O flows or GC/buffer flush induced write I/O) and the incoming read requests. This helps simultaneously improve the user I/O performance and reduce tail latency of an SSD.

4.1.3 I/O Virtualization

I/O virtualization is a crucial component in today's cloud computing infrastructure. It enables time- and space-multiplexing of I/O devices to multiple VMs. Limited physical

PCIe slots and high-density storage requirement makes I/O virtualization an indispensable technique to share storage resources among tenants. Besides, I/O virtualization provides a simplistic and consistent interface for resource management and isolation.

The industry has spent significant effort in increasing hardware resource utilization and minimizing virtualization overhead through different I/O virtualization technologies. There are two distinct approaches, software-based and hardware-based. Software-based techniques do not require hardware supports from a special type of SSD, they adopt virtualization capabilities provided by the hypervisor or OS in the host. The advantage of software-based I/O virtualization is they are cheap and flexible to support random configurations of virtualization, but the cost is severe performance degradation. Hardware-based techniques make use of I/O Memory Management Unit and direct DMA to emulate multiple virtual devices on a single physical device. [136] summarizes three major NVMe virtualization approaches: (1) implementing SCSI to NVMe translation layer on the hypervisor (blind mode), (2) NVMe stack by distributing I/O queues amongst hosted VMs (Virtual Mode) (3) SR-IOV [79] based NVMe controllers per virtual functions (physical mode). Modern virtualization solutions for NVMe Storage such as Virtio [219], Userspace NVMe driver in QEMU [37] and Storage Performance Development Kit (SPDK) [270] are implemented in the userspace of the Linux system. However, their performance are much lower than native drivers. SR-IOV allows bypassing the VM's involvement in data transfer by providing each VM independent memory space, interrupts, and DMA streams. SR-IOV introduces two terms, physical functions (PFs) and virtual functions (VFs). PFs are full-featured PCIe functions just like a normal PCIe device, VFs are only capable of moving data in and out. Each VF can support a separate data path for I/O functions.

Near native I/O performance can be attained with SR-IOV. SR-IOV, together with NVMe namespaces, allows multiple virtualization services to access its exclusive hardware as if it accesses a real device. Each virtualization service can attach an emulated SSD instance by associating with the corresponding VF.

With full virtualization, the guest does not know it's running on a hypervisor and the guest O/S doesn't need any changes to run on a hypervisor. Whenever the guest makes I/O calls, they are trapped on the hypervisor and the hypervisor performs the I/O on the physical device. the guest OS is made aware that it's running in a virtualized environment and special drivers are loaded into the guest to take care of the I/O. The system calls for I/O are replaced with hypercalls. With the paravirtualized scenario, the guest-side drivers are called the frontend drivers and the host-side drivers are called the backend drivers. Virtio is the virtualization standard for implementing paravirtualized drivers. The frontend network or I/O drivers of the guest are implemented based on the Virtio standard and the frontend drivers are aware that they are running in a virtual environment. They work in tandem with the backend Virtio drivers of the hypervisor. This working mechanism of frontend and backend drivers helps achieve high-performance network and disk operations and is the reason for most of the performance benefits enjoyed by paravirtualization.

Virtio and Virtual Function I/O (VFIO) both have their shortcomings: Virtio uses the native virtio-blk driver, and it does not include specific optimization for NVMe. Thus, Virtio needs to go through the thick kernel software layer, including the Virtual File System (VFS), block layer and I/O scheduler. The throughput and latency in VMs in the virtio scheme can only achieve half of the native performance. VFIO scheme uses

direct pass-through to assign the entire NVMe device to a single VM so that VFIO can achieve near-native performance on both latency and throughput, but , it is not possible to enable effective device sharing and migration. Virtio implements a common set of interfaces in the frontend drivers. When a guest process in an VM issues an I/O system call the frontend driver API will be invoked, and the driver passes the data packets to the corresponding backend driver through the virtual queue(virtque).

Besides VM, Linux container technology has recently seen a rapid rise in deployment due to minimal application footprints and improved resource utilization. Under the hood, containers are built upon two critical Linux kernel components – namespaces and cgroups [9, 211, 212], which provide per-process resource management and isolation. The Linux namespace provides a way to achieve logical-level isolation. Each NVMe namespace can be addressed with a ns_id and contains a distinct continuous set of logical blocks. This feature does not provide any guarantee for performance isolation since the namespace blocks are purely logical and can be mapped to any physical blocks. Namespace creates an abstraction of a global resource that gives the container an illusion that it has its isolated instance of the global resource. cgroup is a Linux kernel mechanism that provides resource control, limit and prioritization for each process. These resource include CPU, memory and I/O. The resource controller tracks the resources, accounts them to processes and dynamically allocates/throttles them. With cgroup, users can ensure that the target workload is not impacted by system background or co-located workloads that use excess system resources. Users can also reserve system resource to high priority workloads by limiting available resource to low priority workloads. As container can relatively handle hardware-level isolation of CPU, memory and network, it is challenging to provide similar

level of I/O isolation because the OS kernel is completely disconnected from the device internal states. Theoretically, this makes software-level I/O rate limiting strategies almost futile for guaranteeing I/O QoS since request stalling inside the device is not tamed. We demonstrate this observation in Section 4.1.4.

Table 4.1: Experimental configuration.

OS	Ubuntu 18.04.5
Kernel	Linux 5.4
CPU	AMD Ryzen 9 5900X 4.5 GHz 12 cores
Memory	128 GB DDR4 3600 MHz
Storage	Intel DC-P4510 1TB U.2
Workloads	FIO 8KB random write, iodepth = 4 FIO busrty write, blocksize 60% 256KB, iodepth = 1 FIO 4KB random read, iodepth = 1

4.1.4 Evaluation of Performance Interference

In this section, we first describe how I/O SLA is adversely affected when multiple tenants naively share the NVMe SSD. There are three major sources of performance interference in a multi-tenant storage system: (1) I/O intensive writes that induces background GC activities. (2) unpredictable buffer flush to SSD backend block user I/Os. (3) Internal buffer contention. To quantify the performance degradation caused by neighbor I/O flows, we perform several experiments of running two containers in parallel using an enterprise-

level NVMe SSD [2]. Our experimental setup is summarized in Table 4.1. First, we run a container instance of a latency-sensitive workload which issues 4KB random reads. The average latency when running alone is around 90us. This workload simulates an online-service service that requires stable and low latency. The background workload simulates a write-intensive workload that has a bursty write pattern, which is expected to invoke GC frequently. We vary the interference level by changing its write intensity and pattern. Here, the I/O intensity refers to the rate that the flow issues I/O requests. Before running the experiment, we warm up the device to steady-state [241] with two full-capacity writes and erasure. The drive is filled to its half capacity to simulate real-world use cases. We measure the read latency of the latency-sensitive workload alone and with the background workload co-running. Figure 4.5 shows the latency distribution of the latency-critical workload running alone and with the background workload. Figure 4.6 shows the detailed breakdown of time spent at different layers. When running the latency-sensitive workload alone, the average latency stays stable at 90us and exhibits a tiny tail. When the background workload is started, the average latency and p99 latency increase by 2.6x and 3.7x compared to the baseline, respectively. As we increase the intensity of background writes, the average latency on the read service gets 4.9x higher than the baseline. This can be regarded a major outage as the request latency climbs so high. One may wonder if the performance interference can be mitigated by limiting the activities of background writes. We evaluate the state-of-the-art I/O control mechanisms, cgroup blkio. We limit the I/O rate of the background write activities by setting its maximum throughput. The p99.9 (99.9th percentile) tail latency shows a significant reduction when the background I/O activities are throttled to 50MB/s, however, it is still 1.78x to the

latency when it is running alone. This is because hypervisor-level interference is not the sole cause, and software-based throttling can not resolve any device-level interference. When the background workload is further intensified (by duplicating another background I/O), the average latency experiences a dramatic 9.2x drawdown, the p99.9 tail latency is near 19x worse than the baseline case. As a consequence, it is urgent to investigate SLA violations when multiple tenants share a single SSD.

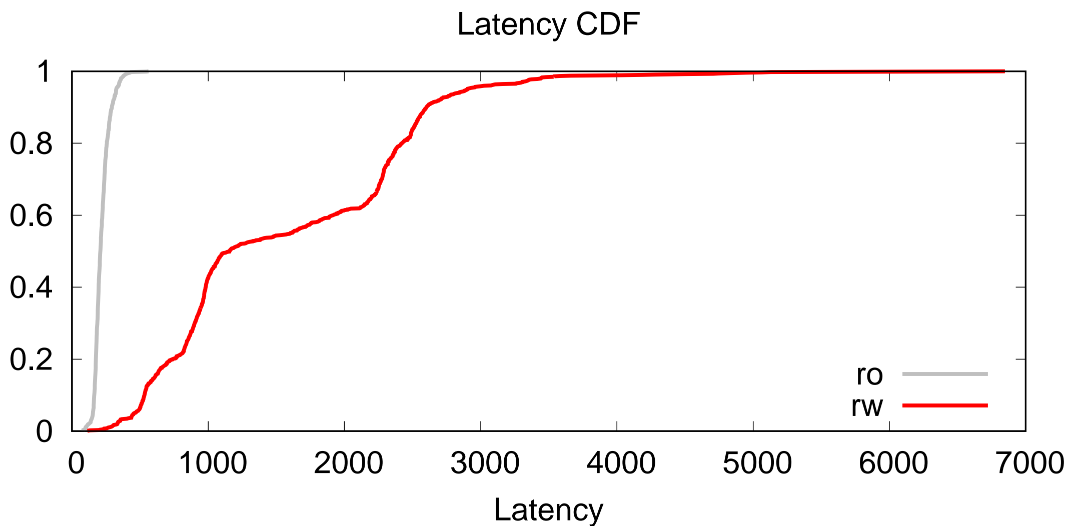


Figure 4.5: The latency (us) distribution of the latency-critical workload.

4.1.5 Access Latency Breakdown

In-depth performance analysis is crucial for systems research. In this section, we provide breakdowns of end-to-end latency to identify root causes of increasing latencies. We follow similar tracing procedures described in [259]. Figure 4.6 plots the latency breakdown of the same test in Table 4.1.4. The total latency is the time consumed after issuing a

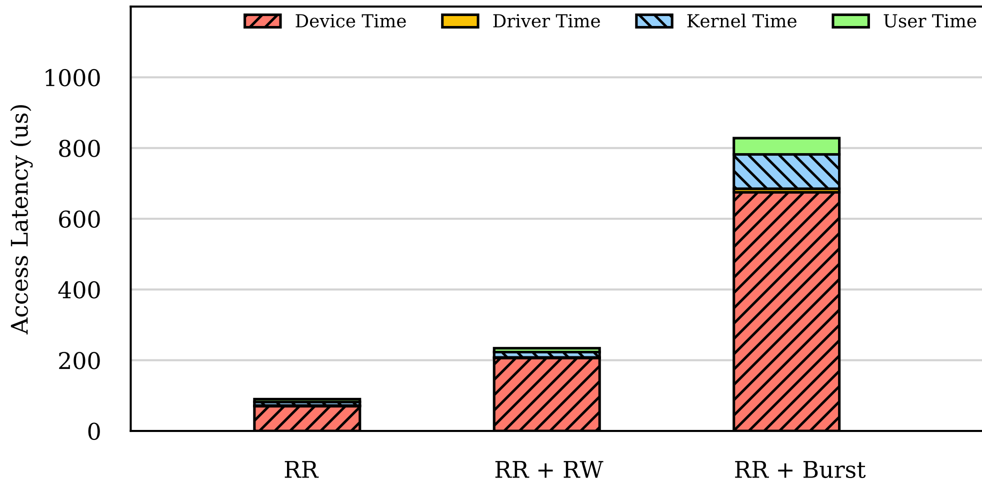


Figure 4.6: The breakdown of average access latency for 4K random read running with RR (random read only), RR + RW (with background random write), and RR + Burst (with multiple write bursts).

request until it is completed.

In the baseline case, the device latency accounts for 78% of the total latency. Excluding the time spent in the device, the remaining time is spent in I/O scheduling and merging in the block I/O layer (kernel), and the device driver (driver), which makes up 22% of the total latency. When a mixed read-write I/O workload is started, the overall latency sees a 2.6x increase, and the time spent in the device becomes an even larger contributor to the overall latency, tagging a 2.9x increase compared to the baseline. When a write-intensive workload that issues write bursts is started, the absolute time spent in the device increases 9.6x, still dominant in the total latency. The absolute overhead of driver and user time modestly increases. Note we observe a noticeable increase in the time spent in the kernel, where the block I/O layer tries to aggregate multiple I/O requests and schedule them. We make an important observation, device-level interference is the

major source of overhead.

4.2 Existing Approaches

Alleviating Performance Interference in Shared SSD. Several studies have been proposed to alleviate performance interference in a multi-tenancy environment [118, 138, 150, 284]. The multi-stream [284] interface enables data placement by lifetime by storing incoming data into separate flash blocks according to the stream ID, in contrast, conventional SSD naively places data in their original request order. This helps to eliminate GC frequencies because all the data stored in the same flash block are more likely to be invalidated at the same time. TTFlash [263] alleviates GC-induced raed performance variability by leveraging parity redundacy among flash chips. FlashBlox [118] partitions parallel hardware resources to achieve resource isolation. DC-Store [150] combines hypervisor-level and device-side isolation at the cost of reduced aggregate bandwidth. NVM Set [6] architecture divides flash memories into multiple NVM sets for I/O isolation, the isolated NVM sets operate independently and this provides I/O determinism between NVM sets. Other works [38, 161] attempt to implement fair I/O schedulers to guarantee fair storage resource share for each VM. Unfortunately, these solutions cannot resolve conflicts at device-level.

Exposing SSD Internals to Host. Several novel interfaces have been proposed to expose SSD internal resources to the host and enable more communication between the host and the SSD. Open-Channel SSDs [42] exposes the physical storage layout and let the host

manage data placement and I/O scheduling, allowing it to fully utilize the device's raw parallelism. Open-channel SSDs are supported by Linux via the LightNVM framwork [42]. The Linux kernel extension, Pblk [93, 124] behaves as a host-based FTL that provides a block interface abstraction to legacy applications. However, attempts in commercializing Open-Channel SSDs have not gained much success mainly due to the development endeavor for new users. Zoned Name Space (ZNS) [10] has been part of the most recent NVMe standard recently, where SSD is divided into several "zones" each of which is managed independently as a log-structured "device", i.e., logical blocks within a zone must be written sequentially. The host explicitly controls the data placement and GC of zones. Therefore GC-induced interference can be avoided for zones requiring deterministic latency. Recent NVMe specifications have introduced a predictable latency mode (PLM) [6, 216], which manages the latency caused by background activities by limiting those activities to non-deterministic windows. During deterministic windows, SSD holds back background activities so that it can provide predictable performance. PLM introduces two NVMe commands that allow the host to query/alter the SSD state. This is a significant interface-level leap bridging the gap between host requirements and SSD status.

4.3 Design

Section 4.1.4 and Section 4.1.5 demonstrate that the performance degradation is largely incurred by resource conflicts at device-level. In this section, we propose a fine-grained FTL architecture, Zoned FTL (ZFTL), which leverages hardware virtualization capabilities

to support the notion of private FTL, and provide device-level resource isolation for each virtualized I/O while preserving close-to-native parallelism. We propose a communication approach that allows SSD to be aware of virtualized I/O service with modest modifications in the host, NVMe driver and controller. The extra information passed to the SSD allows its firmware to achieve strong isolation regarding multiple shared resources and optimize buffering policy to accommodate workload characteristics, which can mitigate the interference from other I/O flows. Most of ZFTL designs are firmware-level re-architecting.

We present an overview of ZFTL architecture in Section 4.7. ZFTL design is different from conventional FTLs in the following ways: (1) As a single PCIe SSD can be exposed as multiple physical instances to the host as if they are real devices, we re-architect SSD firmware to initiate multiple private FTLs (pFTL) that handles requests from its corresponding physical instance. Each pFTL serves one SLA-requiring I/O flow and performs flash management operations on its respective set of physical blocks. (2) ZFTL provides a new abstraction to the host, that is, a dynamic collection of flash resources that are isolated from the rest of the drive. A virtualized I/O workload can attach to a pFTL-enabled emulated SSD by binding to the corresponding virtual function. (3) Each virtualization-aware pFTL maintains its own mapping table and caching management. (4) Each pFTL relies on a centralized block allocation service to apply for spare blocks that have minimal contention with other pFTLs.

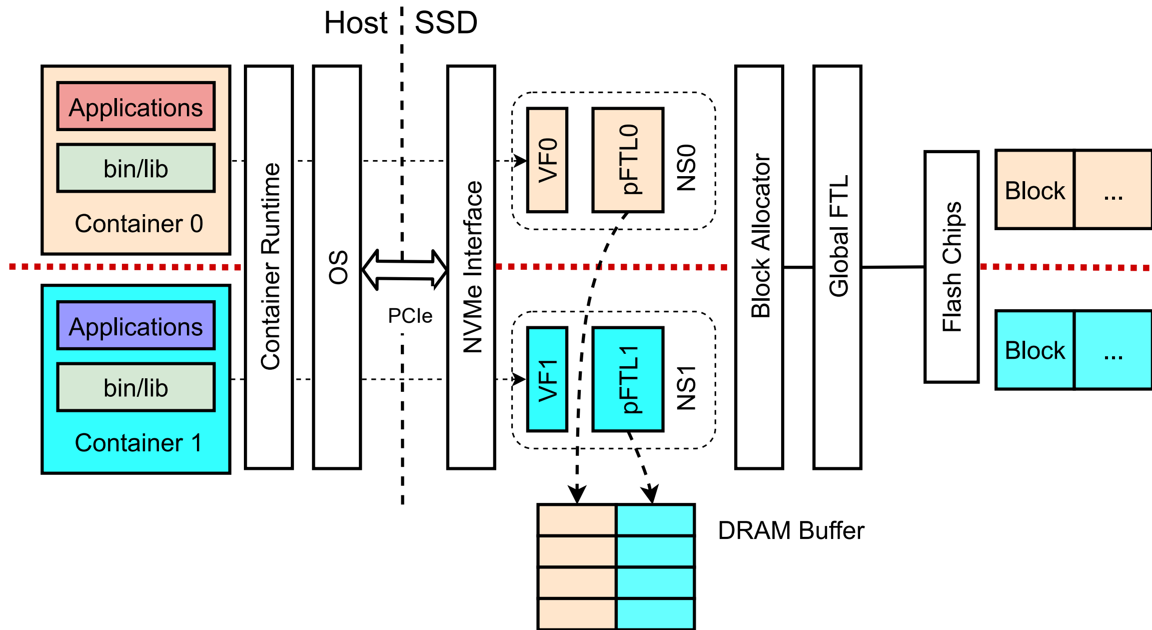


Figure 4.7: Overview of ZFTL architecture.

4.3.1 Design Goals

- First, we should isolate shared hardware resources among I/O flows as much as possible, to alleviate potential interference. The shared hardware resources include but not limited to flash blocks, chip-level flash transaction queue, shared channel data bus, write buffer, cached mapping table and block management. Especially, according to discussions in Section 4.1.2, isolation efforts should emphasize on write buffer and data placement, since they are the main causes of high latencies.

- Second, we shall not redesign the entire interface that overthrows existing software/hardware stack and puts additional burdens on application developers, based on lessons gained from OpenChannel SSD [202]. We wish to hide the isolation efforts from users

but explicitly allow flexible configurations to be deployed. Therefore, we propose to extend existing NVMe semantics and enhance NVMe driver and controller to enable communication between the host, virtualized application and underlying storage device. ZFTL is designed to be an external, black-box interface that requires very little modification on the application side.

4.3.2 Overview

Figure 4.7 illustrates the overall architecture of ZFTL, consisting of a host system, PCIe fabric and the endpoint NVMe SSD. ZFTL is an I/O isolation enhancement architecture using the multiple virtual functions and namespaces, which are logical isolation features for PCIe and NVMe. The core components of ZFTL include: (1) an interface that sets up device virtual functions and namespaces, pFTL (private FTL) instances according to namespace LBA ranges, assigns corresponding virtual functions to the virtualized I/O flow, and bypasses I/O requests to pFTL that they are attached to, (2) private FTLs to handle I/O requests from their corresponding virtualized I/O flows, manage flsh resource and buffering policy, (3) a global controller/FTL that assigns spare blocks to pFTLs, handles the interrupt and posts completion response to completion queue of the corresponding virtual function, (4) a shared on-board DRAM that partitions cache space across pFTLs. In the following sections, we describe the role of each component in more details.

4.3.3 Interface

The conventional block interface exposes a flat address space that can be read or written at a page granularity. Despite of its convenience, it leaves a huge semantic gap between the host and SSD internals, which makes proper resource isolation extremely difficult due to the lack of ”contract” between them. To address this problem, we enable a physical resource isolation scheme across the storage stack using the logical isolation features for PCIe and NVMe. Specifically, we revise I/O service routine that delivers I/O requests from a virtualized application (either a VM or container) to an isolated ”sub-SSD”. First, the host enables SR-IOV feature and sets up virtual functions and namespaces. The host attaches the namespace to the virtual function, and a virtual function is assigned to serve requests from the virtualized application. Meanwhile, the host communicates to the SSD via NVMe admin command to initiate a new private FTL and binds it with the specified namespace. Once the virtualized application issues an I/O request to the host block I/O layer, the kernel composes the namespace ID (NSID) in the block I/O data structure (bio). The modified NVMe driver is responsible for generating nvme_rw commands that encapsulates NSID in the submission queue entry and places the command in the corresponding submission queue. Note a submission queue entry is a 64 byte long nvme_rw command that has a 8 byte reserved field. The NVMe driver can override the attribute from host side to this reserved field of nvme_rw command. When the doorbell rings, the device-side NVMe controller fetches the NVMe command, parses the NSID and passes it to the pFTL that is attached to. After pFTL completes servicing the command, a completion response is placed in the completion queue and raises an interrupt to the host.

Figure 4.8 shows the entire process. The NVMe driver and controller need to be modified to enable the host/SSD controller communication and I/O flow redirection. In the host's perspective, there are multiple NVMe devices emulated via SR-IOV presenting in the PCIe complex. To forward I/O flows to an NVMe deice, the user only needs to bind the virtualized application with the virtual function in the host system. This external, black-box interface allows most convenience and management simplicity for cloud customers. In order to further eliminate software queue contention and unfair scheduling in the block I/O layer, the hypervisor could maintain a virtual queue for each VM. This requires host NVMe driver to alter accordingly to fetch and compose NVMe commands.

4.3.4 Resource Isolation

The existing virtual function and NVMe namespace protocol can expose a single NVMe SSD as multiple logical NVMe devices as discussed in Section 4.1.3. Nevertheless, the scope of isolation is limited to logical level, in other words, different namespace can still point to the same physical block and is subject to the centralized management unit. Therefore, a design goal of ZFTL is that each "namespace" operates independently and resists performance interference from other instances. To achieve this, we enforce physical resource isolation in flash memories, flash management unit and internal buffer, as Section 4.1.2 points out that they are the major sources of contention-induced performance interference.

First, none of the flash blocks are allowed to share among namespaces, in other words, the granularity of resource allocation for each namespace is a flash block. This is

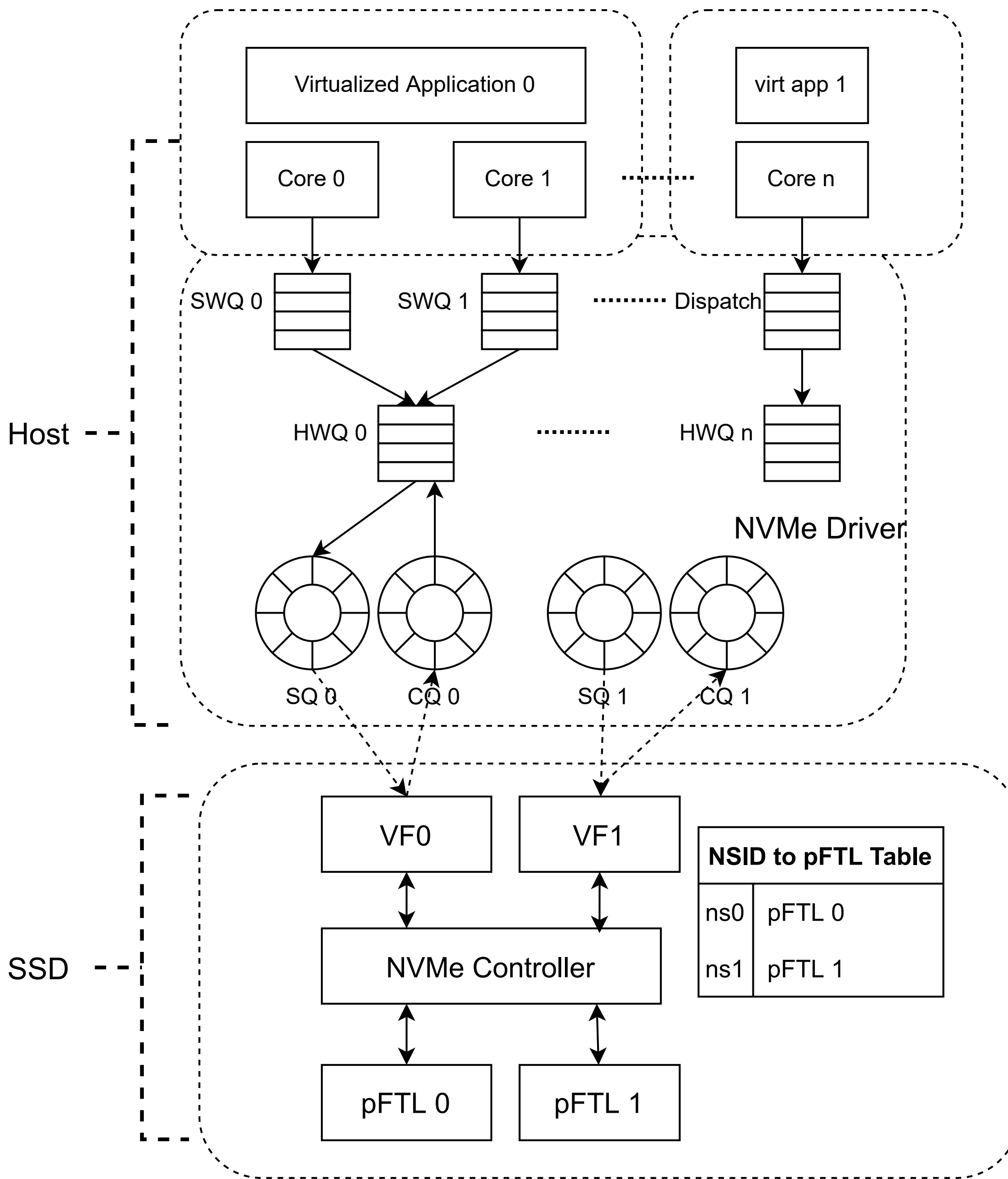


Figure 4.8: Virtualized applications interact with ZFTL interface and command flow.

because data mixture at block level would lead to uneven recycling distribution, result in unpredictable response time for all I/O services sharing the block due to GC interference. Second, flash resource conflicts at higher levels should be minimized, for example, some commodity SSDs adopt a channel-level QoS-aware scheduler that decides which flash chip services the next request, and each flash chip maintains a flash transaction queue for arbitration. To this end, chip-level isolation is desirable for satisfying QoS guarantees of virtualized applications. In ZFTL design, we do not adopt plane-level isolation because that would void the throughput benefit brought by multi-plane operations (plane-level parallelism will be voided. When the FTL invokes GC on one of the planes, because of restrictions of the internal circuity of the flash chips, the other plane cannot service I/O requests and remain idle. Third, the internal cache should not be shared naively. For instance, once the host OS invokes a heavy flush operation, the FTL receives many bursty write requests, these writes will be buffered aggressively before directly writing them to the backend flash memories to maximize the throughput. This raises two problems: (1) The limited cache space will be drained (especially the block size of the I/O request is huge) quickly and begin to starve other I/O services which are susceptible to caching efficiency. (2) writing buffered data is inevitable and makes the target flash memories too busy to serve other I/O requests due to the uncertainty of when to execute the write-back. Therefore, we advocate that each virtualized instance owns its exclusive DRAM region for write buffer and caching mapping entries according to its sensitivity to cache space and policy. Fourth, since every I/O flow has a substantially different lifetime distribution of data blocks, the efficiency of garbage collection and wear leveling can get much higher if they are managed independently.

For above considerations, ZFTL virtualizes the conventional FTL into multiple instances. Each virtualized FTL, we call it private FTL (pFTL), serves the I/O requests from its corresponding virtualized application via the namespace and virtual function binding. A pFTL is a software component that is responsible for: (1) maintain a private page mapping table (2) manage flash operations (GC and WL) on its exclusive set of flash blocks. (3) manage and maintain a list of its owned resource. A pFTL is initialized when a virtualized service binds to a VF. Users can specify the size of pre-allocated blocks, overprovision (OP) ratio, cache space and write policy using the admin NVMe command [5]. A pFTL works similarly to a global FTL except: (1) It is unaware of other pFTLs. (2) It is unaware of the global SSD resources. It needs to apply for dynamic resource allocation from a global FTL (applying new blocks and recycling worn blocks). (3) It can be terminated like a function, and related resources are released when the life cycle of the bound service ends. Such characteristics not only isolate virtualized applications at device level but brings additional performance benefits: (1) Heuristically better GC efficiency as files created by the same application, or at similar times are more likely to expire at about the same time, compared to those with different owners and creation time. (2) Better caching efficiency as each pFTL can specify the cache and eviction policy to accommodate its I/O patterns.

4.3.5 Block Allocator

The block allocator is responsible for assigning new blocks to a pFTL when the host issues a resizing request or a pFTL is running out of open pages. When a pFTL requests a

new physical page, it first looks up its list of ”open” flash blocks (blocks that are not full with written pages). If the list is non-empty, the pFTL will pick the most recent opened block and then attach the page to the end of that block. If no, the pFTL will raise a new block request to the global block allocator. The strategy that block allocator distributes new blocks to different flash channels, packages, dies, or planes could significantly impact the parallelism and response time for each pFTL instance. Given the design goal of ZFTL is to minimize intra-flow interference and bandwidth is less of a concern with the current PCIe Gen4 SSDs are equipped with an ”excessive” number of channels and ways, we treat stable response time, especially tail latency, as a more important metric in ZFTL design. Additionally, the data striping strategy to boost throughput is orthogonal to ZFTL design. Isolation and parallelism is a trade-off question, for example, if the block allocator assigns blocks within a chip to a PFTL, isolation can be compromised, as intra-chip parallelism may cause some GC side-effects: when a plane is busy doing GC, the other plane(s) in the same die can not be accessed, as all the planes in the same die share a single command and address path that is occupied by GC. If the block allocator assigns blocks within a channel to a pFTL, parallelism is compromised in this case. Therefore, ZFTL gives the highest priority of picking blocks within the same chip so that the chip-level queue can better isolate interference from other I/O flows, and striping data across planes to minimize potential GC stalls.

4.3.6 Split Cache

As discussed in Section 4.1.2, the shared internal cache is a major source of interference. Unfair caching is quite common in a shared SSD. For example, if an I/O services issues many write-intensive requests or requests with a larger block size, it will occupy more space in the shared cache region. This degrades the performance of other applications that are susceptible to caching efficiency. Workloads and tenants can have substantially different cache space demand and I/O access patterns. Simply sharing cache resources may also lead to inefficient use of shared cache space. To ensure caching fairness, we provide cache space partitioning for each pFTL instance. Each pFTL either has its exclusive partition or has a common shared cache pool. Each exclusive cache partition is either configured by user definition or with the same number of associativity (empirically set as 2 or 4) but a different number of sets. To accommodate cache demand of each pFTL, we create a cache manager that is a component of the global FTL, to monitor the I/O intensity and cache miss ratio of each pFTL instance. The I/O intensity refers the number of flash requests served within a certain time window. The number of cache sets owned by a pFTL can be dynamically resized based on the I/O intensity and cache miss ratio. If the cache manager observes a spike of write activities in a pFTL, it can be ”lend” extra cache sets from its reserve. When the I/O intensity backs down, the cache manager evicts the least recently used (LRU) cache sets. The cache manager can reclaim an entire cache partition by storing the cached content into data blocks if the I/O service stops. Considering the memory page migration is much cheaper than a flash page operation, the overhead caused by additional cache management is almost negligible.

Overall, an effective cache allocation should follow a general rule, the cache manager should gradually approach the cache space needed to minimize the miss ratio. There are other cache configurations, such as write policy and eviction policy that users can specify using an admin NVMe command. For instance, users can select to bypass the write buffer if the pattern of their write requests are largely sequential.

4.3.7 Put It All Together

In section, we illustrate the entire flow of attaching a container to VF, creating a pFL instance, and servicing a read/write request from virtualized applications. As discussed in Section 4.3.4, a PCIe SSD can be exposed as multiple physical instances via SR-IOV. When a container is launched, the hypervisor creates a new NVMe namespace by sending an NVMe admin command consists of namespace ID(`nid`, NS0) and initial configurations to the SSD firmware. The SSD side creates a pFTL instance (pFTL0) and allocates defined resources to NS0 through the block allocator. Note that the allocated resources here are mostly isolated from the rest of the SSD. After that, pFTL0 is bound with VF0. An NVMe command queue and pairs of Submission Queues (SQs) and Completion Queues (CQs) are created for VF0 and pFTL0. After pFTL0 initialization has been completed, the SSD will subsequently notify the host of the success or failure status through an interrupt. The host will assign VF0 to the virtualized application on an initialization success.

Figure 4.9 shows how ZFTL processes a read request from host to SSD. When a virtualized application issues a read request with a starting logical block address (LBA)

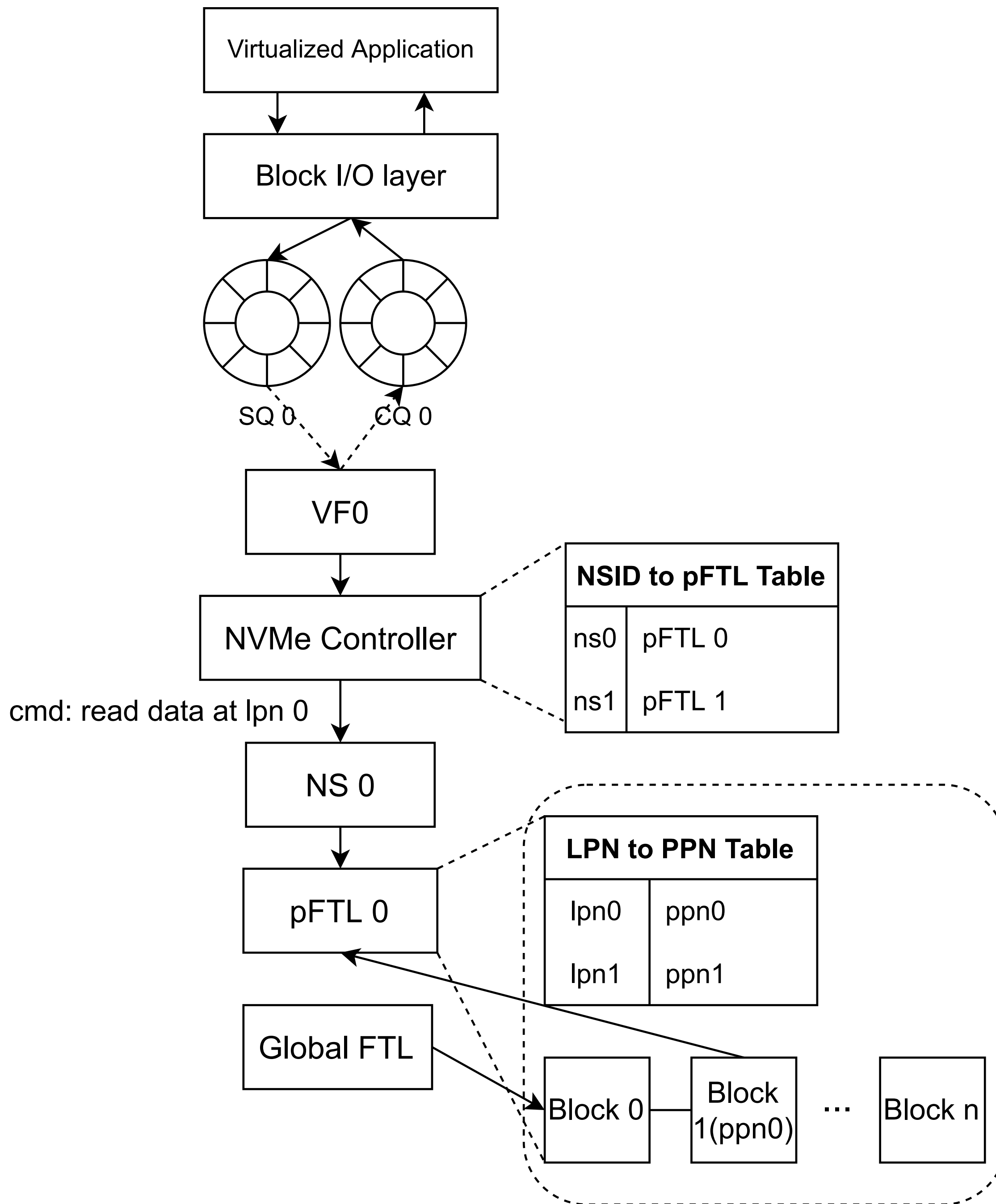


Figure 4.9: Processing a read request across ZFTL stack.

and size, the device driver generates a corresponding NVMe I/O request with such info plus source `nid`. The request is sent through the corresponding VF to the SSD. With `nid`, the SSD can forward the request to the attached pFTL. The pFTL handles the I/O request, places a completion status into a CQ, and generates an interrupt to the host. After the device driver receives the interrupt and processes the completion entries, it releases the completion entries and updates the SQ.

Servicing a write request is similar; the only difference is that it may require allocation of new physical pages or blocks, as shown in Figure 4.10. When a pFTL instance receives a write request, it first checks if there is sufficient space in its open blocks for that request. If not, the pFTL needs to apply new physical blocks through the block allocator.

In summary, ZFTL is designed to bridge the semantic gap between the host and the underlying storage, and alleviate resource contention in storage sharing.

- It partially bridges the semantic gap between host and SSD, by letting the SSD parse the NVMe request generated from the host-side NVMe driver with an namespace tag.

- It enables strong isolation of flash resources among virtualization services while preserving flexibility and parallelism. It is relatively easy to scale with more flash resources because of our finer-granularity method of allocating resource.

- It ensures minimal interference from co-running virtualization applications by setting up an exclusive I/O path, FTL, and cache.

- It achieves group-by-death (data grouped by lifetime) or group-by-application (data grouped by application) GC as a side product decreasing the frequency of GC

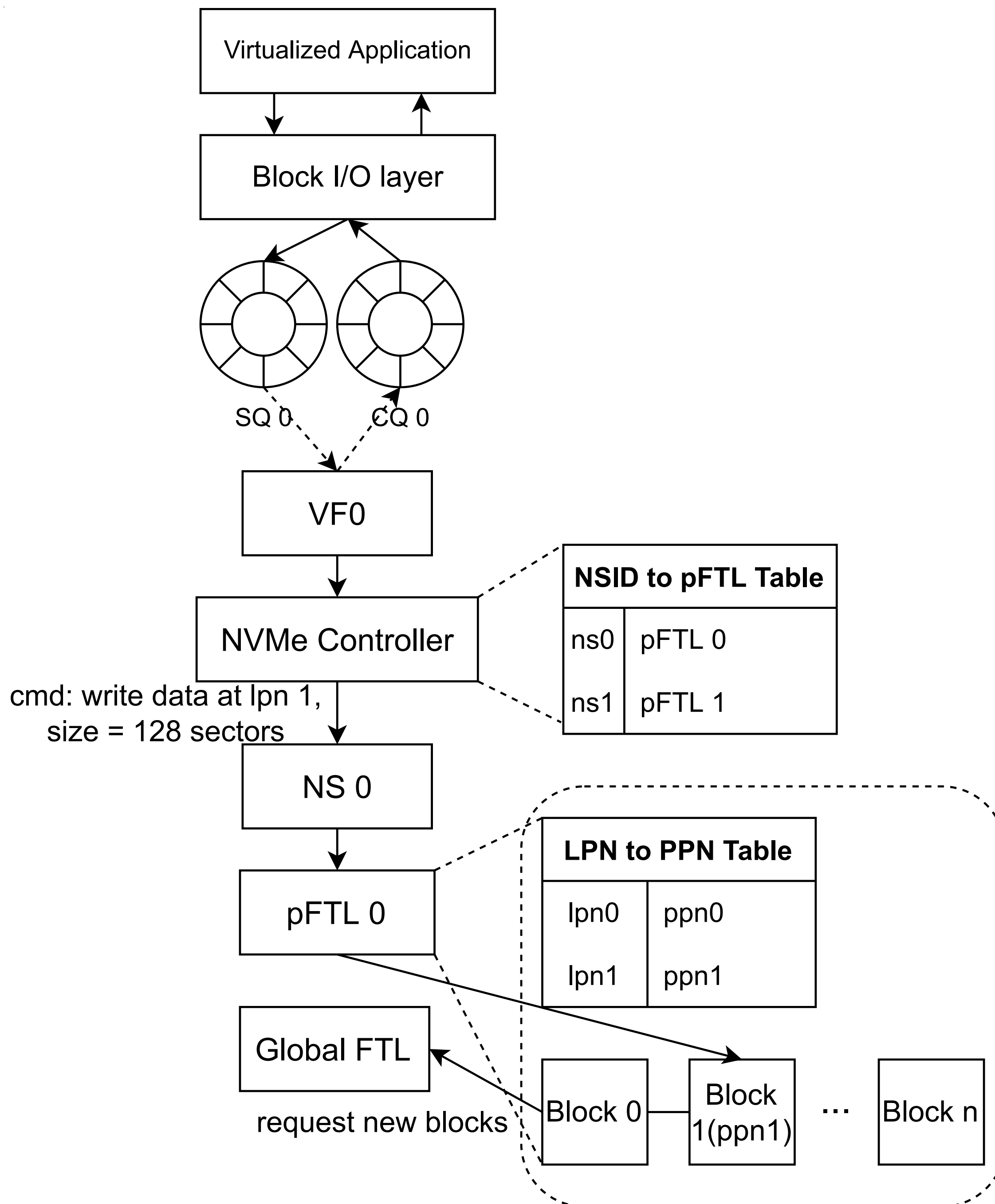


Figure 4.10: Processing a write request across ZFTL stack.

and Write Amplification Factor (WAF). Data are not only written together to a physically related NAND flash space (a flash block), but also separated from data from other applications. Under the same GC policy, this can help decrease the frequency of GC (triggered by co-located data) and Write Amplification Factor (WAF) (reduce costly data movements) like a multi-stream SSD.

- It requires minimal software modification by users as end-users are still exposed to their familiar block interface.

4.4 Evaluation

Implementation Platform. We choose FEMU as the SSD emulation platform for the following reasons: (1) it facilitates split-level research across OS kernel and internal firmware while respecting the NVMe protocol. (2) it enables us to run realistic and up-to-date cloud workloads instead of relying on I/O traces collected on HDDs decades ago. (3) it is easy and cheap to evaluate design tradeoffs than hardware platforms. For example, an ideal candidate, OpenSSD, is limited by its event-based and heavily multiplexed FTL framework, which poses additional complexity in implementing our FTL designs.

However, FEMU has several significant weaknesses that requires heavy modifications: (1) it lacks several performance impacting mechanisms that enable a high-performance NVMe SSD, such as proper data placement schemes, GC policies, and data buffering. (2) it lacks more detailed flash-level characterizations, such as different levels of parallelism and resource conflicts at the chip level.

We model all the major components in a modern NVMe SSD to set up a more

realistic simulation environment. On top of that, we implement the ZFTL design. It should be noted that this work is more performance-centric and focus on firmware-level optimizations at the moment. In the future, we wish to explore software-stack support and optimizations.

Workloads. We evaluate ZFTL with a set of synthetic workloads and real-world applications as listed in Table 4.2. For synthetic workloads, we use FIO and FIOsynth which represent the I/O profiles of a diverse range of workloads. For real-world workloads, we choose the YCSB (Yahoo Cloud Serving Benchmark) on RocksDB. RocksDB issues many update requests due to its journaling mechanism. The OLTP database workload: TPC-C on MySQL is write-intensive and one data-intensive application: MapReduce. We evaluate the run phase of YCSB, where YCSB executes a defined workload with different I/O patterns on the created dataset. TPC-C is configured to run 50 warehouses. The performance of TPC-C is measured in transactions per minute (tpmC) and the 99th percentile latency. The I/O characteristics of workloads used in our evaluation are summarized in Table 4.2.

Experimental Configurations. The emulator runs on a host whose detailed configuration is listed in Table 4.1. We follow the specification of Intel datacenter-grade P4500 SSD to configure our SSD emulator. All of our experiments use EXT4 as the file system. The details of the emulated SSD and benchmark versions are listed in Table 4.3. We compare the proposed ZFTL with the baseline scheme that does not implement any physical isolation (similar to on-shelf SSDs).

Table 4.2: I/O characteristics of workloads

Workload	**Description**
YCSB-A	50% read, 50% update
YCSB-B	95% read, 5% update
YCSB-C	read-only
YCSB-D	95% read, 5% insert
YCSB-E	95% scan, 5% insert
YCSB-F	50% read, 50% read-modify-write
MapReduce	r:w = 1:1
ReadHammer	continually read from a small LBA range
Search	read-intensive, latency-critical
TPC-C	a random mix of rw traffic, r:w = 2:1

4.4.1 Raw Performance

In this section, we analyze the performance overhead of ZFTL with vary I/O request sizes, driven by synthetic workloads. To measure the raw performance of ZFTL, we bypass any OS-level caching and issue direct I/O requests to the SSD in each scheme. The baseline means the conventional SSD without resource isolation. Figure 4.11 and Figure 4.12 shows the overhead of ZFTL on FIO random read latency and throughput with varying block sizes, respectively. The results show that ZFTL presents similar random read latency when compared to the Baseline case, within an average of 6% slowdown. As for

the throughput metrics, ZFTL shows an average of 14.3% less throughput than Baseline. However, as we apply the same page interleaving scheme on the Baseline case, this gap shrinks to only 4.6%. For sequential benchmarks, we observe ZFTL exhibits almost identical performance as the Baseline case. This is because the write buffer efficiency is not adversely impacted in ZFTL design. This set of experiments demonstrates that the performance overhead of ZFTL is slight compared to conventional design in terms of raw bandwidth and access latency.

Table 4.3: Parameters of emulated SSD

SSD Layout	**Latency**
Capacity = 256 GB	Page read = 50μs
# of channels = 4	Page write = 300μs
4 chips/channel	Blcok erase = 3.8ms
4 dies/chip	
2 planes/die	
2048 blocks/plane	
256 pages/block	
4KB page	

4.4.2 Evaluation with Synthetic Benchmark

To evaluate the effectiveness of ZFTL, we first host two containerized FIO synthetic benchmarks, one is a 4KB random read, simulating a latency-sensitive service that users expect to receive response within a predictable , the other one is a 4KB write-centric

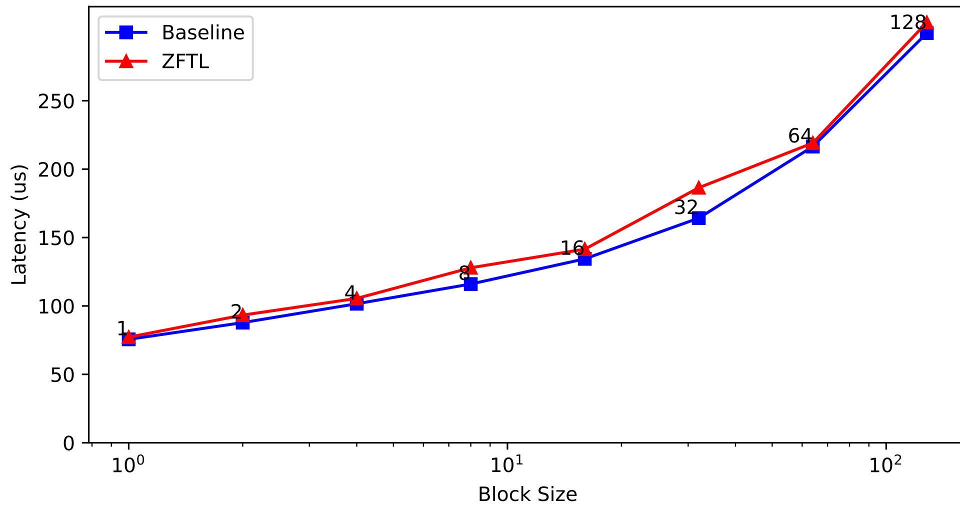


Figure 4.11: FIO random read latency with varying block sizes.

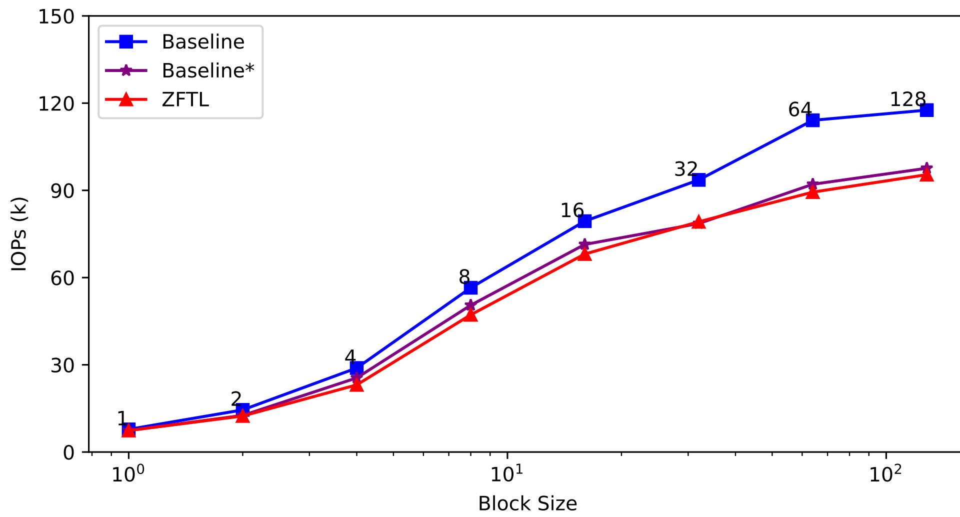


Figure 4.12: FIO random read throughput with varying block sizes.

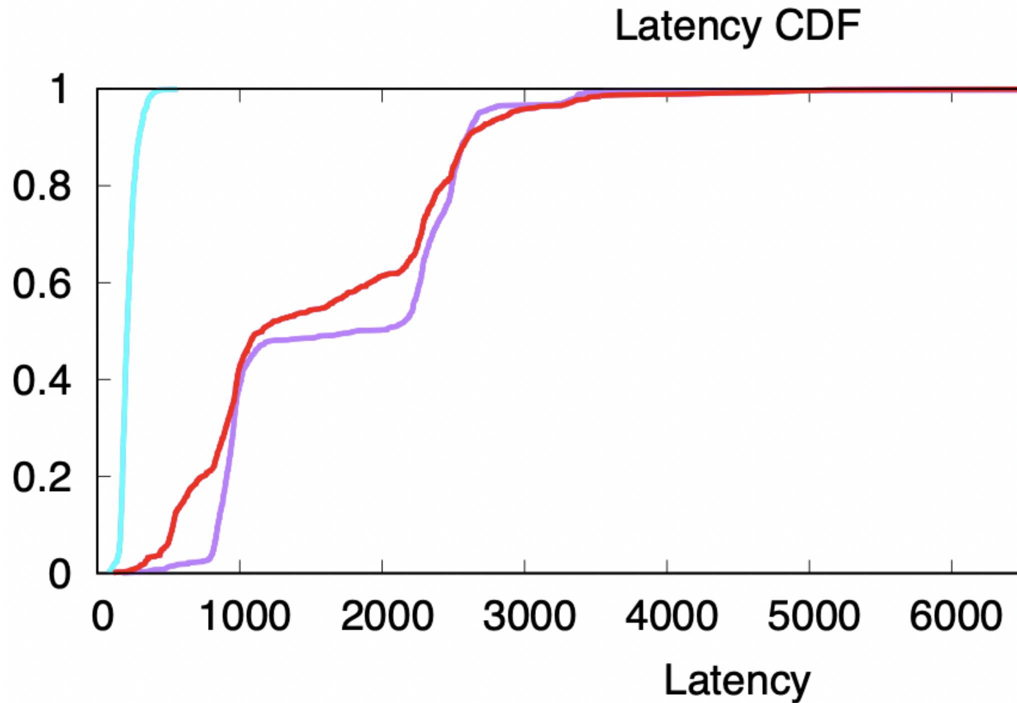


Figure 4.13: The latency (unit:us) distribution of running alone, baseline, and ZFTL.

workload that simulates a throughput-first service that users expect stable and hight throughput. We compare the latency distribution of the first workload under ZFTL and baseline scheme in Figure 4.13. The experiments show that the latency curves in both ZFTL and baseline scheme are stretched due to I/O interference. The p99 tail latency in baseline scheme is 4.17x higher than that of the running alone, indicating the interference of the throughput-heavy workload is fierce. This is because the heavy write operations invoke GC operations, which in turn increases the interference to the first workload. Despite the heavy interference, ZFTL manages a slightly worse p99 tail latency but significantly better than the baseline scheme. The isolation efforts of ZFTL to alleviate interference-induced tail latency when workloads are deployed together are effective.

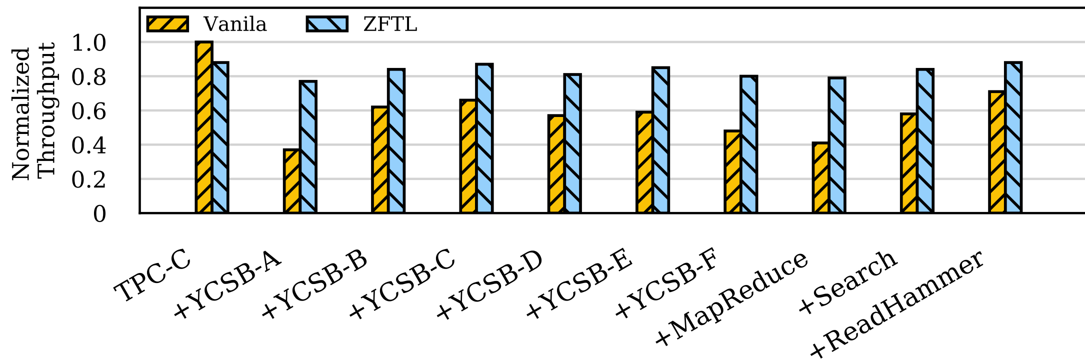


Figure 4.14: Throughput of Baseline, ZFTL scheme normalized to solely running TPC-C on Baseline system.

4.4.3 Evaluation with Realistic Background Interference

To evaluate how well ZFTL behaves in isolating background interference, we evaluate different combinations of containerized workloads that are co-running on the SSD. We choose TPC-C as our test case for it is comprehensive to characterize the performance of a storage device. We measure tpmC, and the 99th percentile read latency after proper warm-up (until blocking I/O and GC activities tend to stabilize). Figure 4.14 and Figure 4.15 show the throughput and tail latency of evaluated schemes normalized to solely running TPC-C on vanilla SSD, respectively.

ZFTL exhibits an average of 50.8% leading in throughput and up to 4.85x lower tail latency compared to the Baseline system. As Baseline performance gets worse in write-intensive co-located workloads due to increased blocking I/O requests, ZFTL, in contrast, delivers similar performance metrics over various combinations, firmly holding on to performance independence. This is mainly because our physical isolation scheme

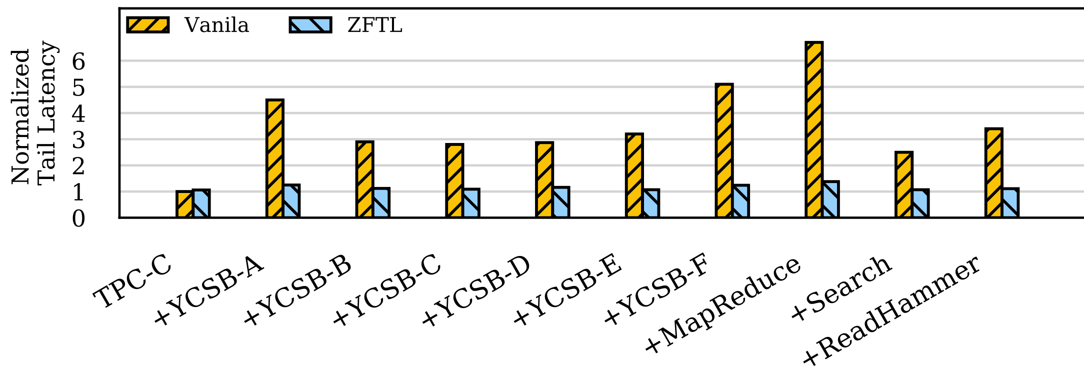


Figure 4.15: 99th percentile latency of Baseline, ZFTL scheme normalized to solely running TPC-C on Baseline system.

ensures that the ongoing I/O requests in other pFTL instances cannot directly impact the QoS on the pFTL hosting TPC-C.

We also notice, when YCSB-C and other read-only workloads present as co-running neighbors, the Baseline system shows significantly better performance than in other cases. The reason behind this performance variation is that when GC I/O is minimal, user I/O is well served by the SSD. This insight motivates us to evaluate how ZFTL behaves when the co-located pFTL is implementing extremely heavy GC operations. We also quantitatively evaluate the impact of split cache design in section Section 4.4.4.

To understand ZFTL performance under heavy GC, we compose an FIO workload issuing random writes requests (block size = 4 KB) and deploy four such containerized instances on the same drive. The drive is occupied 80% of its full capacity to ensure GC is always triggered (for ZFTL, each instance occupies 80% of its assigned capacity). The internal log from the simulator reveals that the SSD is reclaiming old blocks at around

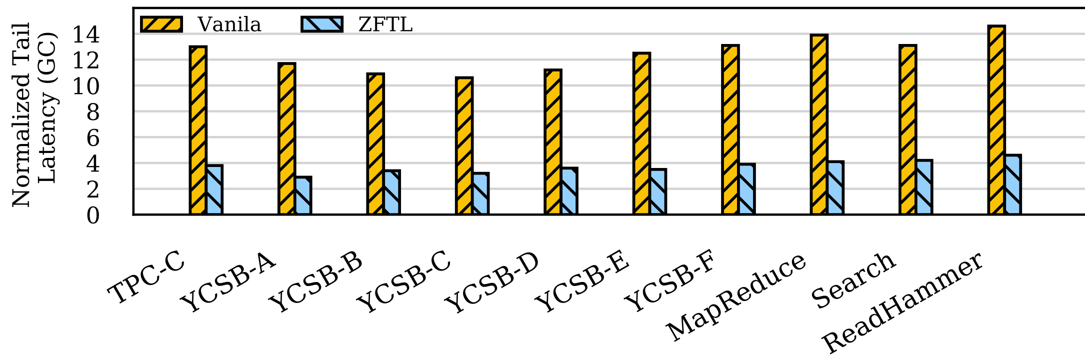


Figure 4.16: Normalized 99th percentile latency of Baseline, ZFTL scheme under heavy GC.

600 blocks per second, and I/O traffic incurred by GC is intensive. Figure 4.16 illustrates the 99th percentile latency of Baseline and ZFTL schemes normalized to that workload solely running on the Baseline system.

As it is even more likely for user I/O to be blocked by writes and block erasures incurred by heavy GC, the tail latency in the Baseline case may spike up to several ms in some combinations. This will likely lead to Out-of-Service (OOS) for response-critical I/O services. As for ZFTL, although I/O requests in a pFTL are decoupled from other GC-heavy pFTLs, ZFTL still encounters worse latency numbers due to increasing I/O rates, which causes contention on shared signal buses. Not surprisingly, the degradation is much more modest than the Baseline case. Overall, ZFTL can deliver a stable QoS in a variety of workload combinations, and it still improves tail latency even in a scenario with intensive GC activities.

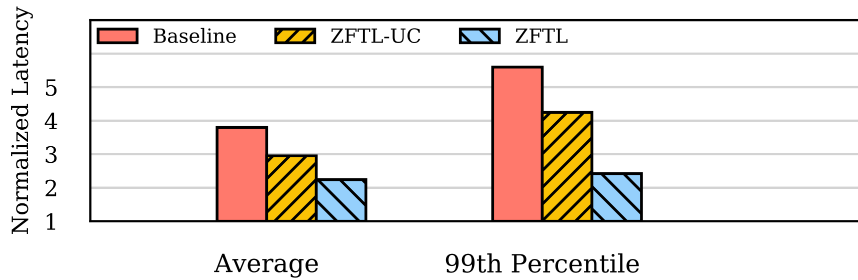


Figure 4.17: Normalized average and 99th percentile latency of Baseline, ZFTL-UC and ZFTL scheme.

4.4.4 Impact of Split Cache

In this section, we quantify the impact of split cache design in achieving low latency overhead. We configure a scheme in which all pFTLs share a unified cache (ZFTL-UC) while other enhancements remain unchanged. We evaluate Baseline, ZFTL-UC, and ZFTL in a scenario of co-running a containerized FIO 4K random read and MapReduce (throughput-intensive). Figure 4.17 plots the average and 99th percentile FIO read latency (normalized to solo latencies) for each scheme. As the internal buffer is filled with incoming write requests from MapReduce, ZFTL-UC reports a 36.4% slowdown on average. The 99th percentile latency is 2.25x higher than ZFTL. It is worth noting that even if we double the size of the DRAM buffer, the latencies show minor improvements (within 11%), as the shared cache is still quickly drained by write buffering from MapReduce. This observation further proves that split cache is needed to achieve low latency overhead as the ongoing write activities in other pFTL instances are unpredictable.

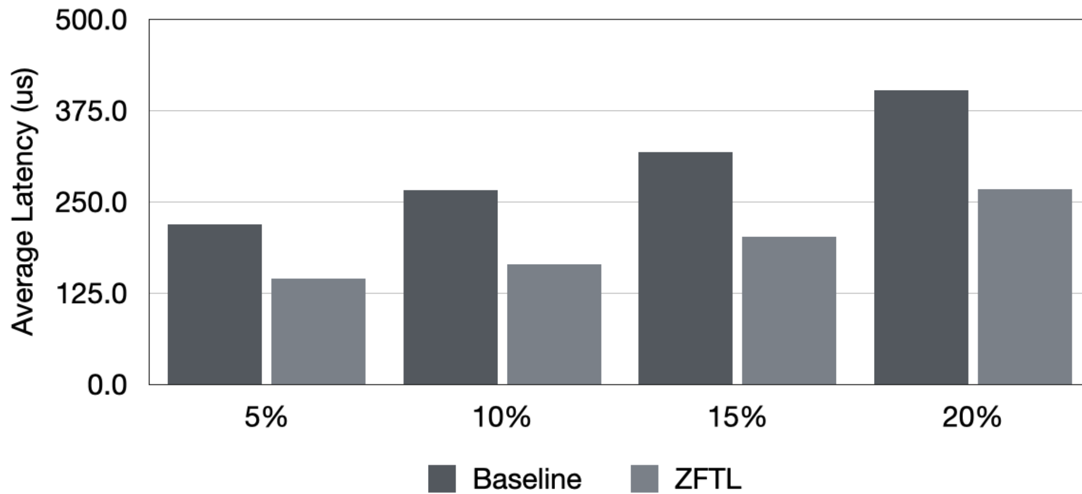


Figure 4.18: The average response time under various GC watermark.

4.4.5 Sensitivity to GC Intensity

Since Garbage collection interference is a major source of interference, the performance of ZFTL is sensitive to GC parameters, such as GC watermark, which is the threshold on the ratio of available free blocks within a flash chip. If actual number drops below the threshold, GC operation within the flash chip is invoked. The higher this number, more frequently GC operations will be invoked, which in turn exacerbates the GC-induced interference. Figure 4.18 shows the average response time of FIO 4k random read (same experimental config as Section 4.4.4) under 5%, 10%, 15% and 20% watermark. Generally, the response time increases sharply with higher watermark value. When required by 20% GC watermark, ZFTL average latency is 37% shorter than the baseline case. This is largely due to the GC isolation mechanism in ZFTL.

4.5 Summary

In this paper, we investigate the root causes of storage performance interference in a datacenter-like scenario. Due to the lack of resource isolation scheme in modern SSD controllers, user applications are likely to be interfered with by other co-running I/O services. To this end, we proposed ZFTL to offer strong physical isolation among multiple virtualized services via hardware virtualization. We implemented a prototyped ZFTL on an SSD emulator. Our evaluations show that ZFTL can improve throughput by 1.51x and reduce tail latency by up to 4.9x while only slightly impacts the raw performance of the SSD.

Chapter 5: Approaching More Effective I/O Control

Containers are gaining more popularity for virtualization capacity in modern datacenters. Ensuring resource isolation for container consolidation is a fundamental requirement in cloud environments. While Chapter 4 offers a framework to alleviate the performance interference in the underlying storage device, they couldn't address I/O contention in the kernel stack. In principle, the kernel I/O control needs to provide I/O resources to containers according predefined targets. In this chapter, we illustrate that the state-of-the-art resource control mechanism, Linux Control Groups (cgroup) is insufficient for controlling I/O resources. We reveal that inappropriate setups of cgroup may even hurt the performance of co-located workloads under I/O or memory intensive scenarios. To address the problem, we examine the entire Linux I/O stack, and add kernel support for limiting page cache usage per cgroup; we also implement a new page allocation/reclamation scheme based on I/O priority and weight.

5.1 Deep Dive into I/O Stack

5.1.1 Linux Block I/O Stack

In Linux, a block device is a hardware abstraction [31], whose data is stored and accessed in fixed size blocks of n bytes. The Linux block I/O stack is composed of many layers [3, 31, 218], Figure 5.1 illustrates a generic Linux I/O stack, from user applications to the underlying storage device. Starting from the top, user applications interact with the kernel through system calls [169]. For synchronous blocking I/Os, the application is blocked in the current thread until the kernel notifies the application that the request is complete.

The Virtual File System (VFS) layer provides an abstraction of different types of file systems [171, 217, 279]. It redirects any I/O operation from user space into specific implementation of the file system. VFS enables applications to read and write to different filesystems through standard Linux system calls. VFS does not know any implementation details of filesystems, the filesystems are programmed to provide the abstracted interfaces that VFS expects. For example, consider an user program performs `write(buf, len, file)` (writes `len` bytes data from buf into the current position in the file pointer). This system call is handled by invoking the writing method for the filesystem on which file resides. In other words, the filesystem actually writes the data to the media. A filesystem manages files, directories, control information and metadata (such as access permissions, size, owner, creation time, and so on). Metadata is stored in a separate data structure from the file, called the `inode`. The VFS layer updates file metadata stored in `inode` [217], a data structure which contains pointers to data blocks with file contents. Besides `inode`,

VFS caches other two types of metadata, superblock and dentry. A dentry metadata mainly contains file name and corresponding inode number, while a superblock metadata mainly contains specific filesystem info. From here, direct I/Os are directly dispatched to the driver without going through other software layers. Buffered I/Os have several more steps in the page cache layer and block I/O layer: on a page read, if the data is in the page cache, the kernel can service the requested page from memory; if not, the kernel goes down to lower kernel layers to access the storage medium and bring the requested page into the memory. On a write request, the kernel updates the page in the cache with new data. The modified block may be sent immediately to the block device, or it turns into dirty state depending on the synchronization policy. The dirty pages of a process are required to be flushed to persistent storage either by the process itself or the kernel cache flushing thread.

At block I/O layer, a bio data structure is created, which is the basic container for block I/O within the kernel. The bio structure contains an array of entries that point to individual memory pages that the block device needs to read/write (it contains a pointer called bi_io_vec pointing to an array of bio_vec structures, which represents the multiple segments, defined in <linux/bio.h>). Note segments are contiguous chunks of a block in memory. The block I/O layer maintains request queues to store pending block I/O requests to physical devices. The request queue struct request_queue is defined in <linux/blkdev.h>. An individual request in the queue is represented by struct request, also defined in <linux/blkdev.h>. Each request can be composed of multiple bio structures. New I/O requests are submitted to the tail of the request queue by filesystems and the device driver fetches requests from the queue

head. However, this conventional request queue has become the biggest bottleneck in the entire I/O subsystem [41] in the million-IOPS era because it operates on a single queue and spin-lock. To address this bottleneck, Linux kernel introduced the multi-queue block layer (blk-mq) that adopts multiple separate queues where each queue is bounded to a CPU core. This helps to eliminate contentions on the single queue. The blk-mq moves requests from per-core submission queues into the hardware queues up to the maximum number specified by the driver. Requests in the submission queue can be reordered or merged to improve I/O efficiency (reordering helps locality and priority, and merging can reduce the total number of requests). Note request merging can't be performed across queues since the submission queues are per-CPU. The block layer supports various hot-plug I/O schedulers, including none (simple FIFO), mq, kyber (aims at meeting certain latency targets by limiting the queue-depth dynamically) and BFQ. BFQ is the only non-trivial I/O scheduler that aims at providing fairness between I/O issuing processes. Without I/O scheduling, the submitted requests are directly dispatched to hardware queues of the block device (e.g., NVMe submission queues). As discussed in Chapter 4, the conventional I/O request reordering and merging does not take I/O contexts (timestamp, ownership, QoS requirement) into account, causing a widening semantic gap between underlying storage and kernel I/O management. The block layer also provides an interruption handler to deal with I/O completions: each time the device driver finishes an I/O request, it routes the interrupt to the block layer interruption handler [218]. The block layer then calls the I/O completion function in the libaio library, or returns from the synchronous read or write system call, which notifies the application of the completion signal. The NVMe driver sits underneath blk-mq, which supports deep per-core NVMe

queues [49]. The NVMe driver interacts with the block device through doorbell registers to handle I/O command submission & completion and interrupts. Every request sent by the NVMe driver is encapsulated in `nvmerw` format and goes through the PHY PCIe layer [21, 180]. Subsequently, NVMe SSDs can pull command and data in host memory region through DMA by parsing the PRP (Physical Region Page) field embedded in the NVMe request [180]. When an I/O request is completed, it sends a message signaled interrupt (MSI) that directly writes the interrupt vector of each core's programmable interrupt controller.

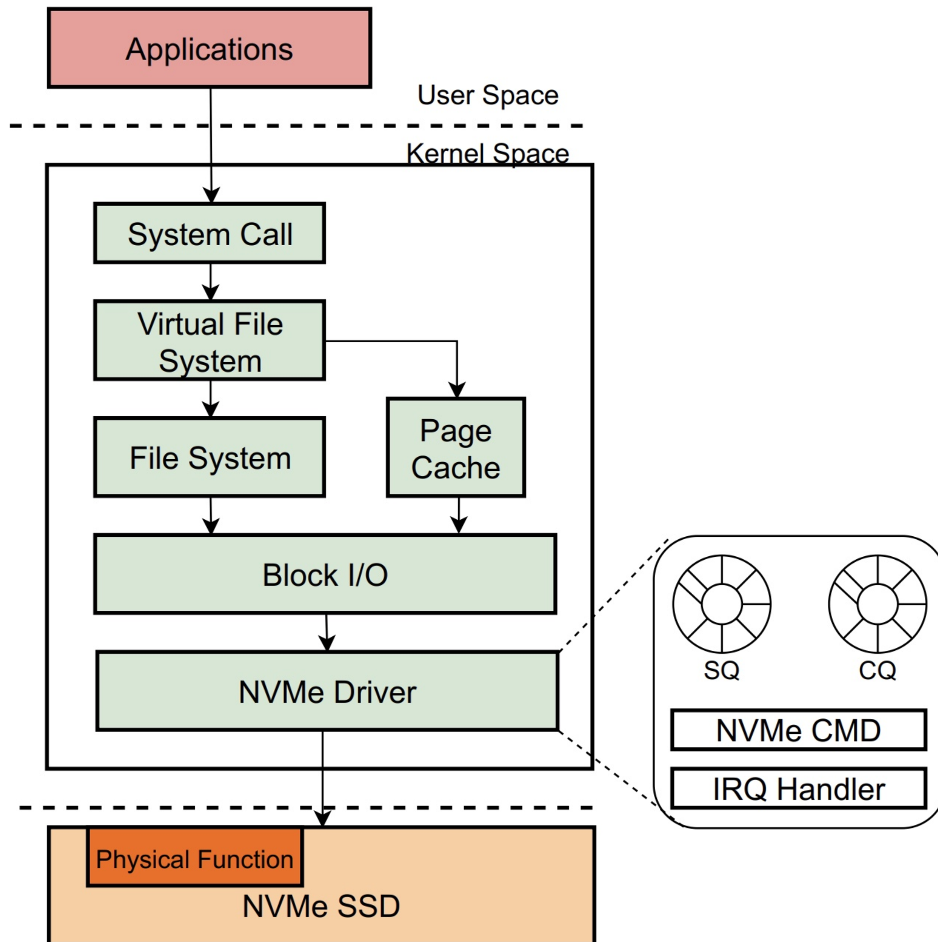


Figure 5.1: Overview of a Linux I/O stack.

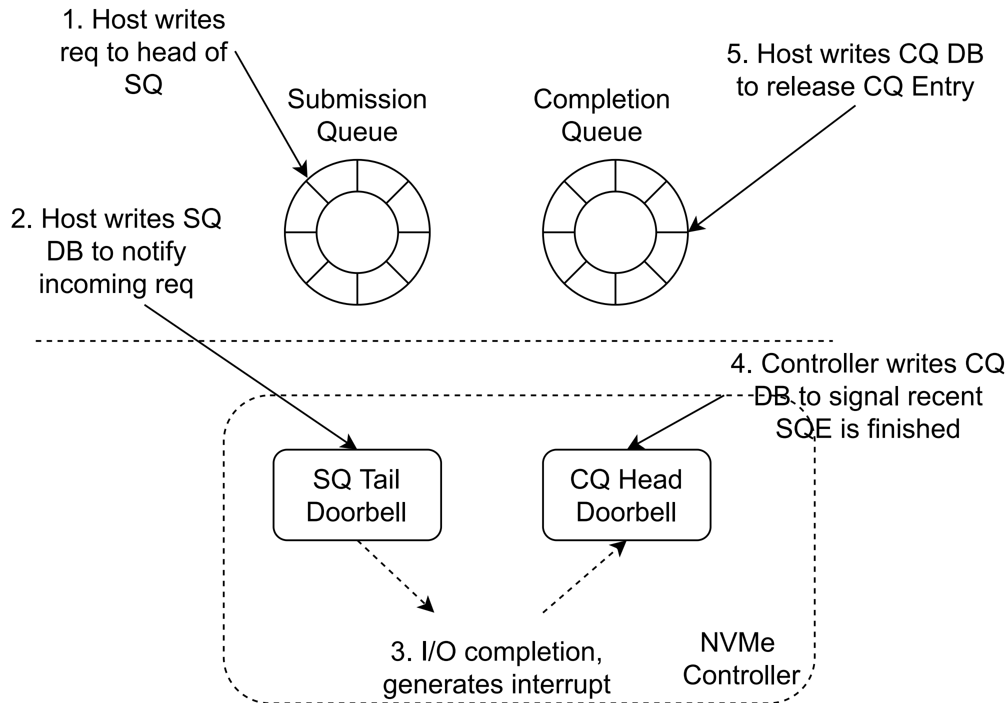


Figure 5.2: Host interacting with an NVMe device.

5.1.2 Software is a Thick Layer

Over the past years, many new flash technologies, such as ultra-low latency flash memory and phase-change memory, have been adopted in emerging SSDs. These storage devices, such as Samsung Z-SSD [8], Intel Optane SSD [1] achieve a few microseconds of access latency and gigabytes per second of bandwidth. With these ultra-low latency SSDs. the kernel storage stack is now becoming a bottleneck for these new devices [106, 200, 283]. To figure out the fraction of I/O access latency that is attributed to the device hardware and kernel software, we conduct an inspiring experiment which breakdowns the total access latency and present time spent in each software layer in Table 5.1, from user space into the storage device. we measure the average latency of the different software

layers when issuing a random 64 byte read system call with O_DIRECT flag on an Intel Optane SSD 900P. The detailed machine specification is listed in Table 5.2, running on Ubuntu 20.04 with Linux kernel 5.8.0. We lock the core frequency to 4.5GHz to eliminate I/O performance fluctuations from the processor side. Figure 5.3 presents the request dispatch path in the regular storage software stack. The results show that the software overhead is already measurable on the first-gen Optane SSD, accounting for 22.5% of the total latency. This ratio is only going higher as storage devices provide sub-ten microsecond of access latency and the software stack is unable to scale to handle more I/O requests. The software stack is thick, and poses a substantial overhead on every I/O request. There are some kernel-bypass frameworks (e.g. SPDK [270]) proposed to eliminate the heavy tax paid in the software stack (including syscall, fs and bio layer) by directly submitting requests to the device. However, kernel bypass also brings application design challenges, as they have to access/manage data on raw devices and I/O isolation and polling efficiency challenges.

Table 5.1: Average latency breakdown of a random read syscall

Overhead Source	**Time Spent**
syscall	520 ns
ext4	1840 ns
bio	492 ns
NVMe Driver	114 ns
Device	10240 ns
Total	13.2 us

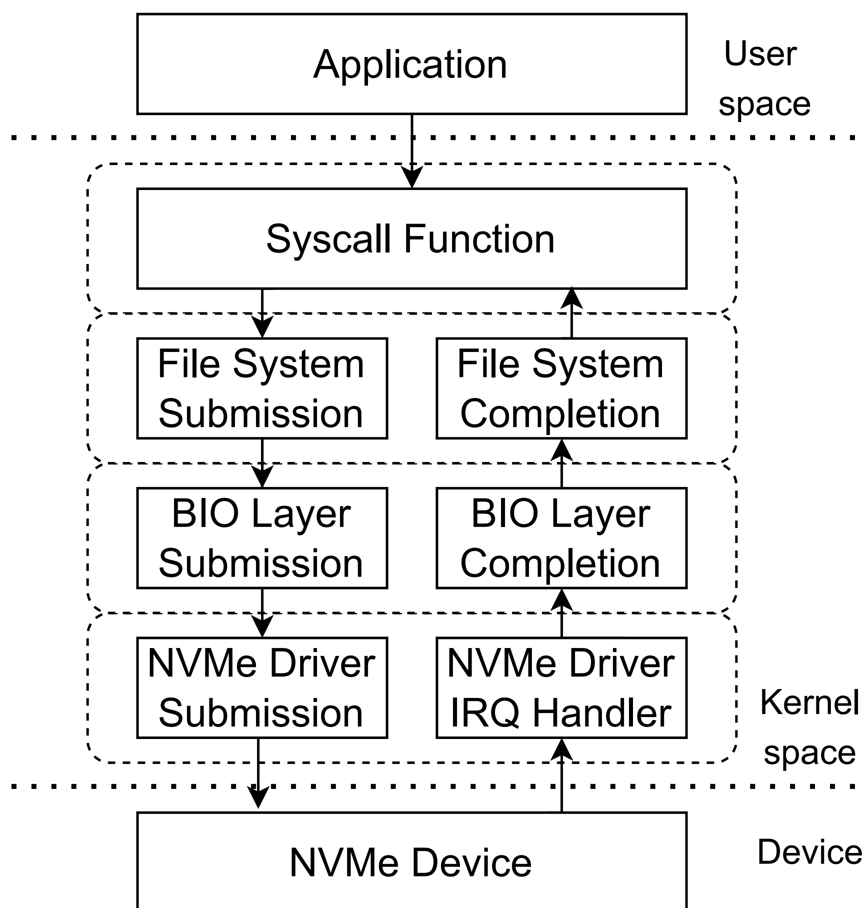


Figure 5.3: Request path in a regular storage software stack.

5.1.3 Page Cache

The page cache is a set of data structures that contain recent accessed files from filesystem files, block device files, or memory-mapped files [95]. Practically all I/O operations rely on the page cache unless they are direct I/Os (with `O_DIRECT` flag enabled) [204]. Buffered I/O requests are served by page cache, while direct I/O requests are served directly from the underlying storage device. Buffered I/O requests interact with the page cache layer in the following way: when filesystem performs a `readpage()`, the kernel

first searches the page cache by `find_get_page(mapping, index)` function. Upon a cache miss, `find_get_page()` return `NULL`, a new page is allocated with `page_cache_alloc()` and added to the page cache with `add_to_page_cache`. After the page exists in the page cache, `page_cache_readahead()` is called, which uses `page_cache_read()` to reads the page from disk. For write requests, things are a little different. Linux adopts the write-back strategy, which updates the data in the cache first and updates the data on disk eventually. When Write requests update the cached data, the updated pages are marked as dirty and added to the dirty list. A process then periodically updates the blocks corresponding to pages in the dirty list they are cached into the page cache, usually written back to the disk eventually and asynchronously. Page cache is a critical component in any modern operating system since accessing pages cached in memories is orders of magnitude faster than accessing the storage device.

A page in the page cache can consist of multiple noncontiguous physical disk blocks. The Linux page cache uses an `address_space` structure to manage entries in the cache and page I/O operations. The `address_space` structure is defined in `<linux/fs.h>`, as shown below.

```
struct address_space {
        struct inode *host;      //Owner, either the inode or the block_device
        struct xarray i_pages;       //Cached pages
        struct rw_semaphore invalidate_lock;     /* Guards coherency between
        page cache contents and file offset->disk block mappings in the
        filesystem during invalidates */
        gfp_t gfp_mask;      //Memory allocation flags to use for allocating pages
        atomic_t i_mmap_writable;     //Number of VM_SHARED mappings
        struct rb_root_cached i_mmap;    //List of all mappings
        struct rw_semaphore i_mmap_rwsem;    //Protects i_mmap and i_mmap_writable
```

```
        unsigned long nrpages;    //Number of page entries
        pgoff_t writeback_index;    //Writeback starts here
        const struct address_space_operations *a_ops;   //Operations table
        unsigned long flags;
        errseq_t wb_err;
        spinlock_t private_lock;
        struct list_head private_list;
        void *private_data;     //Associated buffers
};
```

Pages in the page cache should be quickly found. To facilitate this, the `i_mmap` field, which is a priority search tree of all shared and private mappings, allows the kernel to efficiently find the mappings associated with this cached file. This address space contains a total of `nrpages`. The `a_ops` field points to the address space operations table, also defined in `<linux/fs.h>`. These function pointers point at the functions that implement page I/O for this cached object. Each filesystem defines its concrete interactions with the page cache via its own `address_space_operations`.

```
struct address_space_operations {
        int (*writepage)(struct page *page, struct writeback_control *wbc);
        int (*readpage)(struct file *, struct page *);

        /* Write back some dirty pages from this mapping. */
        int (*writepages)(struct address_space *, struct writeback_control *);

        /* Mark a folio dirty.  Return true if this dirtied it */
        bool (*dirty_folio)(struct address_space *, struct folio *);

        void (*readahead)(struct readahead_control *);

        int (*write_begin)(struct file *, struct address_space *mapping,
                                loff_t pos, unsigned len, unsigned flags,
```

```
                                struct page **pagep, void **fsdata);
        int (*write_end)(struct file *, struct address_space *mapping,
                                loff_t pos, unsigned len, unsigned copied,
                                struct page *page, void *fsdata);

        /* Unfortunately this kludge is needed for FIBMAP. Don't use it */
        sector_t (*bmap)(struct address_space *, sector_t);
        void (*invalidate_folio) (struct folio *, size_t offset, size_t len);
        int (*releasepage) (struct page *, gfp_t);
        void (*freepage)(struct page *);
        ssize_t (*direct_IO)(struct kiocb *, struct iov_iter *iter);
        /*
         * migrate the contents of a page to the specified target. If
         * migrate_mode is MIGRATE_ASYNC, it must not block.
         */
        int (*migratepage) (struct address_space *,
                        struct page *, struct page *, enum migrate_mode);
        bool (*isolate_page)(struct page *, isolate_mode_t);
        void (*putback_page)(struct page *);
        int (*launder_folio)(struct folio *);
        bool (*is_partially_uptodate) (struct folio *, size_t from,
                        size_t count);
        void (*is_dirty_writeback) (struct page *, bool *, bool *);
        int (*error_remove_page)(struct address_space *, struct page *);

        /* swapfile support */
        int (*swap_activate)(struct swap_info_struct *sis, struct file *file,
                                sector_t *span);
        void (*swap_deactivate)(struct file *file);
};
```

`<mm/filemap.c>` provides APIs to manipulate the page cache, as shown below.

```
/* Example APIs for page cache manipulation */
int add_to_page_cache_lru(struct page *page, struct address_space *mapping,
```

```
                                    pgoff_t offset, gfp_t gfp_mask)
/* This function adds a page to the pagecache and the LRU list */

struct page *pagecache_get_page(struct address_space *mapping, pgoff_t index,
                int fgp_flags, gfp_t gfp_mask)
/* This function finds and geta a reference to a page */

void delete_from_page_cache(struct page *page)
/* This function deletes a page from the page cache and free it */

void replace_page_cache_page(struct page *old, struct page *new)
/* This function replaces an old page from the page cache with a
new page */
};
```

Page allocation and page eviction are two core functions in the page cache. They determine when and which page is added to or removed from the page cache, respectively. Their implementations are crucial to I/O performance [86, 252], especially in a multi-tenancy environment where multiple I/O processes access and update the page cache concurrently. Page eviction happens under three cases: (1) cache is full (2) kernel shrinks cache area under memory pressure (3) dirty pages are written back to underlying block device periodically. Linux utilizes a variant of LRU (Least Recently Used) algorithm, which consist of two lists called `active_list` and `inactive_list`, declared in `mm/page_alloc.c`. The `active_list` contains frequently accessed pages, `inactive_list` contains reclaim candidates. Pages in the `active_list` are considered hot and will not be evicted. When a page is first accessed, it will be placed at the head of `inactive_list`. If the page is accessed again, it will be inserted at the tail of `active_list`. Both LRU lists are maintained in a pseudo-LRU manner (add to the tail, remove from head). When caches

are being shrunk, pages are moved from the active_list to the inactive_list by refill_inactive(). refill_inactive() takes a parameter of the number of pages to move, which is calculated in shrink_cache() as a ratio depending on nr_pages, the number of pages in active_list and the number of pages in inactive_list. shrink_cache() takes pages from the inactive_list and decides which pages should be swapped out. Two parameters nr_pages and priority determine the number of pages to be swapped out. Each time shrink_cache() is called without freeing enough pages, the priority will be decreased until the highest priority 1 is reached. The maximum number of pages that will be scanned during the reclamation process is determined by max_scan. The API that deals with the LRU lists is defined in <mm/vmscan.c>. A kernel thread called kswapd (defined in <mm/vmscan.c>) is responsible for reclaiming dirty pages when memory is below low threshold, and it keeps freeing pages until the high watermark is reached.

During the page eviction process, pages are treated equally for insertion and eviction, without considering all the niceties such as fairness, adaptive pausing, bandwidth proportional allocation and configurability. This can lead to insufficient control to target I/O applications. A fundamental flaw is that the conventional page cache management functions do not limit the page cache usage, or throttle dirty page write-back based on user configurations. Additionally, the page cache usage of a cgroup is indirectly managed by the memcontroller module. We demonstrate that current designs are insufficient to achieve effective I/O control in a multi-container environment in the following sections.

```
/* APIs for LRU list management, mm/vmscan.c */
void shrink_active_list(unsigned long nr_to_scan,
```

```
                                        struct lruvec *lruvec,
                                        struct scan_control *sc,
                                        enum lru_list lru)
/* This function moves pages from the active LRU to the inactive LRU */

unsigned long try_to_free_mem_cgroup_pages(struct mem_cgroup *memcg,
                                           unsigned long nr_pages,
                                           gfp_t gfp_mask,
                                           bool may_swap)
/* This function tries to free (soft reclaim) pages from a certain
memory cgroup */

static unsigned long
shrink_inactive_list(unsigned long nr_to_scan, struct lruvec *lruvec,
                     struct scan_control *sc, enum lru_list lru)
/* This function shrinks the inactive LRU list and returns the number
of reclaimed pages */

void del_page_from_lru_list(struct page *page,
                            struct lruvec *lruvec)
```

5.1.4 Block I/O Scheduler

Block devices have request queues to schedule pending read or write requests. Since directly sending `bio` requests to the block device results in poor performance, Linux kernel deploys block I/O schedulers to sort and merge requests before dispatching them to the block device, for the target target of improving system throughout and I/O quality of service. Merging means merging two or more requests into a single request, which helps reduce addressing overhead and improve throughput. Sorting means prioritizing I/O requests in some way, up to the scheduler policy. There are four major block I/O

schedulers in the latest Linux kernel: NOOP, mq-deadline, BFQ, and kyber. In this section, we briefly discuss their implementation and evaluate their system overhead. Note some of the schedulers were designed for mechanical disks and they may not be able to take advantage of the internal parallelism and low latency of modern NVMe SSDs. All of them are targeted to achieve certain performance properties, e.g., preventing asynchronous writes from starving synchronous reads.

5.1.4.1 NOOP Scheduler

The NOOP scheduler [205] does not sort requests before inserting to the queue. It simply maintains a single request queue that takes all incoming I/O requests from all processes running on the system, regardless of the type and urgency of the I/O request. The request queue does not perform additional sorting but simply follows a first-in-first-out (FIFO) policy. The NOOP scheduler does request merging by taking adjacent requests and merging them into a single request to improve throughput. Figure 5.4 shows the simplified diagram of NO-OP scheduler. NOOP assumes that the block device will further schedule the I/O requests and optimize the performance, such as what NVMe device does in its controller. For this reason, NOOP is the default scheduler for NVMe devices that favors random access.

5.1.4.2 BFQ Scheduler

The full name of BFQ scheduler [251] is Budget Fair Queueing (BFQ) I/O Scheduler. Compared to its predecessor CFQ which allocates time slices for each process to access

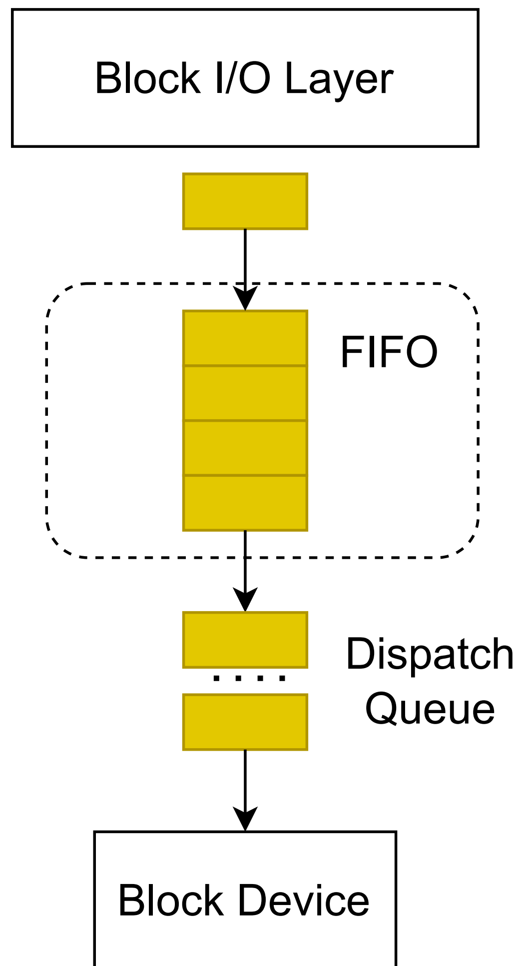


Figure 5.4: Diagram of NOOP scheduler.

the disk, BFQ enables an equal-share round-robin I/O scheduling algorithm according to sectors read/written. It allows proportional I/O control by assigning a sector budget to each request. However, it does not take the complicated interaction with memory management module into account, which may result in isolation failure due to memory interference. Figure 5.5 shows the simplified diagram of NO-OP scheduler.

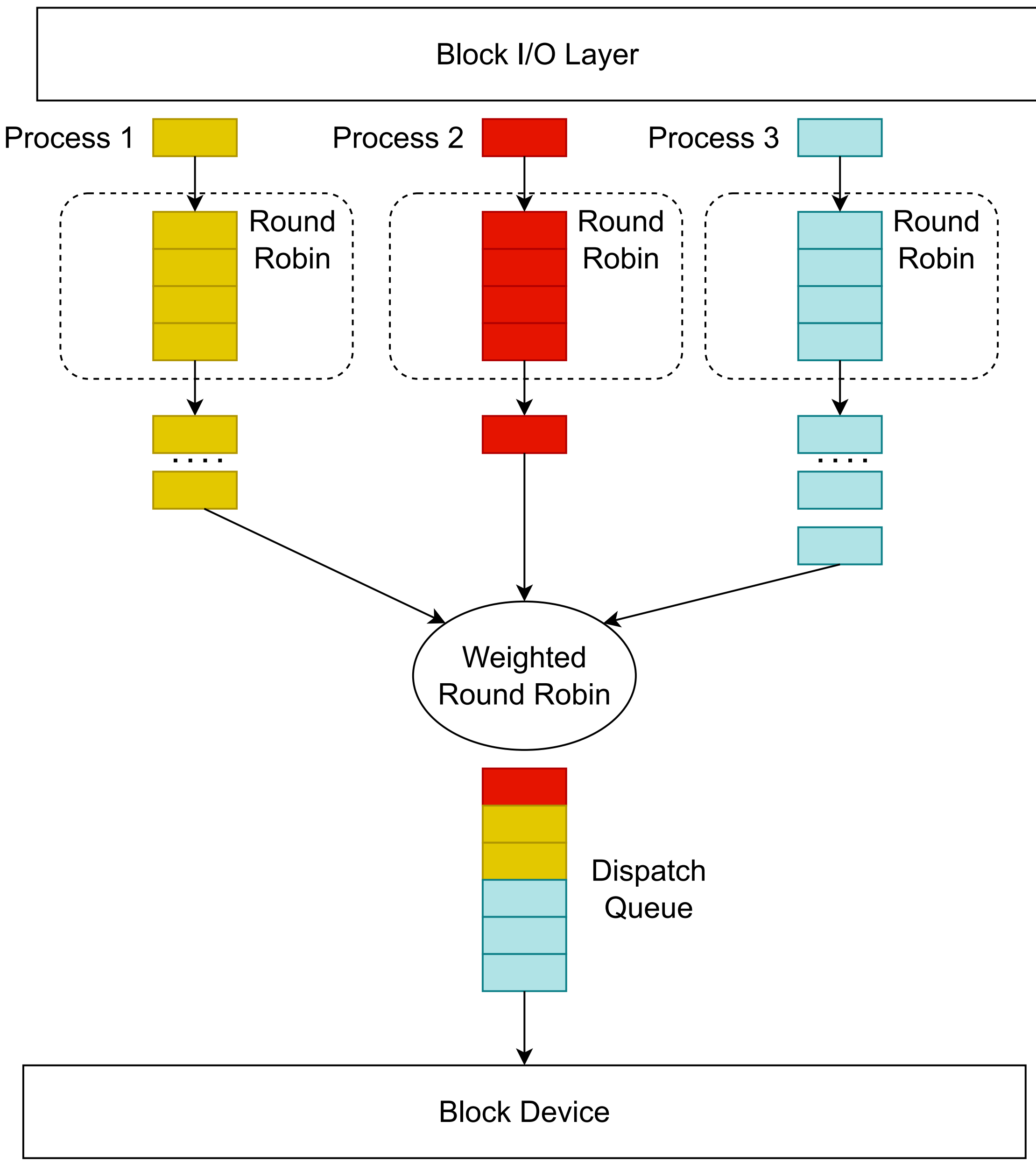


Figure 5.5: Diagram of BFQ scheduler.

5.1.4.3 Deadline Scheduler

The Deadline scheduler [156] maintains two deadline queues in addition to the sorted queues for reads and writes. The deadline queues are sorted by their deadline times (time to expiration), with shorter times moving to the head of the queue. The core idea of deadline scheduler is to reorder requests to improve I/O performance while ensuring no I/O request is being starved. Figure 5.6 shows the simplified diagram of Deadline scheduler.

5.1.4.4 Kyber Scheduler

Kyber is designed for fast multi-queue devices and features for its simplicity – it is implemented in less than 1k LOC (lines of code) [84]. The core idea is that the number of operations (regardless of reads and writes) sent to the dispatch queues is strictly limited, keeping those queues relatively short. This short dispatch queue design guarantees that the time spent in waiting in the queue is short. Kyber dynamically tunes the actual number of requests allowed into the dispatch queues by measuring the completion time of each request and adjusting the limits to achieve the desired latencies. Therefore, Kyber is often used to meet certain latency targets (by default 2 ms for reads, 10 ms for writes). Figure 5.7 shows the simplified diagram of Kyber scheduler.

5.1.4.5 Evaluations

We conduct performance analysis of the above mentioned schedulers. The experimental setup is listed in Table 5.2. We use synthetic benchmarks FIO to measure the performance

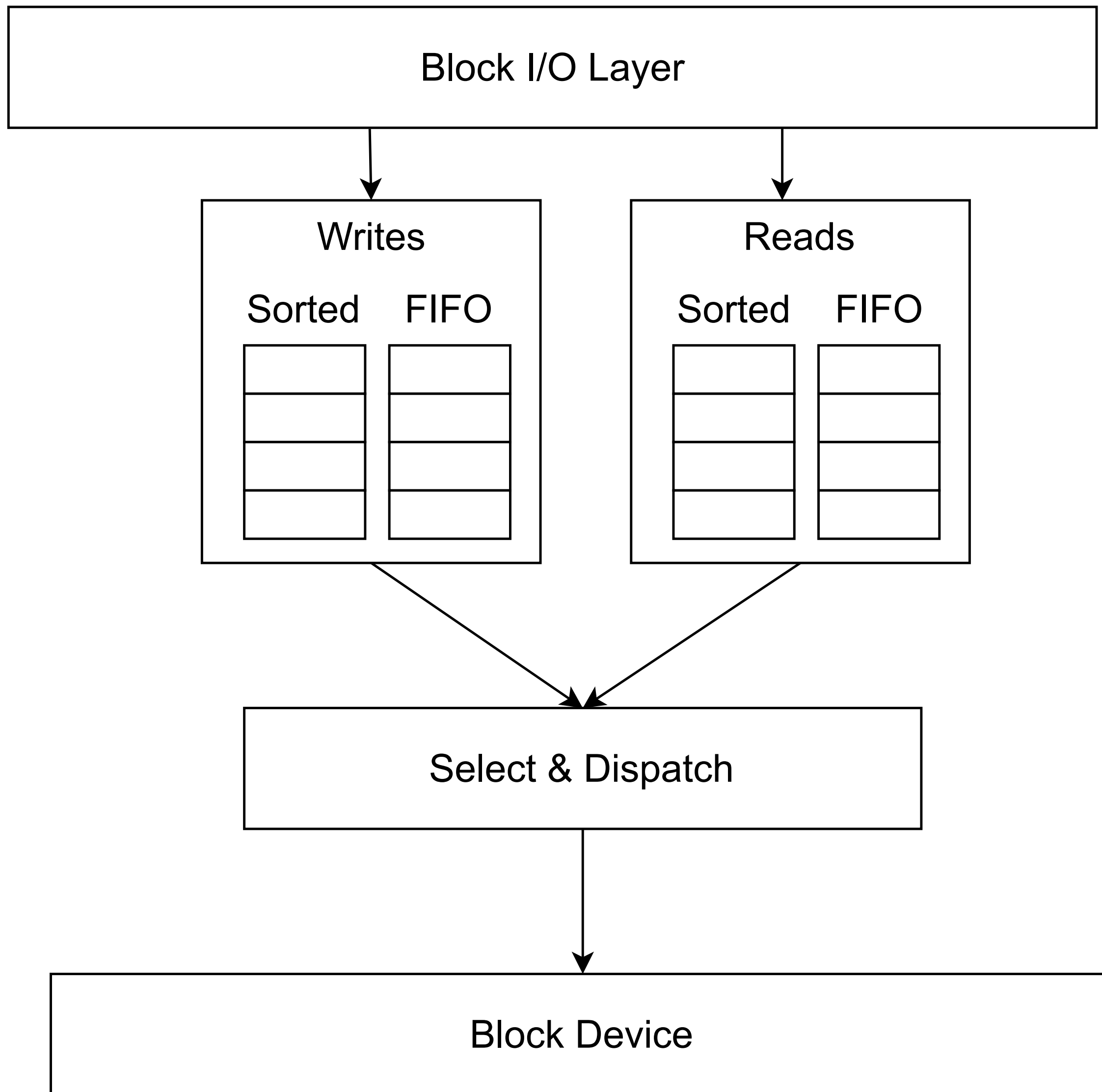


Figure 5.6: Diagram of Deadline scheduler.

of random reads, random writes, sequential reads, and sequential writes; to measure the performance of various I/O schedulers on real-world applications, we use the pgbench provided by PostgreSQL and facebook RocksDB [54, 186]. Figure 5.8, Figure 5.9 and Figure 5.11 present the results. In almost all cases, NOOP takes the lead, reads favor kyber, and Deadline, BFQ performs better in write-heavy workloads.

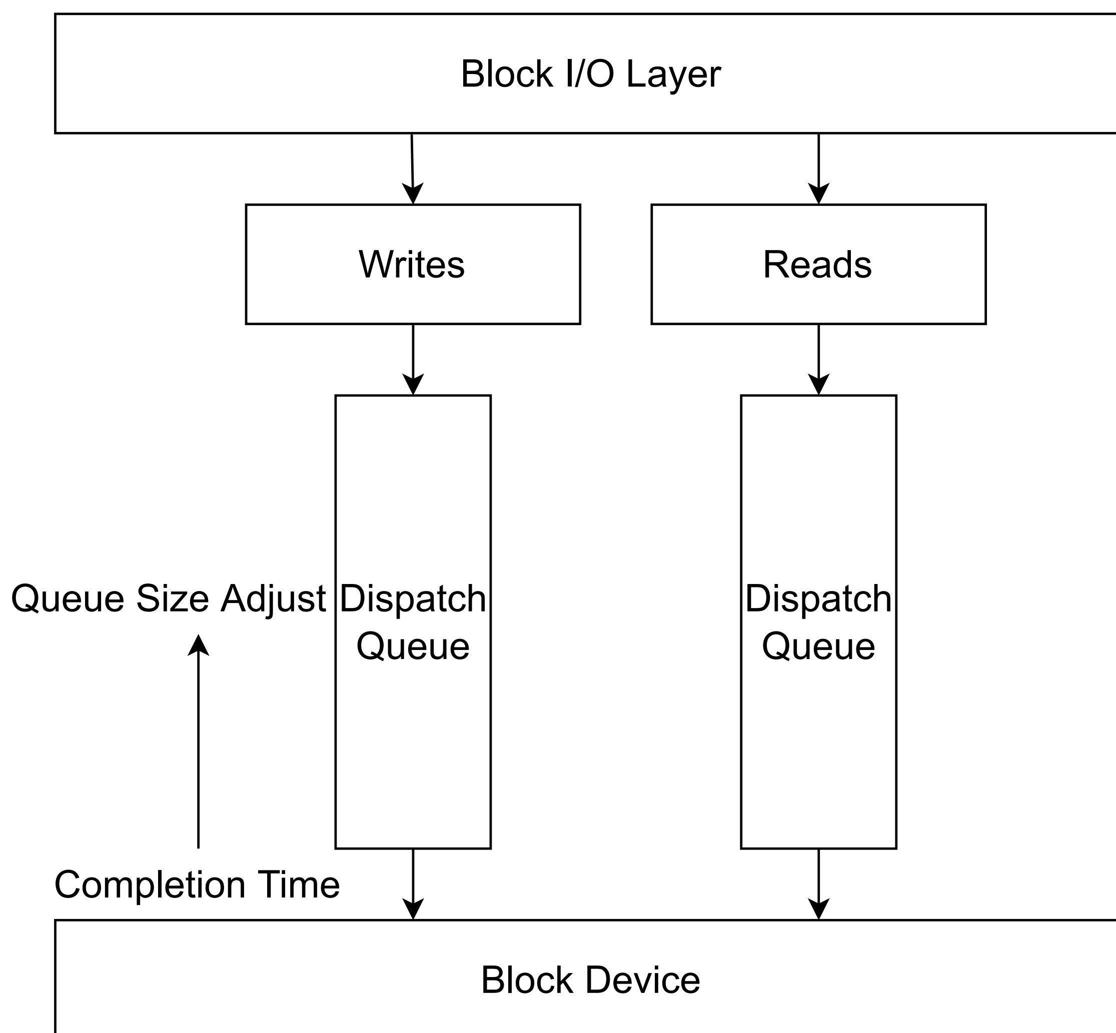


Figure 5.7: Simplified diagram of Kyber scheduler.

5.1.5 Linux Containers

Cloud computing is currently employing two virtualization technologies, Virtual Machine (VM) based approach and container-based approach. They aim to provide different levels of abstraction and different types of isolation. The VM-based approach virtualizes the entire OS by presenting to VM an abstraction of virtual devices. The

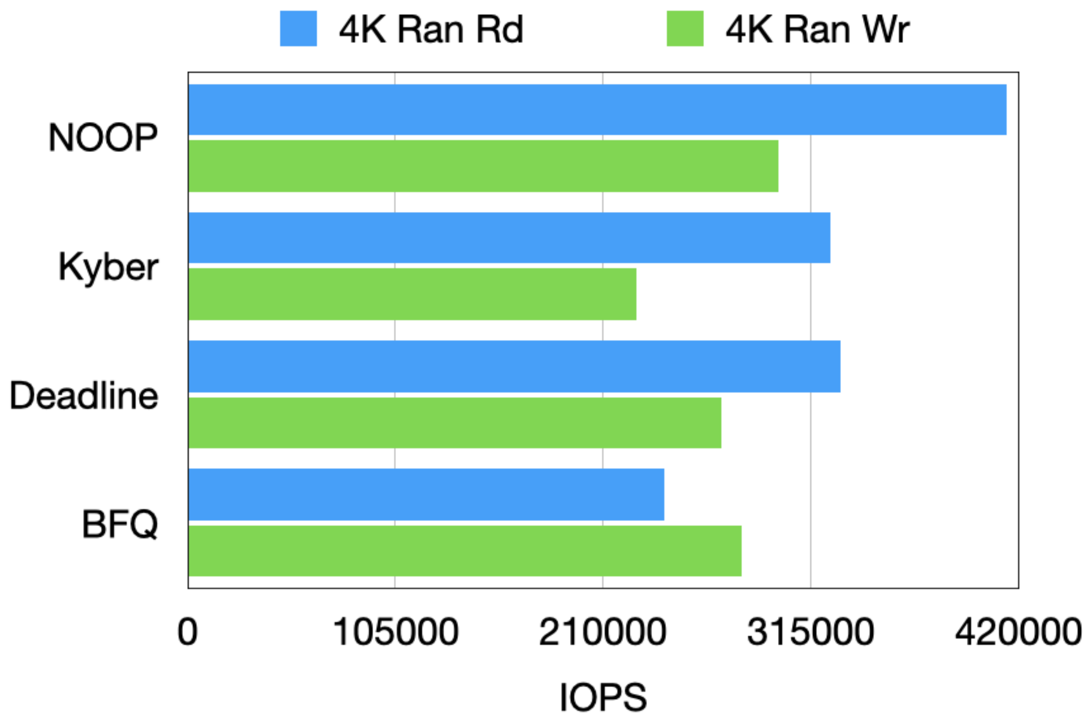


Figure 5.8: IOPS of FIO 4KB random reads and writes on different I/O schedulers.

OS hosted on the virtual machine can be different from the host OS, and this enables an independent ecosystem from the host system. Meanwhile, the OS hosted on the virtual machine thinks that it is the only OS running on the physical hardware, although in reality, the hardware is shared by multiple virtual machines and the host OS. The VM-based virtualization emulates the hardware and provides hardware-isolated virtual address space. In contrast, container-based virtualization creates isolated environments with kernel supports, containers do not emulate hardware but communicate with the hardware on the host system through the host kernel. Therefore, the resource footprint of containers is significantly smaller than VM-based virtualization as containers do not need to emulate hardware and host a full-fledged OS. These features allow container-based

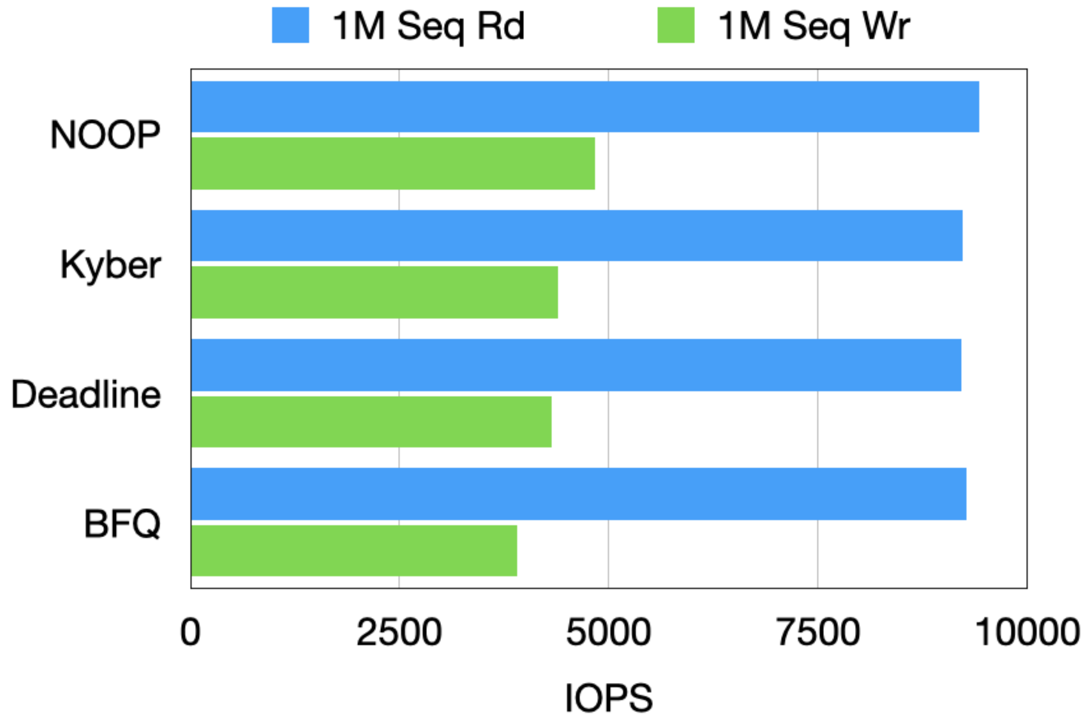


Figure 5.9: IOPS of FIO 1MB sequential reads and writes on different I/O schedulers.

virtualization better performance and scalability. Linux containers are consisted of three Linux kernel primitives for resource allocation and isolation: Linux namespaces, cgroups, layered file systems.

Linux namespace is an isolation mechanism within the Linux kernel designed to facilitate isolation between user space applications. A namespace controls which resources within the kernel a process can see. All the controls are defined at the process level. Only other processes within the same namespace can see the change of global resources. Following namespace types are available on Linux, each namespace has its own unique properties:

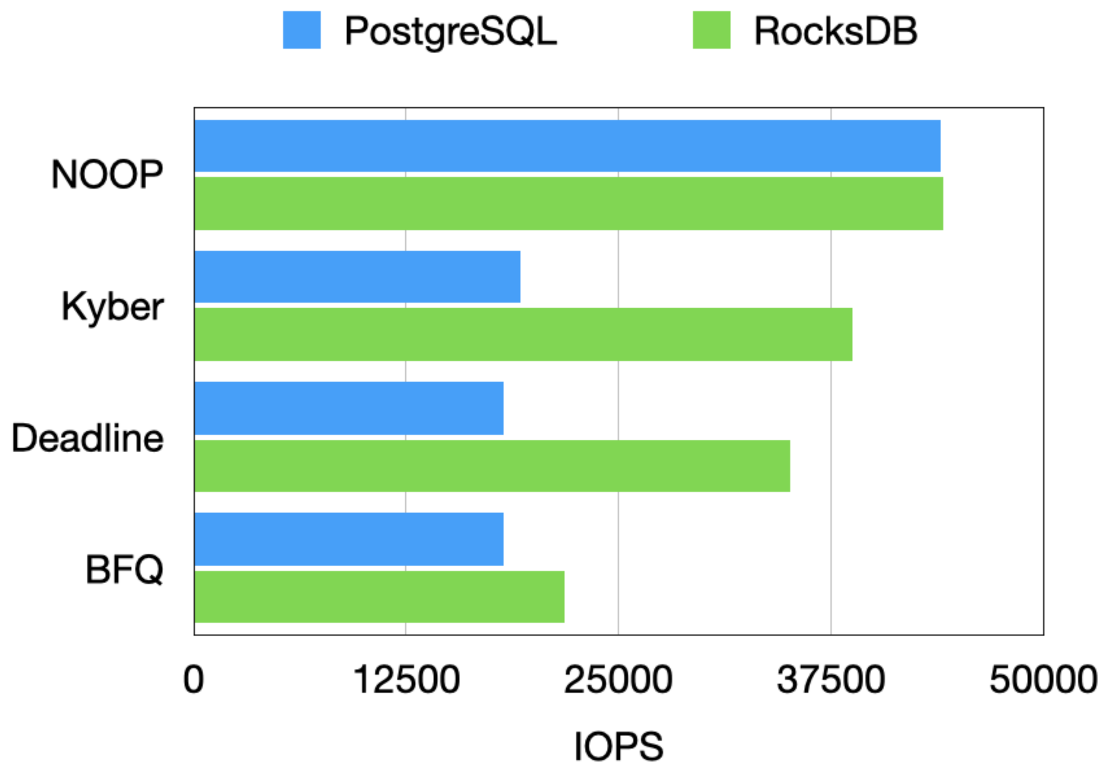


Figure 5.10: IOPS of PostgreSQL and RocksDB on different I/O schedulers.

- An **user** namespace has its own set of user IDs and group IDs for assignment to processes. In particular, this means that a process can have root privilege within its user namespace without having it in other user namespaces.
- A **process ID (PID)** namespace assigns a set of PIDs to processes that are independent from the set of PIDs in other namespaces.
- A **network** namespace assigns a separate set of network subsystems to a container, including different network interfaces, routes, and iptables.
- A **mount** namespace controls what mount points can be seen by processes. This

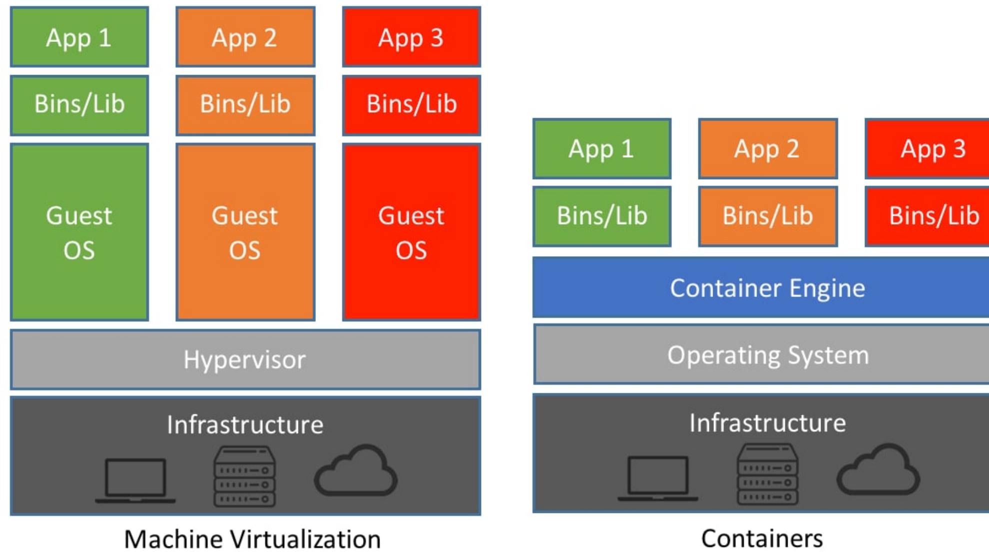


Figure 5.11: Comparison between virtual machine and container hierarchy.

means fileystems can be mounted and unmounted into a mount namespace without affecting the host filesystem.

- An **interprocess communication (IPC)** namespace has a set of independent IPC resources, for example POSIX message queues. Two processes within the same namespace can communicate over IPC, but two processes in two different namespaces cannot communicate over IPC.
- A **UNIX Time-Sharing (UTS)** namespace allows a process to have different host and domain names other than the global namespace.
- A **cgroup** namespace restricts the visibility of the cgroup file system to the process's owner cgroup.

Therefore, kernel namespaces is one of the key components to achieve process isolation, which is essential for implementing container-based virtualization.

Control groups, usually referred to as cgroups, is a Linux kernel feature which allows processes to be organized into hierarchical groups, and enforces resource controls for processes within the namespace. Besides configuring a cgroup to limit the specific amount of system resources it can use, users can also prioritize resource allocations compared to other cgroups when there is resource contention. The cgroup kernel module monitors and reports resource accounting at the cgroup level. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs. A new control group is created as a subdirectory in the file system tree and processes can then be moved to it. Initially all processes are added to the root of the cgroup hierarchy. Different subsystems, also called controllers, are responsible for configurations of different resources in cgroupfs. Users interact with all the controllers via regular Unix file operations. Over time, various cgroups control have been developed, such as CPU, network, memory, I/O and etc. We discuss two example cgroups here, CPU cgroup, and I/O cgroup. The CPU cgroup provides two types of CPU resource control: (1) `cpu.shares` contains an integer value that specifies a relative share of CPU time available to the tasks in a cgroup. (2) `cpu.cpuacct` specifies accounting for CPU usage by groups of processes. Block I/O cgroup (1)provides fairness to the individual cgroup or throttle I/O activities with upper limits per cgroup to specific block devices. The block I/O cgroup is defined in `<linux/blk-cgroup.h>`.

```
struct blkcg {
        struct cgroup_subsys_state        css;
```

```
        spinlock_t  lock;
        refcount_t        online_pin;

        struct radix_tree_root        blkg_tree;
        struct blkcg_gq        __rcu    *blkg_hint;
        struct hlist_head                blkg_list;

        struct blkcg_policy_data    *cpd[BLKCG_MAX_POLS];

        struct list_head    all_blkcgs_node;
#ifdef CONFIG_BLK_CGROUP_FC_APPID
        char     fc_app_id[FC_APPID_LEN];
#endif
#ifdef CONFIG_CGROUP_WRITEBACK
        struct list_head    cgwb_list;
#endif
};
```

`struct blkcg` represents the block I/O cgroup. Each block I/O cgroup is mapped to a request queue in the block I/O layer (discussed in last section) per device. The association between an block I/O cgroup and a request queue is defined in `<linux/blk-cgroup.h>`.

```
struct blkcg_gq {
        /* Pointer to the associated request_queue */
        struct request_queue        *q;
        struct list_head    q_node;
        struct hlist_node   blkcg_node;
        struct blkcg                *blkcg;

        /* all non-root blkcg_gq's are guaranteed to have access to parent */
        struct blkcg_gq     *parent;

        /* reference count */
        struct percpu_ref   refcnt;
```

```
        /* is this blkg online? protected by both blkcg and q locks */
        bool    online;

        struct blkg_iostat_set __percpu        *iostat_cpu;
        struct blkg_iostat_set  iostat;

        struct blkg_policy_data *pd[BLKCG_MAX_POLS];

        spinlock_t  async_bio_lock;
        struct bio_list async_bios;
        union {
                struct work_struct        async_bio_work;
                struct work_struct        free_work;
        };

        atomic_t    use_delay;
        atomic64_t  delay_nsec;
        atomic64_t  delay_start;
        u64     last_delay;
        int     last_use;

        struct rcu_head            rcu_head;
};
```

Cgroup v1 generally provides two I/O control policies (blkio). One is a proportional I/O share policy according to group weights implemented by the BFQ (Budget Fair Queueing) I/O scheduler. The BFQ scheduler assigns each group a queue and then gives a time slice to each queue, thereby handling fairness. When BFQ is applied to group level, the scheduling happens at the group level instead of at the process level. Each BFQ cgroup is associated with a weight and a queue. The BFQ cgroup paramaters can be configured within each group directory by `blkio.bfq`. BFQ grants exclusive access to the device

in a time-slice manner, and every cgroup queue is associated with a budget, measured in number of sectors. The BFQ group structure is defined in </linux/block/bfq-iosche.h>, each (device, cgroup) pair has its own bfq_group.

```
struct bfq_group {
        /* must be the first member */
        struct blkg_policy_data pd;

        /* cached path for this blkg */
        char blkg_path[128];

        /* reference counter */
        int ref;

    /* schedulable entity to insert into the parent group sched_data */
        struct bfq_entity entity;

        struct bfq_sched_data sched_data;

    /* the bfq_data for the device this group acts upon. */
        void *bfqd;

        struct bfq_queue *async_bfqq[2][IOPRIO_NR_LEVELS];

        /* array of async queues for all the tasks belonging to
       the group, one queue per ioprio value per ioprio_class,
       except for the idle class that has only one queue. */
        struct bfq_queue *async_idle_bfqq;

        struct bfq_entity *my_entity;

        int active_entities;

        struct rb_root rq_pos_tree;
```

```
        struct bfqg_stats stats;
};
```

The second is a hard limit on the bandwidth used or I/O operations per second (IOPS) per device. Most of the throttling functions are defined in `</linux/block/blk-throttle.c>`. Function `blk_throttl_bio` is a major one, which is responsible for throttling a specific `bio` according to its owner's limit. The following code snippet first checks if the `bio` can be dispatched to the device driver.

```
/* if above limits, break to queue */
if (!tg_may_dispatch(tg, bio, NULL)) {
        tg->last_low_overflow_time[rw] = jiffies;
        if (throtl_can_upgrade(td, tg)) {
                throtl_upgrade_state(td);
                goto again;
        }
        break;
}

/* within limits, charge and dispatch directly */
throtl_charge_bio(tg, bio);

static void throtl_charge_bio(struct throtl_grp *tg, struct bio *bio)
{
        bool rw = bio_data_dir(bio);
        unsigned int bio_size = throtl_bio_data_size(bio);

        /* Charge the bio to the group */
        if (!bio_flagged(bio, BIO_THROTTLED)) {
                tg->bytes_disp[rw] += bio_size;
                tg->last_bytes_disp[rw] += bio_size;
        }
```

```
        tg->io_disp[rw]++;
        tg->last_io_disp[rw]++;

        /*
         * BIO_THROTTLED is used to prevent the same bio to be throttled
         * more than once as a throttled bio will go through blk-throtl the
         * second time when it eventually gets issued.  Set it when a bio
         * is being charged to a tg.
         */
        if (!bio_flagged(bio, BIO_THROTTLED))
                bio_set_flag(bio, BIO_THROTTLED);
}
```

Function tg_may_dispatch determines if the bio is within the limits for that cgroup or not by checking both the bytes per sec limit as well as the I/O per sec limit for the cgroup. If the limit is not exceeded, the bio is first charged to the cgroup. Function thtrotl_charge_bio charges the bio to the throttle group. The bio is subsequently passed to the cgroup parent using the following code:

```
 /* bio passed through this layer without being throttled.
    Climb up the ladder.  If we are already at the top, it
    can be executed directly. */
qn = &tg->qnode_on_parent[rw];
sq = sq->parent_sq;
tg = sq_to_tg(sq);
```

If the limit is exceeded, the bio follows a different path. Function throtl_add_bio_tg adds the bio to the throttle service queue to meet the resource limitations. This throttle service queue is drained later. For delayed operations, a dispatcher function will then cause pending operations to be executed using pre-calculated timers in order to throttle

requested operations. The required time limiting is calculated by function `throtl_trim_slice`, yielding the time slice the operation may be executed in.

```
static void throtl_add_bio_tg(struct bio *bio, struct throtl_qnode *qn,
                              struct throtl_grp *tg)
{
        struct throtl_service_queue *sq = &tg->service_queue;
        bool rw = bio_data_dir(bio);

        if (!qn)
                qn = &tg->qnode_on_self[rw];

        /*
         * If @tg doesn't currently have any bios queued in the same
         * direction, queueing @bio can change when @tg should be
         * dispatched.  Mark that @tg was empty.  This is automatically
         * cleared on the next tg_update_disptime().
         */
        if (!sq->nr_queued[rw])
                tg->flags |= THROTL_TG_WAS_EMPTY;

        throtl_qnode_add_bio(bio, qn, &sq->queued[rw]);

        sq->nr_queued[rw]++;
        throtl_enqueue_tg(tg);
}
```

Overlay File System: Layered file system enables two of the biggest advantages of containers, portability and lightweight. A container is composed of multiple layers, one or more image layers and a container layer. Image layers are read-only, container layer is writable. Containers use a concept called overlay file system to present these multiple layers (filesystems) as a single root file system to the user. All of these layers are

stored in the native file system (filesystems can be different, see Figure 5.12). Overlay file system works on top of these filesystems. When a container is started, a thin writable layer is initiated on top of read-only image layers. Once the container modifies a file from the image layer, that file is copied into the writable container layer and will be modified there. The overlay file system searchers from upper to lower layers until the target file is found, in other words, the copied file overlays the file from the image layer.

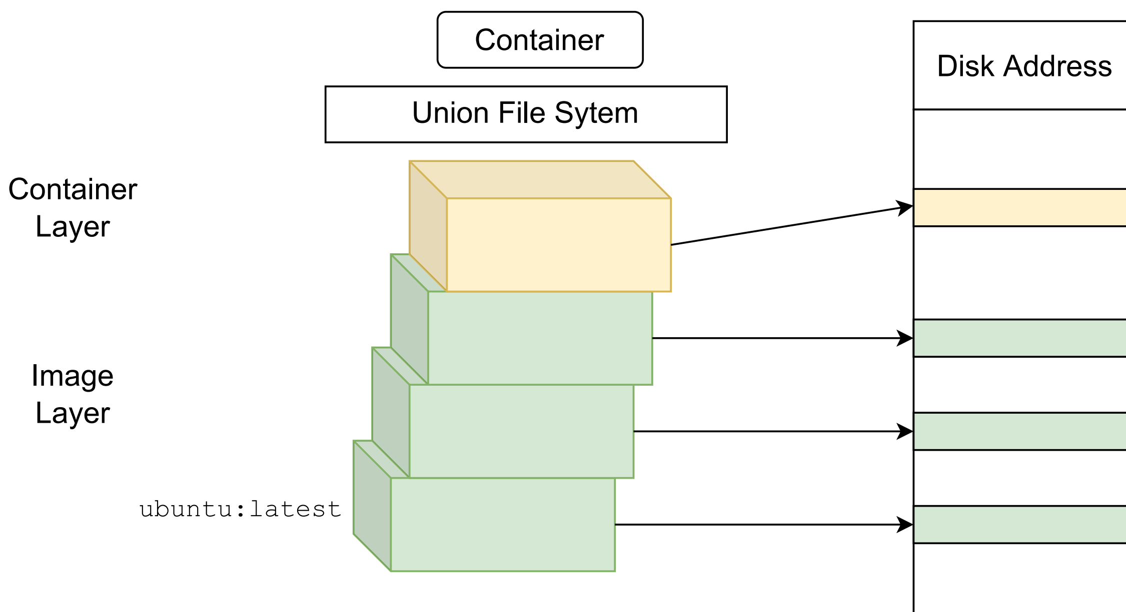


Figure 5.12: Layered File System of Container.

Docker is one of the most popular container runtimes and used by many companies and organizations from both industry and academia. Docker consists of three components: (i) a registry service; (ii) a daemon running on a Docker host; and (iii) a client to interact with the daemon. The Docker registry is used to store and distribute Docker images across daemons. An image consists of multiple layers, each of which is a set of files that will be

included in a running container. Layers can be shared across different images. The Docker daemon is the persistent process that runs on a host machine and manages containers and images while the Docker client is used to send user requests to the Docker daemon such as starting or stopping a container from a specific image. When the daemon receives a request for creating a new container, it will use Linux cgroups to isolate compute resources and generate a private namespace based on Linux namespaces. It will then union the image's read only layers and create a writable layer to provision the file system for the container.

5.1.6 Control Groups: I/O and Memory Controller Explained

Resource allocation is managed by kernel software at runtime. As a result, resource contention and performance interference is a more significant problem in a containerized environment. The degree of resource contention grows with the number of containerized instances, resulting in SLA violation for some instances. Therefore, it is necessary to implement an effective resource allocation scheme to mitigate the performance interference problem. Cgroup is a Linux kernel mechanism that provides resource control, limit and prioritization for each process. These resource include CPU, memory and I/O. The resource controller tracks the resources, accounts them to processes and dynamically allocates/throttles them. With cgroup, users can ensure that the target workload is not impacted by system background or co-located workloads that use excess system resources. Users can also reserve system resource to high priority workloads by limiting available resource to low priority workloads. cgroup provides a specific I/O subsystem named

blkio (io in subsequent version) and memory subsystem named memory, we discuss them in more details in this section.

The first version of cgroup appears in Linux kernel 2.6.24. cgroupv1 has a hierarchy per-resource, each resource hierarchy contains cgroups for this resource. Limits and accounting are performed per-cgroup. However, not all controllers follow this hierarchical concept, for exmaple, when throttling in the blkio is configured, all cgroups are treated at the same level. cgroupv2 (since Linux kernel 4.5) changes this to an unified hierarchy for all controllers for simplicity and clarity, each cgroup can support multiple resource domains. cgroupv1 I/O controller implementation is partially discussed in Section 5.1.5, we discuss memory controller and cgroupv2 io controller implementation in detail.

The main purpose of memory cgroup (memcg) is to account usage of memory. Each cgroup has a memory controller specific data structure (mem_cgroup, defined in <linux/memcontrol.h>) associated with it.

```
struct mem_cgroup {
        struct cgroup_subsys_state css;

        /* Private memcg ID. Used to ID objects that outlive the cgroup */
        struct mem_cgroup_id id;

        /* Accounted resources */
        struct page_counter memory;                /* Both v1 & v2 */

        union {
                struct page_counter swap;        /* v2 only */
                struct page_counter memsw;       /* v1 only */
        };

        /* Legacy consumer-oriented counters */
```

```
        struct page_counter kmem;                   /* v1 only */
        struct page_counter tcpmem;                   /* v1 only */

        /* Range enforcement for interrupt charges */
        struct work_struct high_work;

        unsigned long soft_limit;

        ...
        /* memory.events and memory.events.local */
        struct cgroup_file events_file;
        struct cgroup_file events_local_file;

        /* handle for "memory.swap.events" */
        struct cgroup_file swap_events_file;

        /* thresholds for memory usage. RCU-protected */
        struct mem_cgroup_thresholds thresholds;

        /* memory.stat */
        struct memcg_vmstats        vmstats;

        /* memory.events */
        atomic_long_t                memory_events[MEMCG_NR_MEMORY_EVENTS];
        atomic_long_t                memory_events_local[MEMCG_NR_MEMORY_EVENTS];

        unsigned long                socket_pressure;

        /* Legacy tcp memory accounting */
        bool                        tcpmem_active;
        int                        tcpmem_pressure;
        struct mem_cgroup_per_node *nodeinfo[];
};
```

Memory account occurs per `mem_cgroup`, each `mm_struct` knows about which cgroup

it belongs to by `owner` field in `struct mm_struct`. Each page is pointed to a `page_cgroup`, indicating its owner.

```
struct page_cgroup {
    unsigned long flags;
    struct mem_cgroup *mem_cgroup;
    struct page *page;
    struct list_head head;
};
```

Currently, the following types of memory usages by a cgroup are tracked and accounted: anonymous memory, file cache and kernel data structures such as dentries and inodes. Accounting a memory page to its owner cgroup is proceeded as follows: (1) locate the owner cgroup of the given memory page. (2) Function `try_charge()` is invoked to check if the cgroup that is being charged is over its limit. If it is, then invoke a reclaim on the target cgroup to free pages, and re-try the charge later. (3) Function `commit_charge()` commits the charge to the target cgroup and updates the `mem_cg` data structure. Each cgroup maintains a LRU per group. Once the memory usage of the cgroup exceeds its limit, a memory reclaim is triggered. If the reclaim is unsuccessful, an OOM (Out-Of-Memory) routine is invoked to select and kill the process that consumes most memory in the cgroup. As of now, cgroup `memory` provides the following interface to track and manage memory:

- **memory.current**: the total amount of memory currently being used by the cgroup and its descendants.
- **memory.min**: if the memory usage of a cgroup is within its effective min boundary,

the cgroup's memory won't be reclaimed under any conditions.

- **memory.low**: if the memory usage of a cgroup is within its effective low boundary, the cgroup's memory won't be reclaimed unless there is no reclaimable memory available in unprotected cgroups.
- **memory.high**: memory usage throttle limit. This is the main mechanism to control memory usage of a cgroup. If a cgroup's usage goes over the high boundary, the processes of the cgroup are throttled and put under heavy reclaim pressure.
- **memory.max**: if a cgroup's memory usage reaches this limit and can't be reduced, the OOM killer is invoked in the cgroup.
- **memory.oom.group**: determines whether the cgroup should be treated as an indivisible workload by the OOM killer. If set, all tasks belonging to the cgroup or to its descendants (if the memory cgroup is not a leaf cgroup) are killed together or not at all.
- **memory.events**: tracks the number of times that each memory event is triggered.
- **memory.stat**: breaks down the cgroup's memory footprint into different types of memory, type-specific details, and other information on the state and past events of the memory management system.

I/O control relies on the block I/O scheduler implemented in the block layer. The `bio` structures encapsulating request type (read or write), I/O block size, target address (offset, device), issuing cgroup are dispatched to the block layer. If an upper limit

is specified for the cgroup, the block layer checks the dispatched IOPS in the current slice and decides whether to dispatch immediately. To handle proportional policy, the CFQ/BFQ scheduler arbitrates dispatching of I/O requests by the specified I/O weights. However, these arbitration only applies to direct I/Os and they do not work along with the multi-queue block layer. Existing protocols can't manage read requests serviced from page cache or buffered writes. Why don't kernel developers build throttling protocol in upper layers? First, the VFS layer and the page cache layer are not exclusive for block I/Os. Second, I/O control protocols may need to be implemented for each file system if the file system layer is responsible for I/O arbitration. Therefore, it becomes a natural choice to implement I/O control in the block layer. Up to today, the blkio control provides several basic strategies, I/O throttling with maximum and minimum limit, CFQ/BFQ proportional weight and writeback throttling based on latency.

Cgroupv1 `blkio` enables users to set upper and lower limits in the form of read/write IOPs or bytes per second, or set proportional weights to cgroups. For example, if the weights of two containers are 1 and 2 respectively, the I/O throughput ratio is expected to be $1 : 2$. There are several limitations: (1) numerical limits are not scalable with a large variety of SSDs and workloads. If configured incorrectly, the isolation may not work as expected or even degrade the performance of the entire I/O stack. (2) I/O proportion is only implemented with the BFQ I/O scheduler. However, BFQ I/O scheduler is not used under many use scenarios (e.g. for NVMe SSDs). (3) `blkio` only controls direct I/Os since it is built upon the block layer. This leaves other types of I/O, such as buffered I/O (the majority of I/Os) out of control. (4) It can't track which cgroup generates IOPS and therefore it can't throttle I/O requests correctly. To address problem 3 and 4, cgroup2

introduces more efficient mechanism to control buffered I/Os, and enables tracking of page cache writeback for each cgroup. Cgroup2 [80] relies on a combination of memory subsystems (memcg) and I/O subsystems (blkcg) to control buffered write rate and track the owner cgroup of the writeback. This approach partially solved problem 3: If a cgroup flushed its dirty pages to disk, it conforms to its own I/O and memory limits instead of blocking page cache indefinitely. It also works in the opposite direction, the frequency of dirty page writeback can be impacted by memory limits. Cgroup2 also introduces a new I/O controller `IOLatency` [250], which allows setting I/O latency targets for the controlled cgroup. `IOLatency` throttles other cgroups if the target maximum latency is violated for the specified cgroup. Though cgroup2 provides more effective and refined I/O control, most of the aforementioned problems (1, 2 and 3) remain unsolved. We demonstrate these limitations in the following Section 5.2: current infrastructure (1) can't guarantee proportional IOPs. (2) can't guarantee page cache fairness, resulting in severe performance degradation of co-located workloads. (3) sensitive to other parts of the memory subsystem.

As of now, cgroup `memory` provides the following interface to track and manage io:

- **io.stat**: io usage statistics by cgroup, including Bytes read, Bytes written, Number of read IOs, Number of write IOs, Bytes discarded, Number of discard IOs.
- **io.cost**: configures the Quality of Service of the IO cost model based controller (CONFIG_BLK_CGROUP_IOCOST) which currently implements "io.weight" proportional control.

- **io.weight**: The first line is the default weight applied to devices without specific override. The weights are in the range [1, 10000] and specifies the relative amount IO time the cgroup can use in relation to its siblings.

- **io.max**: BPS and IOPS based IO limit. BPS and IOPS are measured in each IO direction and IOs are delayed if limit is reached. Temporary bursts are allowed.

- **io.latency**: Quality of Service mechanism to guarantee a cgroup's level of IO completion latency. If the average completion latency is longer than the target set here, other processes are throttled to provide more IO, effectively prioritizing the job with the lowest io.latency setting.

- **io.pressure**: gives the percentage of wall time in which some or all tasks are waiting for a block device, or IO.

- **blkio.prio**: controls the behavior of the I/O priority cgroup policy. Following types of I/O priority policies are provided from low to high, no-change, none-to-rt, restrict-to-be, idle.

The io.min controller allows a minimum limit for a cgroup. If one cgroup doesn't reach its low limit, no other cgroups can use more bandwidth/iops than their low limit. The io.max controller enforces a maximum limit for a cgroup. We have discussed their implementations in Section 5.1.5.

There are multiple ways to set up **io.weight** as of now.

```
# create test cgroups
mkdir /sys/fs/cgroup/test1/
```

```
mkdir /sys/fs/cgroup/test2/

## 1. configure io.weight through bfq scheduler

# enable ioscheduler and bfq weight, suppose nvme0n1 is the target device
echo bfq > /sys/block/nvme0n1/queue/scheduler

# configure weight
echo 100 > /sys/fs/cgroup/test1/io.bfq.weight
echo 200 > /sys/fs/cgroup/test2/io.bfq.weight

# clear blk-iocost if set
echo 0 > /sys/fs/cgroup/test1/io.weight
echo 0 > /sys/fs/cgroup/test2/io.weight

# clear nvme-wrr weight if set
echo "`cat /sys/block/nvme1n1/dev` none" > /sys/fs/cgroup/test1/io.wrr
echo "`cat /sys/block/nvme1n1/dev` none" > /sys/fs/cgroup/test2/io.wrr

## 2. configure io.weight through io.cost

# disable io scheduler and bfq weight
echo none > /sys/block/nvme0n1/queue/scheduler

# configure qos generated from the io.cost model
echo "`cat /sys/block/nvme0n1/dev` ctrl=user model=linear rbps=1000000000 rseqiops=300000

# configure weight
echo 100 > /sys/fs/cgroup/test1/io.weight
echo 200 > /sys/fs/cgroup/test2/io.weight

# clear nvme-wrr weight if set
echo "`cat /sys/block/nvme1n1/dev` none" > /sys/fs/cgroup/test1/io.wrr
echo "`cat /sys/block/nvme1n1/dev` none" > /sys/fs/cgroup/test2/io.wrr
```

The **io.cost** interface in cgroup v2 provides the blk-iocost feature and limits the I/O

quality of service (QoS) rate based on the weight. The function dynamically predicts how expensive a given stream of I/Os would be on a specific device. The cost model is based on adjusting the overall IO rate according to how busy the device is. The detailed logic of io.cost is defined in `<block-iocost.c>`.

The **io.latency** interface allows users to set a latency target for the target cgroup. The controller tracks the actual latency across the whole block layer at `bio` level. The latency is measured as the average over the 100ms window. By default no throttling will be triggered, the queue_depth is set to `UINT_MAX` so that each cgroup queue can have unlimited outstanding `bio`. If the threshold is exceeded for a given time period (250ms by default), the controller protects the target latency by throttling other peers that have a higher latency target. Two ways to throttle I/O are provided: (1) queue depth throttling. Once the threshold is exceeded, the controller starts to shrink the queue depth allowed for outstanding bios in flight. This is achieved by function `scale_change()` defined in `<-iolatency.c>`. (2) induced delay throttling. This is for the case that a group is generating I/O that has to be issued by the root cgroup to avoid priority inversion. The induced delay will throttle back the activity that is generating the root cg issued I/O's by function `blkcg_add_delay()`, wethere that's some metadata intensive operation or the group is using so much memory that it is pushing us into swap. The **io.latency** controller accounts for I/O time by counting from the time that each I/O is submitted to the time it is completed.

```
struct iolatency_grp {
        struct blkg_policy_data pd;
        struct latency_stat __percpu *stats;
```

```
	struct latency_stat cur_stat;
	struct blk_iolatency *blkiolat;
	struct rq_depth rq_depth;
	struct rq_wait rq_wait;
	atomic64_t window_start;
	atomic_t scale_cookie;
	u64 min_lat_nsec;
	u64 cur_win_nsec;

	/* total running average of our io latency. */
	u64 lat_avg;

	/* Our current number of IO's for the last summation. */
	u64 nr_samples;

	bool ssd;
	struct child_latency_info child_lat;
};
static void blkcg_iolatency_throttle(struct rq_qos *rqos, struct bio *bio)
{
	struct blk_iolatency *blkiolat = BLKIOLATENCY(rqos);
	struct blkcg_gq *blkg = bio->bi_blkg;
	bool issue_as_root = bio_issue_as_root_blkg(bio);

	if (!blk_iolatency_enabled(blkiolat))
		return;

	while (blkg && blkg->parent) {
		struct iolatency_grp *iolat = blkg_to_lat(blkg);
		if (!iolat) {
			blkg = blkg->parent;
			continue;
		}

		check_scale_change(iolat);
		__blkcg_iolatency_throttle(rqos, iolat, issue_as_root,
```

```
                                        (bio->bi_opf & REQ_SWAP) == REQ_SWAP);
                blkg = blkg->parent;
        }
        if (!timer_pending(&blkiolat->timer))
                mod_timer(&blkiolat->timer, jiffies + HZ);
}

static void __blkcg_iolatency_throttle(struct rq_qos *rqos,
                                        struct iolatency_grp *iolat,
                                        bool issue_as_root,
                                        bool use_memdelay)
{
        struct rq_wait *rqw = &iolat->rq_wait;
        unsigned use_delay = atomic_read(&lat_to_blkg(iolat)->use_delay);

        if (use_delay)
                blkcg_schedule_throttle(rqos->q, use_memdelay);

        /*
         * To avoid priority inversions we want to just take a slot if we are
         * issuing as root.  If we're being killed off there's no point in
         * delaying things, we may have been killed by OOM so throttling may
         * make recovery take even longer, so just let the IO's through so the
         * task can go away.
         */
        if (issue_as_root || fatal_signal_pending(current)) {
                atomic_inc(&rqw->inflight);
                return;
        }

        rq_qos_wait(rqw, iolat, iolat_acquire_inflight, iolat_cleanup_cb);
}
```

The **blkio.prio** interface allows to set the I/O priority class for a request. Four priority classes are provided for this attribute: (1) NO_CHANGE (2) NONE_TO_RT

(3) RESTRICT_TO_BE (4) ALL_TO_IDLE. Each attribute is associated with numerical values from 0 to 3, higher I/O priority numbers correspond to a lower priority. Each cgroup maintains a `ioprio_blkcg` (defined in `<-ioprio.c>` structure to manage a per (cgroup, request queue) policy structure. The I/O priority class of a *bio* request is updated to the maximum of the I/O priority class policy number and the numerical I/O priority class.

```
/**
 * enum prio_policy - I/O priority class policy.
 * @POLICY_NO_CHANGE: (default) do not modify the I/O priority class.
 * @POLICY_NONE_TO_RT: modify IOPRIO_CLASS_NONE into IOPRIO_CLASS_RT.
 * @POLICY_RESTRICT_TO_BE: modify IOPRIO_CLASS_NONE and IOPRIO_CLASS_RT into
 *                 IOPRIO_CLASS_BE.
 * @POLICY_ALL_TO_IDLE: change the I/O priority class into IOPRIO_CLASS_IDLE.
 *
 * See also <linux/ioprio.h>.
 */
enum prio_policy {
        POLICY_NO_CHANGE        = 0,
        POLICY_NONE_TO_RT        = 1,
        POLICY_RESTRICT_TO_BE        = 2,
        POLICY_ALL_TO_IDLE        = 3,
};
/*
 * struct ioprio_blkcg - Per cgroup data.
 * @cpd: blkcg_policy_data structure.
 * @prio_policy: One of the IOPRIO_CLASS_* values. See also <linux/ioprio.h>.
 */
struct ioprio_blkcg {
        struct blkcg_policy_data cpd;
        enum prio_policy         prio_policy;
        bool                             prio_set;
};
```

Cgroup offers control for page cache writeback. Page cache is dirtied through buffered writes and written asynchronously to storage device by the writeback mechanism. In cgroup v1, all writeback I/O is charged to root `blkcg` because there are inherent differences in how cgroup memory page and block I/O ownership is tracked, memory is tracked per page while writeback per inode. Cgroup v2 introduces new methods to account writeback with cooperation between `blkcg` and `memcg`. Each `memcg` maintains pointers that point to the corresponding `blkcg`. Therefore, the writeback I/O is charged to the `blkcg` that the `memcg` of the page corresponds to. The throttling mechanism works by shrinking the queue depth of the device once the average measured read latency exceeds the configured threshold. Interfaces to configure cgroup writeback are used: (1) vm.dirty_background_ratio, vm.dirty_ratio apply the same to cgroup writeback with the amount of available memory capped by limits imposed by the memory controller and system-wide clean memory. (2) vm.dirty_background_bytes, vm.dirty_bytes, for cgroup writeback, this is calculated into ratio against total available memory. The memory controller defines the memory domain that dirty memory ratio is calculated and maintained for and the io controller defines the io domain which writes out dirty pages for the memory domain. Both system-wide and per-cgroup dirty memory states are examined and the more restrictive of the two is enforced.

5.2 Performance Pitfalls

In this section, we discuss the limitations of existing I/O control mechanisms.

Cgroup I/O control alone is insufficient for proportional I/O as data structure sharing in page cache results in I/O and memory interference between workloads.

Current I/O control stack relies on the block-level I/O scheduling to achieve throughput throttling and distribution. However, buffered I/O are occasionally serviced from the page cache, only read requests experiencing a cache miss go to the block layer. As the page cache layer does not take I/O weight or priority into account, Linux I/O control mechanism does not cater to buffered I/Os from design. We perform a set of experiments to demonstrate how I/O weight control impacts the actual throughput distribution. Experimental setup is listed in Table 5.2. The first process (foreground) and the second process's (background) weight is set as 300, and 100 respectively. That being said, the ideal throughput ratio is 3:1. In the first experiment, we choose workload A and workload B as the foreground and background process. Workload A simulates a latency-critical, read-intensive application that keeps issuing sequential 4KB requests. Workload A's performance heavily relies on the page cache. Workload B issues sequential direct I/Os to the storage. We ensure that the aggregated peak bandwidth is well below the maximum bandwidth of the device. The throughput ratio between A and B is 3.24. Meanwhile, the average request latency of A and B remains nearly unchanged. The result shows that proportional performance can be achieved when direct I/Os go through the weight-aware block-level scheduler. The second experiment changes background process to workload C. The only difference between B and C is that the I/O type is now buffered. The

throughput ratio floats around 3:1 during the first two minutes, which gradually drops to 1.48 afterwards. The throughput of workload A shrinks by 25.6%. This is because less I/Os of workload A are serviced from page cache – we observe a lower cache hit rate as workload C contends for the page cache entries. Meanwhile, the average latency of workload A increases by 24.9%, the 95th percentile latency increased by 39.2%. These two experiments reveal that current I/O priority framework in Linux does not work well for buffered I/O. Rather, when multiple cgroups contend for page cache resource, it would likely lead to I/O performance interference between processes, since the page cache management does not consider I/O weights or priority. In extreme scenarios where several I/O flows put heavy pressure on page cache or memory subsystem, the OS kernel is forced to free up the memory pages owned by other cgroups in a very random manner (including page cache and anonymous pages). The interaction with memory management can easily lead to I/O control failure and even unexpected performance degradation.

With enhanced memory management, cgroup2 may indirectly limit page cache usage and writeback rate. However, inappropriate configurations may result in performance degradation in peer cgroups.

Cgroup2 introduces a more effective I/O control scheme by combining `blkcg` and `memcg`. Basically, I/O control interacts with the memory management functions such as page reclaim to indirectly manipulate page cache footprint. To limit page cache usage, developers need to carefully configure the maximum memory usage (`memory.max`), dirty page ratio (`vm.dirty_ratio`) per application and per-device. `vm.dirty_ratio` is calculated in the memory domain and I/O domain to regulate the proportion of dirty memory by

Table 5.2: Experimental setup.

OS	Ubuntu 20.04
Kernel	Linux 5.8
CPU	AMD Ryzen 9 5900X 12 cores@4.5GHz
Memory	32GB DDR4 3600 MHz
Storage	Intel P4510 1TB U2
Workloads	A: FIO 4KB sequential RW, read ratio = 0.95, QD = 16 B: FIO 4KB sequential read, direct I/O, QD = 32 C: FIO 4KB sequential read, buffered I/O, QD = 32 D: YCSB Workload C on RocksDB
Experiments	A + B, weights = 300 : 100. A + C, weights = 300 : 100. A + D

balancing dirtying and write I/Os. Users must be aware of such interactions and other services utilization, or inappropriate configurations may deteriorate the I/O performance of peer cgroups. We demonstrate this performance pitfall in environments mentioned in Table 5.2, with workload A and D. We set the `memory.max` limit for D, which will not be allocated more memory than the `max` limit. When D's hard memory limit is set 50% below its normal demand, we observe degraded I/O performance in workload A: the IOPS drops to approximately 1/4 of its original rate, the average latency increases by 2.7x. By analyzing the number of page reads handled by page cache, and the number of D hitting the memory limits by `memory.event`, we find out that the performance degradation

in A is due to page cache thrashing. When D demands more memory than the limit, it is forced to reclaim page cache to satisfy the constraints. Frequent page insertion and eviction triggered by workload D prevents A from acquiring the mutual lock, since OS kernel does not respect the ownership or priority of page cache entries. This results in unfair page cache allocation and access. In another experiment where we deliberately let D produce too many dirty pages, the pages owned by A can even be 'stolen' (by observing the `active_file`), which leads to similar performance degradation.

Reckless use of I/O latency control may cause a sharp decline in I/O performance of peer cgroups.

Cgroup2 introduces a new feature `io.latency` that can be specified to approach a latency target. If the protected workload experiences average completion latency longer than its target latency, the cgroup I/O controller can throttle its peers via block-level I/O scheduling. Note the limits are applied only at the peer level in the cgroup hierarchy. According to developers' document, throttling works in two ways: (1) It throttles the number of outstanding I/O that a cgroup is allowed to have. (2) It adds additional delay to swapping and metadata I/Os. We demonstrate that reckless use of I/O latency control may hurt I/O performance of other cgroups. We use workload A/D and configure A's target latency slightly below the average latency of running alone. We discover that A dominates the I/O flow no matter the weight difference in D. The I/O activity of D suffers a sharp decline. In the best case, workload A and D's throughput distribution is around 6 to 1. Workload D's throughput recovers only until A's target latency is raised to 1.6x of its average latency. In an I/O-heavy scenario, if the target latency of a cgroup is configured

without considering the application and device performance, other peer cgroups might suffer a sharp decline in performance. Developers must be aware of the system-level resource utilization, and per-application, per-device performance projections to apply resource limit correctly.

5.3 Design and Implementation

To partially improve the aforementioned limitations, a feasible solution is enhancing the page cache module in the kernel: (1) track which cgroup is responsible for a page cache entry, especially when the page is associated with multiple cgroups. (2) monitor and limit the page cache usage per cgroup. (3) implement a page cache allocation/reclamation scheme that respects the I/O weight. We explain the implementation details in this section.

Figure 5.13 illustrates the detailed process of a generic read request. When a buffered read request arrives at the page cache layer, it first checks if the requested page is present in the page cache. If yes, the read request is directly serviced by the cached page. If not, a new page is to be allocated by the `page_cache_alloc()` function. The cgroup controller should check if the current page cache usage exceeds the cgroup limit before the new allocated page is added to the page cache and the LRU list. If the limit is exceeded, the cgroup controller tries to free memory resources through page reclamation service.

To implement page cache limit per cgroup, We add `mem_cgroup` in the page

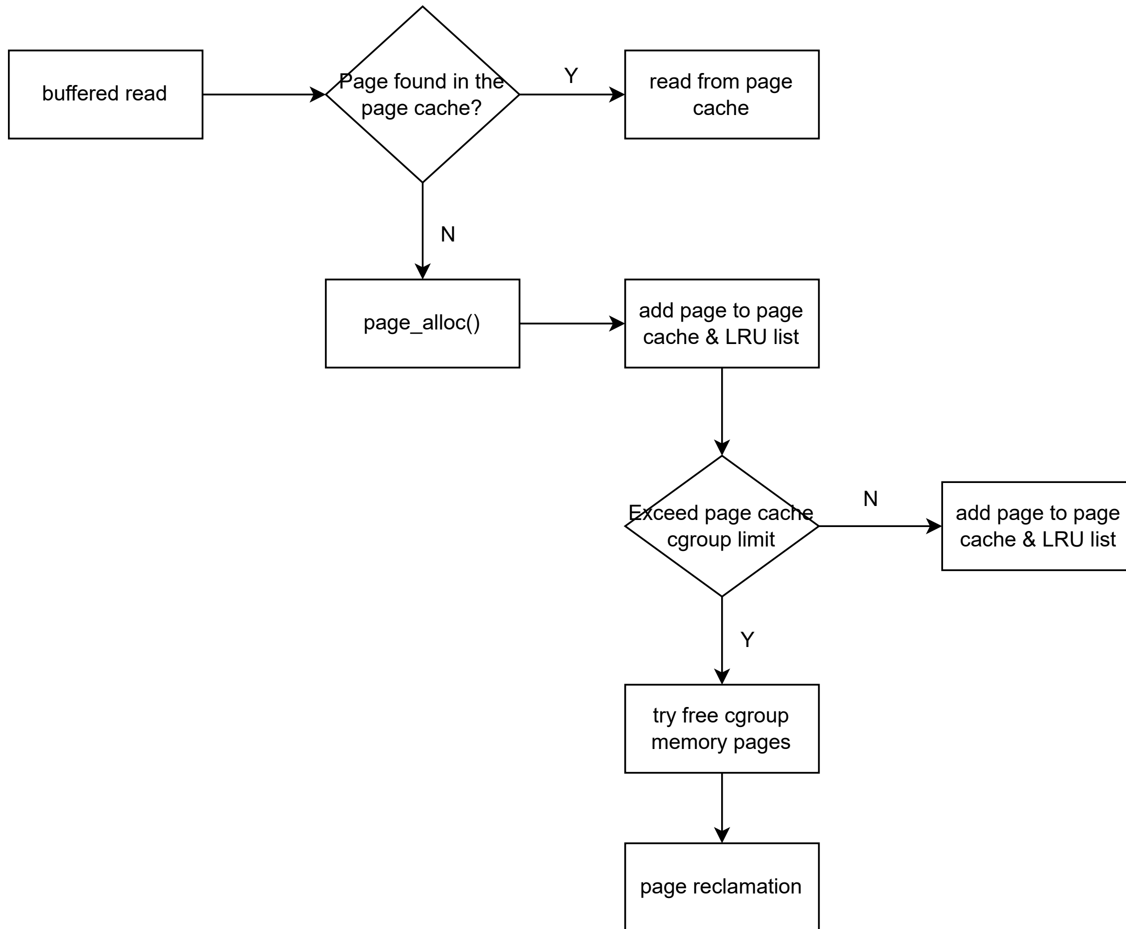


Figure 5.13: Process of a Buffered Read Request.

structure (mm/memcontrol.h) to bind every page with its owner cgroup. During every page allocation and reclamation process related to the cgroup, we update the nrp_pages variable in the mem_cgroup to keep track of the number of pages each cgroup owns. In memcontroller code (mm/memcontrol.c), we insert an ethical check if the page requests are for page cache to update the page cache counter and enforce the upper limit. We add a new function pagecache_cgroup_left to calculate the remaining share of page cache for the cgroup. To reflect the weight of each cgroup, we implement a token bucket algorithm that assign tokens to controlled cgroups at a set rate in memory page reclaimation (mm/vmscan.c). Applying a new page costs a token, the application for new entries is blocked if the number of tokens is negative. Note only buffered writes and block-level reads are throttled when the cgroup doesn't own any tokens. Before the requested page entries are allocated, we also check if the applying cgroup exceeds the upper limit. If yes, we enforce its children cgroups (the ancestor cgroup aggregates children counters) to scan for candidate victim pages. During this process, the priority of a cgroup decides the search range for victim pages. In this way, the reclamation process tries to keep more open entries owned by higher priority cgroups in cache. The victim page can either be in the active_list or the inactive_list. If the page to be reclaimed is dirty, we must wait the page to finish its writeback. After the page reclamation is finished, the requested pages are allocated. Meanwhile, the page counters of the current cgroup and its associated cgroups are updated. Besides, the page ownership is updated when another cgroup attempts a dirty-write or eviction. As a result, the rate of writeback is credited to the page's real owner. For user manipulation, we add two APIs memory.pagecache_usage, memory.pagecache_limit in the cgroup memcg

interface, to monitor and enforce an upper limit on page cache usage, respectively.

```
filmap.c: __add_to_page_cache_locked
/* add page mapping attribute to signal the page is a cached page or not. */

memcontrol.c: mem_cgroup_try_charge, try_charge
/* check if the newly added page by page->mapping attribute is a cached page before try_c

memcontrol.c: mem_cgroup_migrate
/* use page_counter_charge() to charge pagecache if the new page is a cached page */

vmscan.c: try_to_free_mem_cgroup_pages
/* if nr_pages > page_cache_limit, claim page cache as much as possible. */
try_to_free_mem_cgroup_pages
        do_try_to_free_pages
        shrink_zones
        mem_cgourp_soft_limit_reclaim
        mem_cgroup_soft_reclaim
        mem_cgroup_shrink_node
        shrink_node_memcg
            shrink_list
            shrink_active_list
            shrink_inactive_list
```

5.4 Evaluation

We implement our design on Linux kernel 5.8 and all evaluations are conducted on a machine listed in Table 5.2. Cgroup1 is disabled to enable the modified cgroup2 `memcontrol` and `iocontrol`. We show that our design provides a low-overhead and more effective proportional cgroup I/O control.

First, we quantify the overhead of our design. We measure the latency of the

Table 5.3: Read/Write latency (us) of modified kernel compared against base.

Latency	**Baseline**	**Modified**
Read	18.4	18.6
Write	20.9	22.8

modified kernel against the conventional scheme as shown in Table 5.3. The controllers are not configured to perform any control or throttling in order to measure the baseline overheads of our scheme. Both read and write latency is measured with 4KB buffered sequential workload at queue depth 1. The result illustrates that despite having a more sophisticated logic that tracks page cache usage, our design does not introduce noticeable software overhead.

To evaluate the efficacy of I/O proportionality control, we perform several experiments where three synthetic workloads run concurrently. Results are compare against the conventional BFQ-based implementation. Figure 5.14 and 5.15 show IOPS distribution among the workloads with different I/O weights. Three weight combinations 1/1/1, 1/1.5/2.5, 1/2/3 are evaluated. It is visible that our design delivers much better I/O proportionality than the BFQ-based weight protocol. This experiment also demonstrates that our design generally allows high-priority workload to consume more I/O than its weight allows. This might be due to the loose control on high-priority workload page reclamation. With page cache limit it becomes possible to provide proportional I/O control to buffered I/Os.

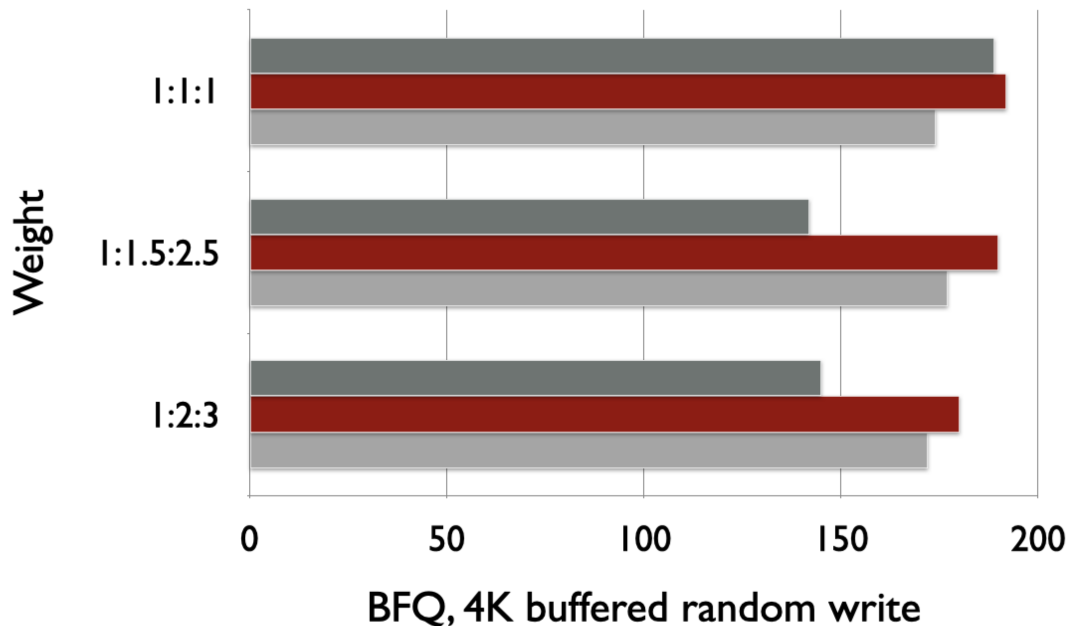


Figure 5.14: IOPS among three 4KB buffered write workloads, using BFQ.

5.5 Discussions

In previous sections, we discuss sophisticated I/O schedulers and throttling mechanisms at the block layer that provide differentiated I/O performance. Given multiple I/O streams, the I/O schedulers arbitrates when and which requests should be dispatched, in order to satisfy the defined constraints. However, block-level I/O scheduling has several limitations: (1) Software Overhead. According to our profiling of time spent in the block layer, scheduling and dispatching can consume up to 30% of the total response time. This overhead is no longer negligible as emerging low-latency SSD access time becomes comparable to the software overhead. (2) CPU overhead and scalability. The high cost of the block I/O scheduling can exacerbate the problem of CPU bottleneck in modern data-intensive applications. Besides, it is no longer a trivial problem for software I/O

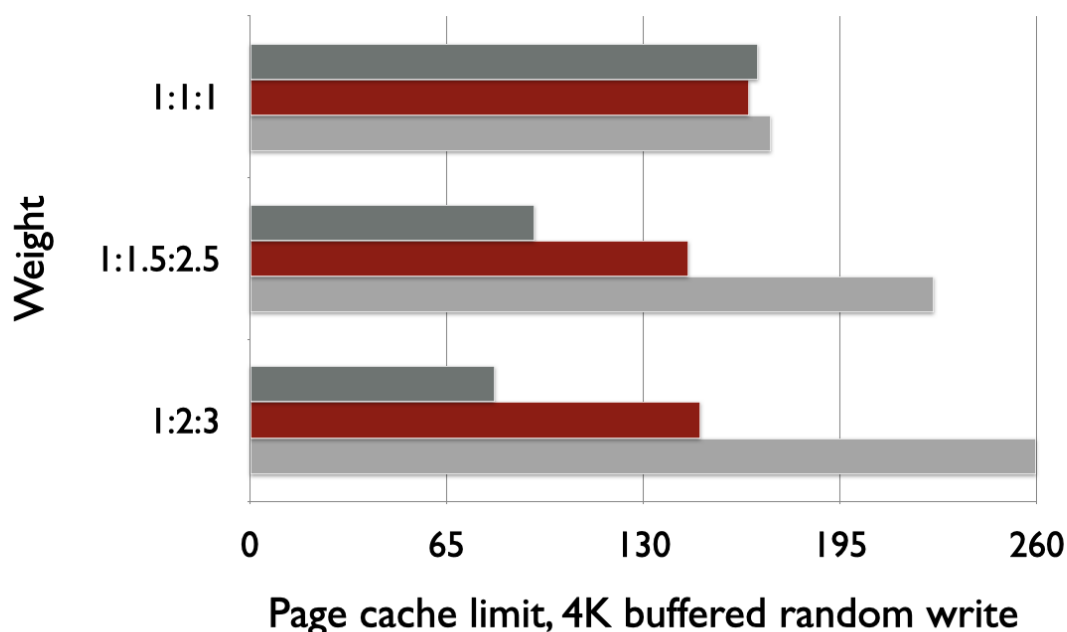


Figure 5.15: IOPS among three 4KB buffered write workloads, using page cache limit.

schedulers to scale with the increasing I/O issuing threads and the available hardware queues in the NVMe SSD. (3) does not have tail latency alleviation mechanism especially in multi-tenant deployments.

Ironically, NVMe devices provide a device-side scheduling protocol, called weighted round-robin (WRR). It provides three priority classes of I/O command queues, each with a configurable weight to differentiate I/O priority and weight. The WRR queue arbitration is processed by the SSD controller, which can eliminate scheduling cost on the host side. It is indeed attracting to exploit device-side arbitration to schedule user I/Os. However, there are missing pieces for this scheme: (1) NVMe WRR only has three priority classes, whereas the number of application I/O flows can be much higher. (2) It is challenging to bridge the semantic gap between I/O QoS requirement and the storage device, such as

the context to determine the ownership of a particular I/O request and the I/O patterns. If these challenges can be tamed, it is desirable to offload I/O scheduling to the NVMe SSD as the device has more potential to decide data placement in order to better (1) satisfy I/O control and QoS (2) achieve better WAF (Write Amplification Factor). With device-side scheduling, it is possible to achieve faster access to the raw device since the block layer no longer performs I/O merging or scheduling thus reduces I/O submission delay significantly. We believe that it is a better option to implement a hardware-assisted I/O control mechanism (e.g. using NVMe arbitration) with software services.

The evolution of emerging hardware and more data-intensive applications has presented new challenges to traditional I/O stack design. And existing kernels are complex and tightly integrated with many other subsystems. To implement more effective I/O control mechanisms, we must combine the device capabilities and the potential impact on other subsystems with kernel design.

5.6 Summary

In this chapter, we analyze the limitations of state-of-the-art I/O control mechanism. To enable more effective I/O proportionality and priority, we propose kernel modifications that add APIs to limit page cache usage per cgroup and a new page allocation/reclamation scheme that respects I/O weight and priority. Our evaluations show that our approach modestly improves I/O fairness, compared with the state-of-the-art protocol. Lastly, we discuss other efficient approaches to enable effective I/O control.

Chapter 6: Related Work

6.1 Emerging Storage Interface

Ironically, the block interface remains familiar to anyone who developed the I/O portion in a program 20 years ago. The conventional block interface abstracts too much of the SSD's internals, which makes it difficult for application users to maximize performance and device endurance by talking to the device. Significant research effort has pushed into optimizing the conventional SSDs' block interface. Many previous work attempts to resolve the performance degradation and unpredictability caused by garbage collection and other FTL tasks [100, 201, 264, 265]. This is because SSD's internal activities are inherently sub-optimal and unpredictable by design, as the block interface prevents FTL from accessing the context of QoS requirement (relative priority among concurrent requests) and future accesses, and prevents users from regulating the background behaviors. Recently, a new type of SSD, OpenChannel SSDs have been proposed to expose the internal parallelism and management of the SSD to the host and let the application user explicitly manage it [42, 253]. OpenChannel SSD exposes the SSD geometry and shares management responsibilities with the host, whereas conventional SSDs implement these trade secret in their confidential firmware. Therefore, the host can manage data placement, I/O scheduling, and garbage collection policy, etc explicitly. This allows developers to optimize I/O

performance based on its access pattern and QoS requirement at host side. LightNVM [43] is an user space library that allows developers to interact and configure OpenChannel SSDs. LightNVM provides an interface for performing vectorized I/O using physical addressing, which means reads/writes are performed on a set of physical blocks that span multiple parallel units of the SSD. Yet, there still does not exist an OpenChannel SSD standard in the industry, and there are only a few manufacturers actually producing Open-Channel SSDs (CNEX Labs [94]). ZNS [10], a OpenChannel variant that learns its lessons, focus more on the integration of existing standards and frameworks. It has been part of the most recent NVMe standard since 2020. ZNS defines zoned namespaces, with the constraint that logical blocks within a zone must be written sequentially. The benefits of ZNS come from the new interface that directs how data is written to physical flash while abstracting the unnecessary details of flash hardware from users. compared to conventional FTLs, ZNS user-space FTLs can be thinner, because it only needs to manage coarser-grained address translation (i.e., at the erasure block granularity instead of the regular page granularity), and it does not do garbage collection. As a result, performance variability and other impacts of garbage collection are eliminated. ZNS SSDs can get cheaper because the reserved capacity for over-provisioning (needed for garbage collection) is no longer needed, and the DRAM to buffer mapping tables is unnecessary. ZNS SSDs are available from several vendors, and large cloud providers are adopting them [40,119,173,207]. On the software side, many researchers are looking into ZNS-compatible filesystems, key-value stores, and development frameworks to adopt ZNS SSDs into production environments [126,155,173,177]. Recent NVMe specifications have introduced a predictable latency mode (PLM) [215]. PLM introduces notions of

deterministic windows and non-deterministic windows, within deterministic windows, users can expect predictable latencies for their I/O requests. NVMe controller manages the latency caused by background activities by limiting those activities to non-deterministic windows. During deterministic windows, SSD holds back background activities so that it can provide predictable performance. PLM introduces two NVMe commands that allow the host to query/alter the SSD state. This is a significant interface-level leap bridging the gap between host requirements and SSD status.

6.2 Improve Performance Isolation

SSDs have become indispensable for large-scale cloud services as they catch up with traditional mechanical disk in terms of cost and capacity and provide rich parallelism. However, the limitations of SSD firmware have hindered them from efficiently hosting multiple tenants on the same SSD. Previous work [118, 150, 264] had proposed novel techniques to help application tenants place their data such that underlying flash pages are allocated from separate blocks. This helps improve performance by reducing the write amplification factor (WAF). Lack of block sharing has the desirable side effect of clumping garbage into fewer blocks, leading to more efficient garbage collection (GC), thereby reducing tail latency of SSDs However, significant interference still exists between tenants because when data is striped, every tenant uses every channel, die and plane for storing data and the storage operations of one tenant can delay other tenants. And this type of isolation does not help a tenant with both random write and sequential workloads on the same dataset. Software isolation techniques can also help split SSD resources

more fairly. However, they cannot maximally utilize the flash parallelism when resource contention exists at a layer below because of the forced sharing of independent resources such as channels, dies, and planes. New SSD designs, such as open-channel SSDs that explicitly expose channels, dies and planes to the operating system [202, 253], can help tenants avoid some of these pitfalls by using dedicated channels.

6.3 Bypass Software Stack and NON-POSIX Interface

Storage continues to rely on the decades-old device interface that only allows read/write block, and even the file system (FS)-level is often limited to open()/read()/write()/close() semantics. With the emergence of many new storage component characteristics, such as byte addressability, compute offload, direct transfer to/from accelerators, it is becoming extremely difficult to efficiently utilize these features through existing interfaces. Yet the software stack is becoming the bottleneck for ulta-low latency and ultra-high throughput SSDs as they present to fast network devices. Research on reducing kernel overhead and eliminating slow kernel storage stack flourished in the past years [112, 120, 152, 158, 270]. One of the most famous and widely adopted framework is a kernel bypass frameworks called the Storage Performance Development Kit (SPDK). SPDK is a set of open-source toolkits and libraries for writing high-performance storage applications, which includes user-mode drivers and packages for NVMe. User-mode drivers can map the hardware I/O to memory accessible by the user mode so that the application can access the hardware without making system calls through the POSIX interface. SPDK processes I/O by using the user-mode for storage applications rather than the kernel

mode, thus minimizing the impact on CPU and memory bus. The overhead of context switch and interrupt handling can be eliminated. Therefore, designing proper systems level support for emerging hardware is very important to bring hardware performance benefits to application.

6.4 Simulation Platforms

There are generally four types of SSD simulation platforms. One only supports simulating low-level aspects of an SSD, such as Disksim [48], SSDSim [19], FlashSim [144]. These simulators model many internal mechanisms discussed in Chapter 2 and can approximate actual SSD's behavior to some degree. They offer a quick way to evaluate new designs and new parameters. Researcher use these simulators to compare the performance and energy consumption of SSD devices employing different FTL schemes. However, they are not suitable for today's NVMe SSDs because they are overly-simplified and far from capturing the critical features of high-performance NVMe SSD architectures. There is neither a specific flash organization nor an internal parallelism model. To remedy this, our simulator carefully considers and models the important functionalities of the underlying flash firmware, which have a great impact on SSD performance.

Another type of simulators simulates a complete I/O model and supports system-level studies by integration with full-system simulators. Examples are MQSim [241], SimpleSSD [96]. They are able to capture system-level behaviors by simulating the interaction with full system simulators like Gem5 [91]. This enables researchers to evaluate device-level designs over application level benchmarks and OS-level storage

stack. Our simulator targets at the same-level key SSD functionalities but lacks the ability of working in full-system mode. It is worth noting that all these simulators were built around the same time and targeting at emerging NVMe SSDs.

The third type is based on QEMU/KVM-based platform that emulates NAND flash latencies on a RAM disk, such as VSSIM [277] and FEMU [159]. They can be used to support kernel-level research such as guest OS modification upon the storage, and split-level research (both Guest OS and storage modifications). However, they do not model FTL in an accurate manner and face scalability issues.

The fourth type is hardware prototyping platforms that supports full-stack software/hardware research, such as Cosmos+ [281], OpenChannel SSD [203]. The Cosmos+ is an FPGA-based, full-fledged NVMe device with an open-sourced firmware which allows researchers to implement their designs. Hardware platforms return the most reliable results, however, the costs are high to implement a design on hardware.

6.5 In-storage Computing and Storage Encryption

In-storage Computing There have been several works exploring near-storage or in-storage processing for applications such as database query processing [77, 78, 134, 146, 225], key-value store [225], map-reduce workloads [98, 134], signal processing [45], and data analysis [243, 244]. They leverage the embedded CPU in the SSD's controller to perform compute operations to avoid data transfer overhead over PCIe. Unlike these applications, intelligent query workloads require support for complex compute operations such as FC and ConvD layers. Although it is possible to execute these workloads using

the wimpy embedded cores [98, 146, 225, 246], it is significantly slower than IceClave. Prior work has tried to place application specific hardware accelerators in the SSD such as [16,32,64,129,130]. However, to the best of our knowledge, we are the first to explore intelligent-query workloads for in-storage acceleration as well as to discuss the trade-offs for exploiting different levels of parallelism in the SSD.

In-Storage Accelerator. Several accelerator designs have been proposed to speed up the training and inference computation of popular DNN models [61,62,66,90,103,104, 110,128,149,162,168,195,209]. Additional optimizations such as quantization, weight pruning [103,274,285], data-flow optimizations [61,90,149] are discussed to speed up the computation and improve energy efficiency. However, none of these accelerators are incorporated in flash storage and are not optimized for intelligent query workloads. Although we do not perform any optimization like quantification, low-precision operations, and others, we believe the optimization work in the accelerator community can be incorporated into the IceClave architecture to gain higher performance and energy efficiency. We consider these possibilities as the extensions of our work.

Trusted Execution Environment To defend applications from malicious systems software, a variety of trusted hardware devices have been developed. A typical example is Intel SGX [36], which can create trusted execution environments for applications to run on untrusted operating systems. Because of the enabled security isolation, the SGX technique is being extended or customized to support various computing platforms [20, 29, 69, 147, 198, 234, 249] and applications [33, 143, 206, 223]. The hardware devices with

TPM [18] serve the similar purpose of security isolation by utilizing hardware support for attestation available in commodity processors from AMD and Intel [182, 183, 230]. As for ARM processors that are used in mobile computing platform and device controllers, they offer TrustZone that enables users to create secure world execution environment isolated from OS [47, 117, 222]. Unfortunately, none of these trusted hardware devices can be directly applied to in-storage computing, and defend against physical attacks. For instance, Guardat [249] and Pesos [147] allow users to specify security policies in storage devices to protect data confidentiality and integrity, the most recent work ShieldStore [143] and Speicher [33] applied SGX to key-value stores. However, none of them can protect the execution runtime of in-storage programs. In this work, we utilize the ARM processors available in SSD controllers and develop specific trusted execution environments for in-storage applications for security isolation with minimal hardware cost.

Storage Encryption and Security As we move computation closer to data in the storage devices, it would inevitably increase the trusted computing base, which poses security threats to user data. To protect sensitive user data while enabling near-data computing, a common approach is data encryption [208]. However, data leakage or loss would still happen at runtime, due to the lack of TEE support in modern SSD controllers. And adversaries can also initiate physical attacks to steal/destroy user data. An alternative approach is to enable computation on encrypted data [72,287]. However, such an approach requires intensive computing resource, which cannot be satisfied by modern SSD controllers due to the limited resource budget [98,178,225]. Our work IceClave presents a lightweight

approach that can enforce security isolation for in-storage applications as well as defend against physical attacks.

Chapter 7: Future Work

Software Support for Emerging Storage Technologies Emerging flash media [1, 224] always drives the scaling of higher I/O performance. Many new features are evolving as real industry standards in the NVMe specification on top of media advancement, such as NVMe Set [6, 22] for better resource isolation, NVMe I/O Determinism [216] for performance predictability, Zoned Namespace (ZNS) [10] for managing the device at host. Our simulator can provide quick evaluations on new designs and storage technology. Yet, all these featuresmay need additional system-level support and abstraction to be effectively exploited. Many performance benefits and management convenience can get compromised across the existing kernel software stack. A possible future direction is to revisit the POSIX interface and file system to abstract the emerging hardware complexities, which is orthogonal to our Zoned FTL work. Another direction is to redesign existing applications to directly exploit hardware by bypassing the thick software layer, similar approaches to eliminate software overhead have been very common in the field of network [60, 220].

Near Data Computing Near data computing has been a promising technique for accelerating data-intensive applications, especially for large-scale data processing and analytics [34,53,77,98,129,146,178,181,225,243]. They provide another place to process

data without consuming the precious host resources, thereby reducing the data transfer overhead and surpassing the memory wall problem. They are gaining more attention to be deployed in public clouds. However, when everyone designs its specialized near data computing devices, the lack of an interface standard makes (1) writing high-performance applications challenging as it requires both hardware and software level expertise to decide the computation part to be offloaded to the near data computing device. (2) a secure shared computing environment difficult as each application is exposed to basically the entire device. To this end, establishing a high-level interface for applications to be loaded in a trusted execution environment is extremely important to facilitate the deployment of near data computing devices in public clouds. Our work Iceclave analyzes the potential vulnerabilities of in-storage computing systems, and provides the first secure offloading interface and framework to address these threats.

Software Defined Storage As data center infrastructures are growing more complicated, the engineering difficulty in guaranteeing user QoS grows accordingly, as the infrastructure expands many layers along the I/O path, including OS, hypervisor, I/O scheduler, file system and heterogeneous distributed storage devices. Moreover, efficient and fair resource sharing in multi-tenant environments becomes more challenging due to increasing service demand. To overcome the these shortcomings, we propose Zoned FTL to leverage the device virtualization capabilities to enfore performance isolation. Another emerging solution called Software-Defined Storage can transfer more control functionality of conventional storage systems to the host. Systems can now utilize Software-Defined Flash to employ different levels of abstraction for flash management, such as timing of garbage collection processes, hot/cold separation, and physical data layout. However, obtaining these features

comes with tradeoff with respect to design difficulties, host resource consumption and scalability, compared to Zoned FTL which is closer to a conventional block device. Future research should consider the co-designing of application, OS and storage devices to enable more fine-grained I/O control.

Chapter 8: Conclusions

This dissertation proposes designs and methods at different levels of the storage stack to address the performance and security issues existed in today's cloud storage systems. These issues include the notorious tail latency problem of latency-critical I/O due to naive device sharing, lack of security isolation of security-critical in-storage programs, and lack of effective I/O control mechanisms for QoS-critical containers, etc. Such issues are overwhelmingly caused by the complex modern flash storage stack and lack of coordination from different layers.

We tackle these issues from the following aspects. For the performance issues, we develop an SSD simulator that enables accurate performance modeling and design space characterization of modern NVMe SSDs. Software/Hardware co-design approaches as well as OS-based resource control mechanisms are also proposed to achieve high-performance, low-tail operations as well as resource-conserving I/O controls. For security, we present a lightweight TEE which enables security isolation between in-storage programs and flash management to protect the security of in-storage programs and user data.

First, we develop a detailed event-based SSD simulator that models many low-level aspects and FTL algorithms, which is capable of approximating an accurate performance model of a commodity NVMe SSD. A modern SSD device, unlike its predecessor, is

structured as a complicated system that consists of an embedded processor, internal DRAM and multiple flash chips organized in a vertical hierarchy. It operates in a black-box manner to serve I/O requests, communicate with the host system and manage flash resources. Device vendors intentionally hide all the firmware details because these are their top trade secrets, which makes it difficult for system architects to understand the states of SSD internals or build a detailed performance model, let alone figure out the performance bottleneck for different workloads. The lack of a detailed SSD model also presents challenges for researchers to understand: (1) the key mechanisms that contribute to the high performance of modern NVMe SSDs; (2) how to design applications, firmware and OS for higher and more robust performance. As modern SSDs and their protocols evolve to meet the changing demands of data centers, the system community needs an SSD simulator that reliably models key features. Unfortunately, existing SSD simulators either lack accurate modeling of major performance factors, or fails to scale with more storage resources, resulting in significant deviation in terms of performance metrics compared to off-the-shelf products. To this end, we build our SSD simulator from scratch with accuracy and flexibility as the first priority. Our simulator is capable of modeling the flash internals including modern host interface, firmware, and storage backend in faithful details. We model a variety of performance factors of NVMe SSD including flash layout and internal parallelism, page allocation scheme, host interface, I/O transaction scheduling, FTL algorithms, and DRAM data caching. By modeling these major performance factors that are absent in existing simulators, our simulator is able to obtain more accurate simulation results than the existing ones. Although, due to many undisclosed technical details in the commodity SSDs and the sheer complexity of SSD systems, the simulation results may

still be different from real measurements. However, we can simulate on different set of flash configurations to approach the realistic performance curve of the off-the-shelf SSD. The improved accuracy of our simulator enables more accurate evaluation of the impact of any architectural change made to the SSD on overall performance metrics. Hence, our simulator can be used to study the performance impact of new design choices more effectively. We conduct a thorough SSD design space characterization over various I/O workloads, which sheds light on future SSD designs. We hope to persuade hardware, system and application designers that co-designing SSD, storage system and applications is desirable. In this dissertation, this simulator has also proven useful in other chapters where we use it evaluate the performance of SSD systems under various designs targeting at different storage layers.

Second, based on today's in-storage computing framework abstraction, we extract a realistic threat model to conclude its vulnerabilities, and propose a TEE framework to counter these threats. In-storage computing has been a promising technique for accelerating data-intensive applications, especially for large-scale data processing and analytics. It moves computation closer to the data stored in the storage devices like flash-based SSDs, such that it can overcome the I/O bottleneck by significantly reducing the amount of data transferred between the host machine and storage devices. As modern SSDs are employing general-purpose embedded processors and large DRAM in their controllers, it becomes feasible to enable in-storage computing in reality today. It has been proven to be an effective approach to alleviate the I/O bottleneck. To facilitate in-storage computing, many frameworks have been proposed. However, few of them treat the in-storage security as the first priority. Due to the lack of a trusted execution environment support in SSD

controllers, an offloaded (malicious) program could intervene offloaded programs, manage flash management, steal and/or compromise user data, which hinders the wide adoption of in-storage computing. In order to mitigate these threats posed on in-storage computing, we develop IceClave, a lightweight trusted execution environment. IceClave enables security isolation between in-storage programs and flash management functions based on ARM TrustZone technology. IceClave also defends potential physical attack by enforcing memory encryption and integrity verification of in-storage DRAM with low overhead. To protect data loaded from flash chips, IceClave develops a lightweight data encryption/decryption module in flash controllers. We evaluate IceClave design with a variety of data-intensive applications such as SQL queries and synthetic in-storage computing workloads. Compared to state-of-the-art in-storage computing approaches, IceClave introduces only 7.6% performance overhead, while enforcing security features with minimal hardware cost. IceClave still keeps the performance benefit of in-storage computing by delivering up to 2.31× better performance than the conventional host-based trusted computing approach. We strongly believe security should be treated equally as programmability and performance, especially for emerging storage systems that are yet to be deployed in massive production environment. There are more challenges existed in getting in-storage computing infrastructures deployed in public clouds, for example, when everyone designs their specialized near data computing devices, the lack of an interface standard makes (1) writing high-performance applications challenging as it requires both hardware and software level expertise to decide the computation part to be offloaded to the near data computing device. (2) a secure shared computing environment difficult as each application is exposed to basically the entire device. Therefore, it is extremely important to establish a high-level interface for applications to load into

a trusted execution environment to facilitate the deployment of in-storage computing infrastructures in the public cloud.

Third, to tackle the I/O performance interference problem in cloud systems, we propose a software/hardware co-design to enforce performance isolation by bridging the semantic gap between host and underlying storage. Resource sharing in cloud environments inherently causes 10x-100x longer tail latency due to underlying resource contention. We reveal the sources of performance degradation and instability by quantitative studies. The identified causes include the necessary internal management activities performed by the SSD firmware such as garbage collection and internal buffer flush that can block user I/Os to a delay in an order of milliseconds and incur contentions on the critical I/O path. These tail events slow down overall system performance and amounts to unreliable service. To this end, we propose ZFTL, which combines host-level modifications and device-level designs to eliminate interference from co-located tenants by enforcing isolation in I/O queues and internal SSD resources. Specifically, we expose an NVMe SSD as multiple physical instances, and attach each virtualized I/O service to a physical instance through existed virtualization techniques and slightly modified NVMe semantics. Unlike OpenChannel SSD that introduces a very different interface, our design targets at a balance point between the changes to existing storage stack and the transparency that allows host to manipulate SSD internals. Our evaluations show that ZFTL can improve throughput by 1.51x and reduce tail latency by up to 4.9x while preserving similar parallelism. Although this work does not completely eliminate the unpredictability of GC or buffer flush, we guarantee that user I/Os are at least not burdened by blocking background I/Os incurred by co-located tenants. SSD and storage system co-design remains an impactful research

direction around how to use these emerging devices to their full potential and build low-tail storage systems. The system community has invested significant research effort toward managing the ill effects of conventional SSDs' block interface with OpenChannel SSD and ZNS SSD. On the software side, a growing list of key-value stores, ZNS-compatible filesystem and development frameworks is developed to facilitate adoption of new devices.

Finally, we propose more effective I/O control strategies in the OS I/O stack. We illustrate that the state-of-the-art resource control mechanisms, used in Linux cgroups, are insufficient for allocating and limiting I/O resources under various scenarios. Inappropriate configurations may even hurt the performance of co-located workloads under memory intensive scenarios. We suggest more general and resource-conserving I/O controls are needed for heterogeneous storage devices in datacenters. While our approaches in Chapter 4 could provide performance isolation in the underlying storage, they couldn't help the I/O control and isolation needed in the kernel I/O stack. In principle, the kernel I/O stack is supposed to provide proportional I/O resources to containers based on weight. However, cgroup I/O management only functions at the block I/O layer, which leaves many I/O requests that are serviced by upper layers out of control. We reveal that inappropriate setups of cgroup may even hurt the performance of co-located workloads under I/O or memory intensive scenarios. To address the problem, we add direct page cache control to the cgroup memcontroller module; we modify the page reclamation scheme to support page allocation and eviction based on priority and weight.

The problems we investigated in this dissertation represent the cutting-edge research in the performance and security of storage systems. There are many promising further

research directions that can be built upon our work.

Emerging flash media has always been driving up new designs to unleash its full potential. Many new features are evolving into true industry standards in the NVMe specification alongside media advancement, such as NVMe Set for better resource isolation, NVMe I/O Determinism for predictable performance, and Zoned Namespace (ZNS) for finer-granularity device management. Our proposed simulators can provide quick assessments of new storage designs and technologies. However, all of these features may require additional system-level support and abstraction to work properly. Many of the performance and manageability benefits can be compromised in the existing software stack. Some existed work has suggested equipping SSDs with new external interfaces that give back control to applications. Another direction is to allow application to describe policies that the SSD should take. The abstraction can guarantee the separation of resources among I/O flows, while the SSD remains a black box for users. For latency-critical I/Os, researchers should seriously consider bypassing the thick software layer to exploit the low latency brought by the flash media.

Although we have seen many advances in accelerated data processing and modern data-intensive application programming frameworks, data processing still uses the traditional generic I/O model and ignore application-level data access patterns. The question we want to ask here is how the storage system needs to evolve to better support the performance of modern applications. As these applications have become the main consumers of CPU cycles in modern data centers, it is time to specialize and optimize the storage stack for them. One approach is to study data access patterns in different application phases (e.g. data shuffling in analytics, ML data processing pipelines, and machineless remote storage

access). Server and design specific optimizations such as caching, prefetch policies, and better programming models to reduce I/O bottlenecks.

Last but not least, the threat of hardware-based attacks must be evaluated on TEEs like IceClave before the TEE is deployed. The most severe threat to TEEs has been side-channel attacks. Such attack aim to obtain information by exploiting indirect effect of the victim hardware's execution. These information can be usually gathered by physical measurements, such as power, electromagnetic, and timing side-channels. Side-channel attacks have been reportedly successful in recovering security critical information from TEEs like TrustZone and SGX. Therefore, it is crucial to verify the side-channel leakages of in-storage TEEs before it is deployed.

In summary, we have explored the potential to improve the performance of SSDs as a storage media and the possibility to enable trusted execution in the in-storage computing hosted in SSDs. In addition, our work also paves the way for further research to address the key challenges faced by SSD storage systems.

Bibliography

[1] Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[2] Intel SSD DC P4510 Series. https://ark.intel.com/content/www/us/en/ark/products/series/122570/intel-ssd-dc-p4510-series.html.

[3] Linux I/O Stack. https://wxdublin.gitbooks.io/deep-into-linux-and-beyond/content/io.html.

[4] Micron 9300 MAX. https://www.micron.com/products/ssd/product-lines/9300.

[5] Nvme command set. https://nvmexpress.org/developers/nvme-command-set-specifications/.

[6] NVMe Whitepaper. https://nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf.

[7] PCI-Express Gen 4. https://en.wikipedia.org/wiki/PCI_Express.

[8] Samsung Z-SSD. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[9] What is a namespace? https://nvmexpress.org/resources/nvm-express-technology-features/nvme-namespaces/.

[10] Zoned namespaces (zns) ssds. https://zonedstorage.io/introduction/zns/.

[11] Generating recommendations at amazon scale with apache spark and amazon dsstne. 2016.

[12] Scaleflux computational storage. https://www.scaleflux.com/, 2016.

[13] Intel SSD DC P4500 Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/dc-p4500-series.html, 2017.

[14] Samsung smartssd. https://forums.xilinx.com/t5/Xilinx-Xclusive-Blog/Samsung-Electronics-Unveils-Xilinx-Based-SmartSSD ba-p/902137, 2018.

[15] Samsung Z Nand MZPJB960HMGC. https://www.samsung.com/semiconductor/ssd/z-ssd/, 2018.

[16] See our machine learning accelerator at embedded world. https://blog.westerndigital.com/machine-learning-accelerator-embedded- 2019.

[17] Intel software guard extensions. https://software.intel.com/en-us/sgx, 2020.

[18] Tpm 2.0 library specification. https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2020.

[19] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceeding of the USENIX 2008 Annual Technical Conference (USENIX ATC'08)*, Boston, MA, June 2008.

[20] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS'18*, 2018.

[21] Jasmin Ajanovic. Pci express 3.0 overview. In *Proceedings of Hot Chip: A Symposium on High Performance Chips*, volume 69, page 143, 2009.

[22] JM David Allen and J Metz. The evolution and future of nvme. *Webminar, Jan 2018*, 2018.

[23] EC Amazon. Amazon. *See https://aws. amazon. com/ec2/(15 June 2018)*, 2006.

[24] Anandtech. Memblaze's pblaze5 x26: Toshiba's xl-flash-based ultra-low latency ssd, 2019.

[25] Arm. Arm corelink tzc-400 trustzone address space controller. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0504c/DDI0504C_tzc400_r0p1_trm.pdf, 2013.

[26] ARM. Arm architecture reference manual for armv8-a, 2020.

[27] ARM. Arm storage. https://www.arm.com/solutions/storage, 2020.

[28] Arm. Arm trustzone technology. https://developer.arm.com/ip-products/security-ip/trustzone, 2020.

[29] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, November 2016.

[30] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 104–115, 2019.

[31] Jens Axboe. Linux block io—present and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[32] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. Intelligent SSD: A Turbo for Big Data Mining. In *Proceedings of the 22nd ACM International Conference of Information Knowledge Management (CIKM'13)*, San Francisco, CA, October 2013.

[33] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: Securing lsm-based key-value stores using shielded execution. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*, Boston, MA, February 2019.

[34] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro*, 34(4), 2014.

[35] Salman A Baset. Cloud slas: present and future. *ACM SIGOPS Operating Systems Review*, 46(2):57–66, 2012.

[36] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, October 2014.

[37] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005.

[38] Janki Bhimani, Zhengyu Yang, Ningfang Mi, Jingpei Yang, Qiumin Xu, Manu Awasthi, Rajinikanth Pandurangan, and Vijay Balakrishnan. Docker container scheduler for i/o intensive applications running on nvme ssds. *IEEE Transactions on Multi-Scale Computing Systems*, 4(3):313–326, 2018.

[39] Matias Bjørling. From open-channel ssds to zoned namespaces. In *Proc. Linux Storage Filesyst. Conf.(Vault)*, page 1, 2019.

[40] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. {ZNS}: Avoiding the block interface tax for flash-based {SSDs}. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021.

[41] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

[42] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.

[43] 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, 2017.

[44] Filip Blagojević, Cyril Guyot, Qingbo Wang, Timothy Tsai, Robert Mateescu, and Zvonimir Bandić. Priority {IO} scheduling in the cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)*, 2013.

[45] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman. Active Flash: Out-of-core Data Analytics on Flash Storage. In *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*, Monterey, CA, April 2012.

[46] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. Dr. sgx: Automated and adjustable side-channel protection for sgx using data location randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 788–800, 2019.

[47] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS'19)*, San Diego, CA, 2019.

[48] John S Bucy, Gregory R Ganger, et al. *The DiskSim simulation environment version 3.0 reference manual*. School of Computer Science, Carnegie Mellon University, 2003.

[49] Keith Busch. Linux nvme driver, 2013.

[50] Werner Bux and Ilias Iliadis. Performance of greedy garbage collection in flash-based solid-state drives. *Performance Evaluation*, 67(11):1172–1186, 2010.

[51] Werner Bux and Ilias Iliadis. Performance of greedy garbage collection in flash-based solid-state drives. *Performance Evaluation*, 67(11):1172–1186, 2010. Performance 2010.

[52] Hong Cai, Ning Wang, and Ming Jun Zhou. A transparent approach of enabling saas multi-tenancy in the cloud. In *2010 6th World Congress on Services*, pages 40–47. IEEE, 2010.

[53] 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, February 2020.

[54] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[55] Li-Pin Chang and Chun-Da Du. Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 15(1):1–36, 2009.

[56] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Improving flash wear-leveling by proactively moving static data. *IEEE Transactions on Computers*, 59(1):53–65, 2010.

[57] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth Pugsley, Aniruddha Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep*, 2012.

[58] Rakesh Cheerla. Computational ssds. *Storage Networking Industry Association*, 2019.

[59] Renhai Chen, Qiming Guan, Chenlin Ma, and Zhiyong Feng. Delay-based i/o request scheduling in ssds. *Journal of Systems Architecture*, 98:434–442, 2019.

[60] Ruining Chen and Guoao Sun. A survey of kernel-bypass techniques in network stack. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, pages 474–477, 2018.

[61] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits (SSC'17)*, 52(1), Jan 2017.

[62] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*, Cambridge, England, May 2014.

[63] Benjamin Y. Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. XSD: Accelerating MapReduce by Harnessing the GPU inside an SSD. In *Proceedings of the 1st Workshop on Near-Data Processing in Conjunction with the 46th IEEE/ACM International Symposium on Microarchitecture (WoNDP)*, Davis, CA, December 2013.

[64] Benjamin Y. Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. XSD: Accelerating MapReduce by Harnessing the GPU inside an SSD. In *Proceedings of the 1st Workshop on Near-Data Processing in Conjunction with the 46th IEEE/ACM International Symposium on Microarchitecture (WoNDP)*, Davis, CA, December 2013.

[65] Yongjae Chun, Kyeore Han, and Youpyo Hong. High-performance multi-stream management for ssds. *Electronics*, 10(4):486, 2021.

[66] Jason Clemons, Chih-Chi Cheng, Iuri Frosio, Daniel Johnson, and Stephen W Keckler. A Patch Memory System for Image Processing and Computer Vision. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, Taipei, Taiwan, October 2016.

[67] Marshall Copeland, Julian Soh, Anthony Puca, Mike Manning, and David Gollob. Microsoft azure. *New York, NY, USA:: Apress*, 2015.

[68] Antonio Corradi, Mario Fanelli, and Luca Foschini. Vm consolidation: A real case based on openstack cloud. *Future Generation Computer Systems*, 32:118–127, 2014.

[69] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[70] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C., August 2003. USENIX Association.

[71] Carlo A Curino, Djellel E Difallah, Andrew Pavlo, and Philippe Cudre-Mauroux. Benchmarking oltp/web databases in the cloud: The oltp-bench framework. In

Proceedings of the fourth international workshop on Cloud data management, pages 17–20, 2012.

[72] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious coopetitive analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[73] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. gonzalez, and Ion Stoica. Oblivious coopetive analytics using hardware enclaves. In *Proceedings of European Conference on Computer Systems (EuroSys'20)*, Crete, Greece, March 2020.

[74] Christophe De Canniere and Bart Preneel. Trivium specifications. In *eSTREAM, ECRYPT Stream Cipher Project*, 2005.

[75] Peter Desnoyers. Analytic modeling of ssd write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–10, 2012.

[76] Western Digital. Risc-v: Accelerating next-generation compute requirements. `https://www.westerndigital.com/company/innovations/risc-v`, 2019.

[77] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, New York, NY, 2013.

[78] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, New York, NY, USA, 2013.

[79] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, volume 2, 2008.

[80] Chris Down. 5 years of cgroup v2: The future of linux resource control. 2021.

[81] Yaakoub El-Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. Exploring the performance fluctuations of hpc workloads on clouds. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 383–387. IEEE, 2010.

[82] Ahmed Elnably, Kai Du, and Peter Varman. Reward scheduling for qos in cloud applications. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 98–106. IEEE, 2012.

[83] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T Kandemir, Chita R Das, and Myoungsoo Jung. Exploiting intra-request slack to improve ssd performance. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 375–388, 2017.

[84] Gerion Entrup, Felix Herrmann, Sophie Matter, Elias Entrup, Matthias Jakob, Jan Eberhardt, and Mathias Casselt. Architecture of the linux kernel. 2018.

[85] K Eshghi and Rino Micheloni. Ssd architecture and pci express interface. In *Inside solid state drives (SSDs)*, pages 19–45. Springer, 2013.

[86] WU Fengguang, XI Hongsheng, and XU Chenfeng. On the design of a new linux readahead framework. *ACM SIGOPS Operating Systems Review*, 42(5):75–84, 2008.

[87] RISC-V Foundation. The risc-v instruction set manual. `https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf`, 2017.

[88] Congming Gao, Liang Shi, Chun Jason Xue, Cheng Ji, Jun Yang, and Youtao Zhang. Parallel all the time: Plane level parallelism exploration for high performance ssds. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 172–184. IEEE, 2019.

[89] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H-M Sha. Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2014.

[90] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’17)*, Xi’an, China, April 2017.

[91] gem5 development team. gem5 simulator, 2020.

[92] Mohammad Hossein Ghahramani, MengChu Zhou, and Chi Tin Hon. Toward cloud computing qos architecture: Analysis of cloud systems and cloud services. *IEEE/CAA Journal of Automatica Sinica*, 4(1):6–18, 2017.

[93] Javier González and Matias Bjørling. Multi-tenant i/o isolation with open-channel ssds. In *Nonvolatile Memory Workshop (NVMW)*, 2017.

[94] Javier González, Matias Bjørling, Seongno Lee, Charlie Dong, and Yiren Ronnie Huang. Application-driven flash translation layers on open-channel ssds. In *Proceedings of the 7th non Volatile Memory Workshop (NVMW)*, pages 1–2, 2016.

[95] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Bruce Perens' Open Source series. Prentice Hall, 2004.

[96] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber*: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–481. IEEE, 2018.

[97] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 217–233, 2017.

[98] B. Gu, A. S. Yoon, D. H. Bae, I. Jo, J. Lee, J. Yoon, J. U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. Biscuit: A framework for near-data processing of big data workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*, Seoul, Korea, June 2016.

[99] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, Washington, DC, USA, 2009.

[100] Hyunho Gwak, Yunji Kang, and Dongkun Shin. Reducing garbage collection overhead of log-structured file systems with gc journaling. In *2015 International Symposium on Consumer Electronics (ISCE)*, pages 1–2. IEEE, 2015.

[101] Gareth Halfacree. Sifive's risc-v cores launch in two ssd families. `https://www.bit-tech.net/news/tech/storage/sifives-risc-v-cores-launch-in-two-ssd-families/1/`, 2018.

[102] Dayne Hammes, Hiram Medero, and Harrison Mitchell. Comparison of nosql and sql databases in the cloud. *Proceedings of the Southern Association for Information Systems (SAIS), Macon, GA*, pages 21–22, 2014.

[103] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*, Seoul, Republic of Korea, 2016.

[104] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'16)*, Vancouver, Canada, November 2016.

[105] Mohamed Hanini, Said El Kafhali, and Khaled Salah. Dynamic vm allocation and traffic control to manage qos and energy consumption in cloud computing environment. *International Journal of Computer Applications in Technology*, 60(4):307–316, 2019.

[106] Bryan Harris and Nihat Altiparmak. {Ultra-Low} latency {SSDs}'impact on overall energy efficiency. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[107] Dan He, Fang Wang, Hong Jiang, Dan Feng, Jing Ning Liu, Wei Tong, and Zheng Zhang. Improving hybrid ftl by fully exploiting internal ssd parallelism with virtual blocks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–19, 2014.

[108] Dan He, Fang Wang, Hong Jiang, Dan Feng, Jing Ning Liu, Wei Tong, and Zheng Zhang. Improving hybrid ftl by fully exploiting internal ssd parallelism with virtual blocks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–19, 2014.

[109] Qinlu He, Genqing Bian, Weiqi Zhang, Fenglang Wu, and Zhen Li. Tcftl: Improved real-time flash memory two cache flash translation layer algorithm. *Journal of Nanoelectronics and Optoelectronics*, 16(3):403–413, 2021.

[110] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W Fletcher. Morph: Flexible Acceleration for 3D CNN-based Video Understanding. In *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'18)*, Fukuoka, Japan, October 2018.

[111] Christoph Hellwig. High performance storage with blkmq and scsi-mq, 2014.

[112] Michael Hennecke. Daos: A scale-out high performance storage stack for storage class memory. *Supercomputing frontiers*, page 40, 2020.

[113] Duwon Hong, Myungsuk Kim, Jisung Park, Myoungsoo Jung, and Jihong Kim. Improving ssd performance using adaptive restricted-copyback operations. In *2019 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2019.

[114] Duwon Hong, Myungsuk Kim, Jisung Park, Myoungsoo Jung, and Jihong Kim. Improving ssd performance using adaptive restricted-copyback operations. In *2019 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2019.

[115] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the international conference on Supercomputing*, pages 96–107, 2011.

[116] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Shuping Zhang, Jingning Liu, Wei Tong, Yi Qin, and Liuzheng Wang. Achieving page-mapping ftl performance at block-mapping ftl cost by hiding address translation. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2010.

[117] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vtz: Virtualizing ARM trustzone. In *26th USENIX Security Symposium (USENIX Security'17)*, Vancouver, BC, August 2017.

[118] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17)*, Santa clara, CA, 2017.

[119] J Hwang. Towards even lower total cost of ownership of data center it infrastructure. In *Proceedings of the NVRAMOS Workshop, Jeju, Korea*, pages 24–26, 2019.

[120] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. {TCP}{}{RDMA}:{CPU-efficient} remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, 2020.

[121] Intel. Running average power limit. `https://en.wikipedia.org/wiki/Perf_(Linux)#RAPL`, 2014.

[122] Intel. Pin - a dynamic binary instrumentation tool, 2018.

[123] Kee-Hoon Jang and Tae Hee Han. Efficient garbage collection policy and block management method for nand flash memory. In *2010 2nd International Conference on Mechanical and Electronics Engineering*, volume 1, pages V1–327. IEEE, 2010.

[124] Jhuyeong Jhin, Hyukjoong Kim, and Dongkun Shin. Optimizing host-level flash translation layer with considering storage stack of host systems. In *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*, pages 1–4, 2018.

[125] Song Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu Chen. S-ftl: An efficient address translation for flash memory by exploiting spatial locality. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2011.

[126] Peiquan Jin, Xiangyu Zhuang, Yongping Luo, and Mingchen Lu. Exploring index structures for zoned namespaces ssds. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 5919–5922. IEEE, 2021.

[127] Kanchan Joshi, Kaushal Yadav, and Praval Choudhary. Enabling {NVMe}{WRR} support in linux block layer. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[128] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, Toronto, Canada, 2017.

[129] S. Jun, A. Wright, S. Zhang, S. Xu, and Arvind. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*, Los Angeles, CA, June 2018.

[130] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. BlueDBM: An Appliance for Big Data Analytics. *SIGARCH Comput. Archit. News*, 43(3), June 2015.

[131] Dawoon Jung, Jeong-UK Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. Superblock ftl: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(4):1–41, 2010.

[132] Myoungsoo Jung and Mahmut T Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *HotStorage*, 2012.

[133] Myoungsoo Jung and Mahmut T. Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 524–535, 2014.

[134] Y. Kang, Y. Kee, E. L. Miller, and C. Park. Enabling cost-effective data processing with smart SSD. In *Proceedings of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST'13)*, Lake Arrowhead, CA, May 2013.

[135] Supriya Karmakar. Quantum dot gate non-volatile memory as single level cell (slc), multi-level cell (mlc) and triple level cell (tlc). In *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–5. IEEE, 2018.

[136] Sangeeth Keeriyadath. Nvme virtualization ideas for machines on cloud. In *Storage Developer Conference 2016*, pages 24–27, 2016.

[137] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, October 2018.

[138] Jaeho Kim, Donghee Lee, and Sam H Noh. Towards {SLO} complying ssds through {OPS} isolation. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 183–189, 2015.

[139] Jihun Kim, Joonsung Kim, Pyeongsu Park, Jong Kim, and Jangwoo Kim. Ssd performance modeling using bottleneck analysis. *IEEE Computer Architecture Letters*, 17(1):80–83, 2017.

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

[141] Kyu Sik Kim and Tae Seok Kim. Performance evaluation of hmb-supported dram-less nvme ssds. *KIPS Transactions on Computer and Communication Systems*, 8(7):159–166, 2019.

[142] Shine Kim, Jonghyun Bae, Hakbeom Jang, Wenjing Jin, Jeonghun Gong, Seungyeon Lee, Tae Jun Ham, and Jae W Lee. Practical erase suspension for modern low-latency {SSDs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 813–820, 2019.

[143] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys'19)*, 2019.

[144] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *2009 First International Conference on Advances in System Simulation*, pages 125–131. IEEE, 2009.

[145] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[146] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*, Cambridge, Massachusetts, 2017.

[147] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*, 2018.

[148] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Janus-ftl: Finding the optimal point on the spectrum between page and block mapping schemes. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 169–178, 2010.

[149] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. *SIGPLAN Not.*, 53(2), March 2018.

[150] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. Dc-store: eliminating noisy neighbor containers using deterministic i/o performance and resource isolation. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 183–191, 2020.

[151] Paul Lassa. The new ez nand in onfi v2. 3. *SanDisk-Flash Memory Summit-Aug*, 2010.

[152] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous {I/O} stack: A low-latency kernel {I/O} stack for {Ultra-Low} latency {SSDs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 603–616, 2019.

[153] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. A semi-preemptive garbage collector for solid state drives. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 12–21, 2011.

[154] Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in flash memory ssd technology for enterprise database applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 863–870, 2009.

[155] Youngjae Lee, Jeeyoon Jung, and Dongkun Shin. Buffered i/o support for zoned namespace ssd. In *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pages 1–4. IEEE, 2021.

[156] Juri Lelli, Claudio Scordino, Luca Abeni, and Dario Faggioli. Deadline scheduling in the linux kernel. *Software: Practice and Experience*, 46(6):821–839, 2016.

[157] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[158] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK 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*, pages 591–605, 2020.

[159] 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 *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*, pages 83–90, 2018.

[160] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009.

[161] Youhuizi Li, Jiancheng Zhang, Congfeng Jiang, Jian Wan, and Zujie Ren. Pine: Optimizing performance isolation in container environments. *IEEE Access*, 7:30410–30422, 2019.

[162] Youjie Li, Xiaohao Wang, Iou-Jen Liu, Deming Chen, Alexander Schwing, and Jian Huang. Accelerating Distributed Reinforcement Learning with In-Switch Computing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*, Phoenix, AZ, June 2019.

[163] Shuang Liang. *Algorithms designs and implementations for page allocation in SSD firmware and SSD caching in storage systems*. PhD thesis, The Ohio State University, 2010.

[164] Heerak Lim, Hwajung Kim, Kihyeon Myung, Heon Young, and Yongseok Son. Isokv: An isolation scheme for key-value stores by exploiting internal parallelism in ssd. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 247–256. IEEE, 2019.

[165] Helger Lipmaa, Phillip Rogaway, and David Wagner. Ctr-mode encryption. In *First NIST Workshop on Modes of Operation*, volume 39. Citeseer. MD, 2000.

[166] Chun-Yi Liu, Yunju Lee, Myoungsoo Jung, Mahmut Taylan Kandemir, and Wonil Choi. Prolonging 3d nand ssd lifetime via read latency relaxation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 730–742, 2021.

[167] Jiahao Liu, Fang Wang, and Dan Feng. Costpi: Cost-effective performance isolation for shared nvme ssds. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[168] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An Instruction Set Architecture for Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*, Seoul, South Korea, June 2016.

[169] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[170] CC Lu, CC Cheng, HP Chiu, WL Lin, TW Chen, SH Ku, Wen-Jer Tsai, TC Lu, KC Chen, Tahui Wang, et al. Analysis and realization of tlc or even qlc operation with a high performance multi-times verify scheme in 3d nand flash memory. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 2–2. IEEE, 2018.

[171] Lanyue Lu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 31–44, 2013.

[172] Ray Lucchesi. Ssd flash drives enter the enterprise. *Silverton Consulting. accessed on*, 8:2008, 2011.

[173] Corne Lukken, Giulia Frascaria, and Animesh Trivedi. Zcsd: a computational storage device over zoned namespaces (zns) ssds. *arXiv preprint arXiv:2112.00142*, 2021.

[174] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*, Munich, Germany, September 2018.

[175] Ruixiang Ma, Fei Wu, Meng Zhang, Zhonghai Lu, Jiguang Wan, and Changsheng Xie. Rber-aware lifetime prediction scheme for 3d-tlc nand flash memory. *IEEE Access*, 7:44696–44708, 2019.

[176] T Madhuri and P Sowjanya. Microsoft azure v/s amazon aws cloud services: A comparative study. *International Journal of Innovative Research in Science, Engineering and Technology*, 5(3):3904–3907, 2016.

[177] Umesh Maheshwari. From blocks to rocks: A natural extension of zoned namespaces. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pages 21–27, 2021.

[178] Vikram Sharma Mailthoday, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wenmei Hwu. DeepStore: In-Storage Acceleration for Intelligent

Queries. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*, Columbus, OH, October 2019.

[179] Saif UR Malik, Samee U Khan, and Sudarshan K Srinivasan. Modeling and analysis of state-of-the-art vm-based cloud management platforms. *IEEE Transactions on Cloud Computing*, 1(1):1–1, 2013.

[180] Kevin Marks. An nvm express tutorial. *Flash Memory Summit*, 2013.

[181] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. GraphSSD: Graph Semantics Aware SSD. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA'19)*, Phoenix, AZ, 2019.

[182] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *2010 IEEE Symposium on Security and Privacy (Oakland'10)*, 2010.

[183] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'08)*, 2008.

[184] Micron. Micron 3d nand flash memory, 2019.

[185] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: random write considered harmful in solid state drives. In *FAST*, volume 12, pages 1–16, 2012.

[186] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[187] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A Tool to Model Large Caches. *HP laboratories*, 2009.

[188] Eyee Hyun Nam, Bryan Suk Joon Kim, Hyeonsang Eom, and Sang Lyul Min. Ozone (o3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers*, 60(5):653–666, 2011.

[189] Shiqiang Nie, Youtao Zhang, Weiguo Wu, Chi Zhang, and Jun Yang. Dir: Dynamic request interleaving for improving the read performance of aged ssds. In *2019 IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2019.

[190] Mais Nijim, Ziliang Zong, Xiao Qin, and Yousef Nijim. Multi-layer prefetching for hybrid storage systems: algorithms, models, and evaluations. In *2010 39th international conference on parallel processing workshops*, pages 44–49. IEEE, 2010.

[191] Isaac Odun-Ayo, Sanjay Misra, Olusola Abayomi-Alli, and Olasupo Ajayi. Cloud multi-tenancy: Issues and developments. In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 209–214, 2017.

[192] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting {SGX} enclaves from practical side-channel attacks. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 227–240, 2018.

[193] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[194] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2017.

[195] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. *SIGARCH Comput. Archit. News*, 45(2), June 2017.

[196] K Parat and A Goda. Scaling trends in nand flash. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 2–1. IEEE, 2018.

[197] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):1–23, 2008.

[198] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh. Nested enclave: Supporting fine-grained hierarchical isolation with sgx. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789, 2020.

[199] Jae-Woo Park, Doogon Kim, Sunghwa Ok, Jaebeom Park, Taeheui Kwon, Hyunsoo Lee, Sungmook Lim, Sun-Young Jung, Hyeongjin Choi, Taikyu Kang, et al. A 176-stacked 512gb 3b/cell 3d-nand flash with 10.8 gb/mm 2 density with a peripheral circuit under cell array architecture. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 422–423. IEEE, 2021.

[200] Jong-Hyeok Park, Soyee Choi, Gihwan Oh, and Sang-Won Lee. Sas: Ssd as sql database system. *Proceedings of the VLDB Endowment*, 14(9):1481–1488, 2021.

[201] Jung Kyu Park and Jaeho Kim. A method for reducing garbage collection overhead of ssd using machine learning algorithms. In *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 775–777. IEEE, 2017.

[202] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-channel ssd (what is it good for). In *CIDR*, 2020.

[203] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-channel ssd (what is it good for). In *CIDR*, 2020.

[204] Nick Piggin. A lockless page cache in linux. In *Proceedings of the Linux Symposium*, volume 2. Citeseer, 2006.

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

[206] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (Oakland'18)*, 2018.

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

[208] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 333–348, Bellevue, WA, August 2012. USENIX Association.

[209] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, Taipei, Taiwan, April 2016.

[210] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 183–196. IEEE, 2007.

[211] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186:70, 2013.

[212] Rami Rosen. Linux containers and the future cloud. *Linux J*, 240(4):86–95, 2014.

[213] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[214] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[215] Tanaya Roy, Jit Gupta, Krishna Kant, Amitangshu Pal, and Dave Minturn. Plm light: Emulating predictable latency mode in regular ssds. In *2021 IEEE 20th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2021.

[216] Tanaya Roy, Jit Gupta, Krishna Kant, Amitangshu Pal, Dave Minturn, and Arash Tavakkol. Plmc: A predictable tail latency mode coordinator for shared nvme ssd with multiple hosts. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–6. IEEE, 2021.

[217] Alessandro Rubini. Kernel korner: The" virtual file system" in linux. *Linux Journal*, 1997(37es):21–es, 1997.

[218] Alessandro Rubini and Jonathan Corbet. *Linux device drivers*. " O'Reilly Media, Inc.", 2001.

[219] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[220] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S Berger, James C Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 152–158, 2021.

[221] Gururaj Saileshwar, Prashant Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose Joao, and Moinuddin Qureshi. Morphable counters: Enabling compact integrity trees for low-overhead secure memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 416–427. IEEE, 2018.

[222] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*, 2014.

[223] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *2015 IEEE Symposium on Security and Privacy (Oakland'15)*, 2015.

[224] Abu Sebastian, Manuel Le Gallo, and Evangelos Eleftheriou. Computational phase-change memory: Beyond von neumann computing. *Journal of Physics D: Applied Physics*, 52(44):443002, 2019.

[225] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, 2014.

[226] Narges Shahidi, Mahmut T. Kandemir, Mohammad Arjomand, Chita R. Das, Myoungsoo Jung, and Anand Sivasubramaniam. Exploring the potentials of parallel garbage collection in ssds for enterprise storage systems. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 561–572, 2016.

[227] Weidong Shi, H-HS Lee, Mrinmoy Ghosh, and Chenghuai Lu. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 123–134. IEEE, 2004.

[228] Xuanhua Shi, Wei Liu, Ligang He, Hai Jin, Ming Li, and Yong Chen. Optimizing the ssd burst buffer by traffic detection. *ACM Transactions on Architecture and Code Optimization (TACO)*, 17(1):1–26, 2020.

[229] Anton Shilov. Samsung to use sifive risc-v cores for socs, automotive, 5g applications.
`https://www.anandtech.com/show/15228/samsung-to-use-riscv-cores`, 2019.

[230] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*, 2011.

[231] Kent Smith. Understanding ssd over-provisioning. *EDN Network*, 2013.

[232] Software and Ludwig-Maximilians-Universität München Computational Systems Lab. CPU Energy Meter. `https://github.com/sosy-lab/cpu-energy-meter`, 2020.

[233] Richard Solomon. Pci express [product reviews]. *IEEE Consumer Electronics Magazine*, 8(3):97–98, 2019.

[234] Stavros Volos and Kapil Vaswani and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, Carlsbad, CA, October 2018.

[235] G Edward Suh, Dwaine Clarke, Blaise Gasend, Marten Van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 339–350. IEEE, 2003.

[236] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368, 2003.

[237] Shivam Swami, Joydeep Rakshit, and Kartik Mohanram. Stash: Security architecture for smart hybrid memories. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[238] László Szekeres, Mathias Payer, Tao Wei, and Dawn Xiaodong Song. Sok: Eternal war in memory. *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[239] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 665–678, 2018.

[240] Ken Takeuchi. Novel co-design of nand flash memory and nand flash controller circuits for sub-30 nm low-power high-speed solid-state drives (ssd). *IEEE Journal of solid-state circuits*, 44(4):1227–1234, 2009.

[241] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. {MQSim}: A framework for enabling realistic studies of modern {Multi-Queue}{SSD} devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 49–66, 2018.

[242] Arash Tavakkol, Pooyan Mehrvarzy, Mohammad Arjomand, and Hamid Sarbazi-Azad. Performance evaluation of dynamic page allocation strategies in ssds. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 1(2):1–33, 2016.

[243] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, San Jose, CA, 2013.

[244] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. Reducing Data Movement Costs Using Energy Efficient, Active Computation on SSD. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*, Hollywood, CA, 2012.

[245] tomshardware. Security flaws found in intel software, data center ssds, `https://www.tomshardware.com/news/intel-security-vulnerabilities-processor-diagnostic-tool-ssd,39845.html`.

[246] H. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *Proceedings*

of the 43rd IEEE Annual International Symposium on Computer Architecture (ISCA'16), Taipei, Taiwan, June 2016.

[247] Hung-Wei Tseng, Laura M. Grupp, and Steven Swanson. Underpowering nand flash: Profits and perils. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*, 2013.

[248] Alexandru Uta and Harry Obaseki. A performance study of big data workloads in cloud datacenters with network variability. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 113–118, 2018.

[249] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15)*, 2015.

[250] Paolo Valente. The two new {I/O} controllers and {BFQ}. 2020.

[251] Paolo Valente and Arianna Avanzini. Evolution of the bfq storage-i/o scheduler. In *2015 Mobile Systems Technologies Workshop (MST)*, pages 15–20. IEEE, 2015.

[252] Rik Van Riel. Page replacement in linux 2.4 memory management. In *USENIX Annual Technical Conference, FREENIX Track*, pages 165–172, 2001.

[253] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*, Amsterdam, The Netherlands, April 2014.

[254] Shunzhuo Wang, You Zhou, Jiaona Zhou, Fei Wu, and Changsheng Xie. An efficient data migration scheme to optimize garbage collection in ssds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(3):430–443, 2020.

[255] Wei Wang and Tao Xie. Pcftl: A plane-centric flash translation layer utilizing copy-back operations. *IEEE Transactions on parallel and distributed systems*, 26(12):3420–3432, 2014.

[256] Stephen A. Weis. Protecting data in-use from firmware and physical attacks. 2014.

[257] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*, 2020.

[258] Guanying Wu and Xubin He. Reducing ssd read latency via nand flash program and erase suspension. In *FAST*, volume 12, pages 10–10, 2012.

[259] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015.

[260] Zhiyong Xu, Ruixuan Li, and Cheng-Zhong Xu. Cast: A page-level ftl with compact address mapping and parallel data blocks. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, pages 142–151. IEEE, 2012.

[261] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. *ACM SIGARCH Computer Architecture News*, 34(2):179–190, 2006.

[262] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. *ACM SIGARCH Computer Architecture News*, 34(2):179–190, 2006.

[263] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Transactions on Storage (TOS)*, 13(3):1–26, 2017.

[264] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. Autostream: automatic stream management for multi-streamed ssds. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–11, 2017.

[265] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. {Don't} stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014.

[266] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. Garbage collection and wear leveling for flash memory: Past and future. In *2014 International Conference on Smart Computing*, pages 66–73, 2014.

[267] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. Reducing garbage collection overhead in {SSD} based on workload prediction. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[268] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, and Mark Horowitz. DNN Dataflow Choice is Overrated. *arXiv preprint arXiv:1809.04070*, 2018.

[269] Yudong Yang, Vishal Misra, and Dan Rubenstein. On the optimality of greedy garbage collection for ssds. *ACM SIGMETRICS Performance Evaluation Review*, 43(2):63–65, 2015.

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

[271] Yingbiao Yao, Jinlong Fan, Jie Zhou, Xiaochong Kong, and Nenghua Gu. Hdftl: An on-demand flash translation layer algorithm for hybrid solid state drives. *IEEE Transactions on Consumer Electronics*, 67(1):50–57, 2021.

[272] Yingbiao Yao, Xiaochong Kong, Jie Zhou, Xiaorong Xu, Wei Feng, and Zhaoting Liu. An advanced adaptive least recently used buffer management algorithm for ssd. *IEEE Access*, 7:33494–33505, 2019.

[273] Zhihao Yao, Ioannis Papapanagiotou, and Robert D Callaway. Sla-aware resource scheduling for cloud storage. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 14–19. IEEE, 2014.

[274] R. Yazdani, M. Riera, J. Arnau, and A. González. The Dark Side of DNN Pruning. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*, Los Angeles, CA, June 2018.

[275] Jianwei Yin, Yan Tang, Shuiguang Deng, Bangpeng Zheng, and Albert Y Zomaya. Muse: a multi-tierd and sla-driven deduplication framework for cloud storage systems. *IEEE Transactions on Computers*, 70(5):759–774, 2020.

[276] Balgeun Yoo, Youjip Won, Seokhei Cho, Sooyong Kang, Jongmoo Choi, and Sungroh Yoon. Ssd characterization: From energy consumption's perspective. In *HotStorage*, 2011.

[277] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. Vssim: Virtual machine based ssd simulator. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2013.

[278] Jinsoo Yoo, Youjip Won, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. Analytical model of ssd parallelism. In *2014 4th International Conference On Simulation And Modeling Methodologies, Technologies And Applications (SIMULTECH)*, pages 551–559. IEEE, 2014.

[279] Erez Zadok and Ion Badulescu. A stackable file system interface for linux. In *LinuxExpo Conference Proceedings*, volume 94, page 10, 1999.

[280] Seyed Yahya Zahedi Fard, Mohamad Reza Ahmadi, and Sahar Adabi. A dynamic vm consolidation technique for qos and energy consumption in cloud environment. *The Journal of Supercomputing*, 73(10):4347–4368, 2017.

[281] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. The feniks fpga operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys'17)*, 2017.

[282] Jie Zhang, Myoungsoo Jung, Gyuyoung Park, David Donofrio, and John Shalf. Dram-less accelerator for energy efficient data processing. In *12TH ANNUAL NON-VOLATILE MEMORIES WORKSHOP*. IEEE, 2021.

[283] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. {FlashShare}: Punching through server storage stack from kernel to firmware for {Ultra-Low} latency {SSDs}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, 2018.

[284] Ning Zhang, Junichi Tatemura, Jignesh Patel, and Hakan Hacigumus. Re-evaluating designs for multi-tenant oltp workloads on ssd-basedi/o subsystems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*, 2014.

[285] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, Taipei, Taiwan, October 2016.

[286] Yu Zhang, Qingsong Wei, Cheng Chen, Mingdi Xue, Xinkun Yuan, and Chundong Wang. Dynamic scheduling with service curve for qos guarantee of large-scale cloud storage. *IEEE Transactions on Computers*, 67(4):457–468, 2017.

[287] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Boston, MA, March 2017.

[288] Ke Zhou, Yu Zhang, Ping Huang, Hua Wang, Yongguang Ji, Bin Cheng, and Ying Liu. Efficient ssd cache for cloud block storage via leveraging block reuse distances. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2496–2509, 2020.