THE UNIVERSITY OF CHICAGO

FAST AND STABLE DATA AND STORAGE SYSTEMS IN MILLI/MICRO-SECOND ERA

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
MINGZHE HAO

CHICAGO, ILLINOIS
DECEMBER 2020

Copyright © 2020 by Mingzhe Hao

All Rights Reserved

To my family.

TABLE OF CONTENTS

LIST OF FIGURES vii

LIST OF TABLES x

ACKNOWLEDGMENTS xi

ABSTRACT xii

1 INTRODUCTION 1
1.1 Analysis of Storage Performance Instability 2
1.2 Building Fast and Stable Storage Systems 4
1.2.1 Tiny-Tail Flash Devices 4
1.2.2 Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface . 5
1.2.3 Predictability on Unpredictable Flash Storage with a Light Neural Network 7
1.3 Thesis Organization 8

2 BACKGROUND AND MOTIVATIONS 10
2.1 Performance Instability in Storage Devices 10
2.1.1 Extended Motivation – Read Latency Instability in Production 12
2.2 Tail-Tolerance Mechanisms 13
2.2.1 Extended Motivation – No “TT” in NoSQL 15
2.3 Heuristic-based and ML-based Approaches 15
2.3.1 Extended Motivation – Rarity of Research in the “ML for OS” area 17

3 THE TAIL AT STORE: A REVELATION FROM MILLIONS OF HOURS OF DISK AND SSD DEPLOYMENTS 19
3.1 Methodology 21
3.1.1 RAID Architecture 21
3.1.2 About the Dataset 23
3.1.3 Metrics 24
3.2 Results 26
3.2.1 Slowdown Distributions and Characteristics 27
3.2.2 Workload Analysis 31
3.2.3 Other Correlations 34
3.2.4 Post-Slowdown Analysis 37
3.2.5 Summary 40
3.3 Tail-Tolerant RAID 41
3.3.1 Tail-Tolerant Strategies 42
3.3.2 Evaluation 43
3.4 Discussions 44
3.5 Related Work 46
3.6 Conclusion 47

4 MITTOS: SUPPORTING MILLISECOND TAIL TOLERANCE WITH FAST REJECTING SLO-AWARE OS INTERFACE . . . 48
4.1 MittOS Overview . . . 49
4.1.1 Deployment Model and Use Case . . . 49
4.1.2 Use Case . . . 50
4.1.3 Goals / Principles . . . 51
4.1.4 Design Challenges . . . 52
4.2 Case Studies . . . 53
4.2.1 Disk Noop Scheduler (MittNoop) . . . 53
4.2.2 Disk CFQ Scheduler (MittCFQ) . . . 54
4.2.3 SSD Management (MittSSD) . . . 55
4.2.4 OS Cache (MittCache) . . . 58
4.2.5 Implementation Complexity . . . 59
4.3 Applications . . . 59
4.4 Millisecond Dynamism . . . 61
4.5 Evaluation . . . 63
4.5.1 Microbenchmark Results . . . 64
4.5.2 MittCFQ Results with EC2 Noise . . . 65
4.5.3 Tail Amplified by Scale (MittCFQ) . . . 68
4.5.4 MittCache Results with EC2 Noise . . . 70
4.5.5 MittSSD Results with EC2 Noise . . . 70
4.5.6 Prediction Accuracy . . . 71
4.5.7 Tail Sensitivity to Prediction Error . . . 73
4.5.8 Other evaluations . . . 73
4.6 Discussions . . . 77
4.6.1 MittOS Limitations . . . 77
4.6.2 Beyond the Storage Stack . . . 78
4.6.3 Other Discussions . . . 79
4.7 Related Work . . . 80
4.8 Conclusion . . . 80
4.9 Appendix: Details . . . 81

5 LINNOS: PREDICTABILITY ON UNPREDICTABLE FLASH STORAGE WITH A LIGHT NEURAL NETWORK . . . 83
5.1 Overview . . . 85
5.1.1 Usage Scenario . . . 85
5.1.2 Overall Architecture . . . 86
5.1.3 Challenges . . . 88
5.2 LinnOS Design . . . 89
5.2.1 Training Data Collection . . . 90
5.2.2 Labeling (with Inflection Point) . . . 90
5.2.3 Light Neural Network Model . . . 93
5.2.4 Improving Accuracy . . . 96

5.2.5 Improving Inference Time . . . 98
5.2.6 Summary of Advantages . . . 99
5.2.7 Implementation Complexity . . . 100
5.3 Evaluation . . . 101
5.3.1 Setup . . . 101
5.3.2 Inflection Point (IP) Stability . . . 104
5.3.3 Latency Predictability . . . 105
5.3.4 (Low) Inaccuracy . . . 110
5.3.5 Trade-offs Balance . . . 111
5.3.6 Other Evaluations . . . 112
5.4 Conclusion and Discussions . . . 115

6 OTHER STORAGE WORK . . . 117
6.1 LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs . . . 117
6.2 The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator . . . 121
6.3 Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs . . . 122
6.4 Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems . . . 124

7 CONCLUSION AND FUTURE WORK . . . 126
7.1 Conclusion . . . 126
7.2 Future Work . . . 127

REFERENCES . . . 134

LIST OF FIGURES

2.1 **Latency unpredictability.** *The figures show CDFs of block-level read latencies, as discussed in Section 2.1.1. For the left figure, we ran one FIO workload on five different SSD models (the 5 CDF lines). For the right figure, we plot the latencies of 7 block-level traces obtained from 4 read-write servers (colored lines) and 2 read-only servers (bold gray lines) in Azure, Bing, and Cosmos clusters. The x-axis is anonymized for proprietary reasons. The traces are available from Microsoft with NDA.* 12

3.1 **Stable and slow drives in a RAID group.** 21
3.2 **RAID width and dataset duration.** 23
3.3 **Conceptual drive slowdown model.** 25
3.4 **Slowdown (S_i) and Tail (T^k) distributions (Section 3.2.1-Section 3.2.1).** *The figures show distributions of disk (top) and SSD (bottom) hourly slowdowns (S_i), including the three longest tails (T^{1-3}) as defined in Table 3.2. The y-axis range is different in each figure and the x-axis is in log_2 scale. We plot two gray vertical lines representing 1.5x and 2x slowdown thresholds. Important slowdown-percentile intersections are listed in Table 3.3.* 27
3.5 **Temporal behavior (Section 3.2.1).** *The figures show (a) the CDF of slowdown intervals (#hours until a slow drive becomes stable) and (b) the CDF of slowdown inter-arrival rates (#hours between two slowdown occurrences).* 29
3.6 **Slowdown extent (Section 3.2.1).** *Figure (a) shows the fraction of all drives that have experienced at least one occurrence of X-time slowdown ratio as plotted on the x-axis; the y-axis is in log_{10} scale. Figure (b) shows the fraction of slow drives that has exhibited at least X slowdown occurrences.* 31
3.7 **CDF of size and rate imbalance (Section 3.2.2).** *Figure (a) plots the rate imbalance distribution (RI_i) within the population of slow drive hours ($S_i \geq 2$). A rate imbalance of X implies that the slow drive serves X times more I/Os, as plotted in the x-axis. Reversely, Figure (b) plots the slowdown distribution (S_i) within the population of rate-imbalanced drive hours ($RI_i \geq 2$). Figures (c) and (d) correlate slowdown and size imbalance in the same way as Figures (a) and (b).* 32
3.8 **Drive age (Section 3.2.3).** *The figures plot the slowdown distribution across different (a) disk and (b) SSD ages. Each line represents a specific age by year. Each figure legend is sorted from the left-most to right-most lines.* 35
3.9 **SSD models (Section 3.2.3).** *The figure plots the slowdown distribution across different SSD models and vendors.* 36
3.10 **RAID I/O degradation (Section 3.2.4).** *The figures contrast the distributions of RIO degradation between stable-to-stable and stable-to-tail transitions.* 37
3.11 **Unplug/replug events (Section 3.2.4-Section 3.2.4).** *The figures show the relationships between slowdown occurrences and unplug/replug events. The top and bottom figures show the distribution of "wait-hour" and "recur-hour" respectively.* 39

3.12 **ToleRAID evaluation.** *The figures show the pros and cons of various ToleRAID strategies based on two slowdown distributions: (a) Rare and (b) Periodic. The figures plot the T^1 distribution (i.e., the RAID slowdown). T^1 is essentially based on the longest tail latency among the necessary blocks that each policy needs to wait for.* . . 43

4.1 **MITTOS Deployment Model (Section 4.1.1).** 50
4.2 **MITTOS use-case illustration (Section 4.1.2).** 52
4.3 **Millisecond-level latency dynamism in EC2.** *The figures are explained in Section 4.4. The 20 lines in Figure (a)-(f) represent the latencies observed in 20 EC2 nodes.* . 61
4.4 **Latency CDFs from microbenchmarks.** *The figures are explained in Section 4.5.1.* 64
4.5 **MITTCFQ results with EC2 noise.** *The figures are explained in Section 4.5.2.* . . . 67
4.6 **Tail amplified by scale (MITTCFQ vs. Hedged).** *The figures are explained in Section 4.5.3.* 68
4.7 **MITTCACHE vs. Hedged.** *The figures are explained in Section 4.5.4. The left figure shows the same CDF plot as in Figure 4.5a and the right figure the same %reduction bar plot as in Figure 4.6b.* 69
4.8 **MITTSSD vs. Hedged.** *The figures are explained in Section 4.5.5. The left figure shows the same plot as in Figure 4.5a and the right figure the same %reduction bar plot as in Figure 4.6b.* 70
4.9 **Prediction inaccuracy.** *(As explained in Section 4.5.6).* 71
4.10 **Tail sensitivity to prediction error.** *The figures are described in Section 4.5.7.* . . . 72
4.11 **MITTCFQ with macrobenchmarks and production workloads.** *The figures are discussed in Section 4.5.8.* 74
4.12 **C3 and bursty noises.** *The figure is described in Section 4.5.8.* 75
4.13 **MITTOS-powered Riak+LevelDB.** *The figure is explained in Section 4.5.8.* 77

5.1 **Usage scenario.** *This usage scenario is explained in Section 5.1.1. "LC" implies latency critical.* 86
5.2 **LinnOS architecture.** *The figure displays LinnOS architecture including LinnApp, as summarized in Section 5.1.2. The two SSD pictures represent the same SSD instance; the left one depicts tracing/training and the right one live inference on the SSD.* . . . 87
5.3 **Anticipating heterogeneity.** *The figure shows heterogeneous trained models, as mentioned in Section 5.1.3.* 89
5.4 **Inflection point (fast/slow threshold).** *The figures show the results of using higher, lower, and semi-optimum inflection point (IP) for fast/slow threshold as explained in Section 5.2.2. The figures format is latency CDF, as in Figure 2.1.* 91
5.5 **Light neural network.** *The figure depicts LinnOS 3-layer neural network explained in Section 5.2.3.* 94
5.6 **IP stability.** 104
5.7 **Average latencies.** *The figures show that LinnOS consistently outperforms all other methods, as explained in Section 5.3.3. The top and bottom graphs represent experiments on the consumer and enterprise arrays, respectively.* 106

5.8 **Percentile latencies.** *Explained in Section 5.3.3, the figures show that LinnOS+HL delivers the most predictable latencies (y-axis) across all percentiles (x-axis), even at p99.99. In Figure (a), "AZ/C" means Azure running on consumer array.* 107
5.9 **Low inaccuracy.** *The figure shows the percentage of false submits and false revokes. Note that only false submits really matter (see Section 5.3.4). Additionally,* **"P"** *represents other device models that we can access from a public cloud. For graph readability, here for "P" we only show the results for 1 device model, while the observations stand across the rest. In total, the accuracy evaluation covers 10 device models (1C+3E+6P).* 110
5.10 **LinnOS+H99.** 112
5.11 **On public traces.** *As explained in Section 5.3.6.* 113
5.12 **MongoDB on different filesystems.** *This figure shows that LinnOS can easily help data applications achieve more predictable latency (Section 5.3.6).* 114

6.1 **Goals.** 119

7.1 **Buffer learning and management.** 130

LIST OF TABLES

2.1 **Tail tolerance in NoSQL.** *(As explained in Section 2.2.1).* 15

2.2 **#Papers in ML/systems.** *The table shows the numbers of papers in the intersection of ML and systems in recent years. The full data of our literature study can be found in [14]. The numbers in the "Systems for ML" row exclude works that cover a larger scope that automatically covers optimization for ML workloads (e.g., improving stream processing for general workloads including but not specifically for ML ones). The "ML for HW configuration" works (3rd row) usually uses kernel/HW-level statistics and modify the HW configuration within a privilege setting, but not within the OS.* 18

3.1 **Dataset summary.** 22

3.2 **Primary metrics.** *The table presents the metrics used in our analysis. The distribution of N is shown in Figure 3.2a. L_i, S_i and T^k are explained in Section 3.1.3.* . . . 24

3.3 **Slowdown and percentile intersections.** *The table shows several detailed points in Figure 3.4. Table (a) details slowdown values at specific percentiles. Table (b) details percentile values at specific slowdown ratios.* 28

5.1 **I/O characteristics of re-rated traces (Section 5.3.1).** *The upper part (first four rows) is for the consumer-level flash array and the lower is for the enterprise-level one. Every max-IOPS value is measured within a 10-second window.* 103

5.2 **Inflection point (IP) settings.** *This table, as explained in Section 5.3.2, shows the IP values that our algorithm in Section 5.2.2 computed for every workload-device pair.* . 105

5.3 **Trade-offs balance.** *This table is explained Section 5.3.5. All the $+/-$ of accuracy and performance values are compared to our final neural network model described in Section 5.2.* 111

6.1 **Real storage functions, not offload ready.** *The table summarizes why real cloud drive services are either not completely offload ready or not easily composable with each other.* 118

6.2 **Summary of our findings and suggestions.** 125

7.1 **Write buffer sizes.** 129

ACKNOWLEDGMENTS

A journey of eight years, let alone additionally pursuing a Ph.D. degree, can substantially change a person. Knowing that a journey can be full of uncertainties, I am thankful for where it has led me and want to express my sincere gratitude to the important people I have met along this journey.

To my advisor Haryadi S. Gunawi. I have grown through a lot under your mentorship since my first AdvOS class in 2013 – one-year long research at Argonne and NetApp in 2014, my first submission and rejection in 2015, TailAtStore in FAST 2016, MittOS in SOSP 2017, formally mentoring a group of students in 2018, crystallization of my research experience in 2019, LinnOS in OSDI 2020, and eventually a Ph.D. degree. One needs to be lucky to have a great collaborator, an insightful mentor, and a care-taking boss, and I am super grateful to meet all these three in one.

To my committee members, Andrew A. Chien and Hank Hoffmann. I have learned various aspects from Andrew in my TailAtStore, MittOS, and early stage of LinnOS study and deepened my understanding on machine learning for systems through conversations with Hank.

To Shan Lu, who has been helping me in post-graduation career development.

To my friends – Boxue Wang, Cesar A. Stuardo, Chao Cui, Dixin Tang, Guangpu Li, Huaicheng Li, Jian Chen, Jeffrey F. Lukman, Lang Wu, Lintao Li, Nanqinqin Li, Ninglin Li, Riza O. Suminto, Ruiqiang Zhao, Siyu Chen, Shengshu Li, Tanakorn Leesatapornwongsa, Tong Li, Tianxiang Zhang, Xiaolin Huang, Xing Lin, Yi Ding, and Yueling Zhang. Though some of us are no longer in close contact as before, the memory that emerges in my recall from time to time remains a perpetual and precious treasure.

To those who bring my various types of experiences – pleasant or tough, sweet or bitter. It is these experiences that make me what I am today.

To my family. I dedicate this journey to you.

ABSTRACT

As numerous new data-intensive applications and storage hardware emerge, maintaining performance sustainability and robustness of data and storage systems is becoming more intricate and challenging. Users want numerous demands (*e.g.*, real-time latency, continuously high throughput, workload elasticity) to be met. Service providers are facing a hard task of delivering acceptable service-level objectives (SLOs) such as low and highly stable latencies. Both parties essentially wish for the same goal, but the gap in between continues to widen tragically and become more complex. Customers keep introducing more data paradigms (*e.g.*, big data, machine learning, IoT) and bombing providers with application-specific requirements that are more than non-trivial to fulfill, which brings a growing threat to designing generic systems that can persistently deliver rapid performance.

This dissertation *aims at building fast and stable next-generation data and storage systems. Specifically, we architect these systems generically to achieve rapid responses of low latency even in the most turmoil scenarios.* As systems grow in complexity, this dissertation tackles this significant problem from four different angles:

1. **Data approach:** We should have a thorough and large-scale understanding of real-world issues with increasing complicacy to help us pinpoint the potential crux and solutions. Here, we present TAILATSTORE, which mines performance logs tracking half a million disks and thousands of SSDs, and to the best of our knowledge is the most extensive study of storage device-level performance variability. TAILATSTORE reveals that storage performance instability is not uncommon, and the primary causes of slowdowns are the internal characteristics and idiosyncrasies of modern disk and SSD drives, motivating the design of tail-tolerant mechanisms.

2. **Hardware-level approach:** While other approaches attempt to reduce performance variability at the application level with approaches like speculation, we see a different point of view, whereas cutting performance variability "at the source" is more effective. Specifically, in TINY-TAILFLASH, we re-architect SSDs that collaborate with the host and circumvent almost all noises

induced by background operations. Furthermore, to further highlight the importance of hardware-level approach and facilitate its development, we present FAILSLOWATSCALE – a study on hardware with performance degradation, and FEMU – a software flash emulator for fostering future SSD research.

3. **OS-level approach:** At the heart of the system stack is the OS; hence, the question is how the OS should evolve today to provide stable performance for the deep stack. In tackling this problem, our insight is that the OS is not just the OS for personal computers, but rather the OS for the "datacenter". In this context, we present MITTOS – an OS that is SLO-aware and capable of predicting every I/O latency and failing over slow I/Os to peer OSs. MITTOS's no-wait approach helps reduce I/O completion time up to 35% compared to wait-then-speculate approaches.

Additionally, as another effort on "OS for datacenter", we present LeapIO, which promotes address transparency across components in the cloud storage stack to smooth the offload of complex storage services to today's I/O accelerators. LeapIO employs a set of OS/software techniques on top of hardware capabilities to provide a uniform address space across x86 cores and I/O accelerators, allowing the host to portably leverage the accelerators.

4. **ML-for-system approach:** Current systems are growing too complex for human designers to come up with a heuristic-based policy for optimal system control. So many different storage models exist, which are very heterogeneous with performance unpredictability. Applications cannot reason about how they work, and predicting systems' performance is a black art. This situation raises the question of whether machine learning can help. To answer this, we present LINNOS, which uses neural networks to predict the performance of every request and every I/O, making unforeseeable systems performance highly predictable. LINNOS supports black-box devices and real production traces without requiring any extra input from users, while outperforming industrial mechanisms and other approaches. Compared to hedging and heuristic-based methods, LINNOS improves the average I/O latencies by 9.6-79.6% with 87-97% inference accuracy and 4-6µs inference overhead for each I/O, demonstrating that it is possible to incorporate machine learning

inside operating systems for real-time decision-making.

Lastly, this dissertation raises discussions on future research to build fast and stable data and storage systems and help storage applications achieve performance predictability in milli/micro-second era.

CHAPTER 1

INTRODUCTION

Storage has grown enormously with a long journey from tapes to disks, SSDs, and persistent memory. The market projects to ship zettabytes of disk drives this year [5]. The solid-state drive (SSD) market is currently valued at $20–34 billion according to multiple sources [44, 51–53] and forecasted to reach $47–80.34 billion by 2025. Similarly, the non-volatile memory market is expected to reach $82 billion by 2022 [40, 41]. The storage market continues to expand with the explosion of big data, which is doubling every two years and reaching 175 zettabytes by 2025 [5].

Furthermore, we are not only addicted to data, but also more of it in real time. In a world of continuous collection and analysis of big data, storage performance is critical for many applications. Modern applications particularly demand low and predictable response times. In Google, an extra 500ms in search-result generation can drop traffic by 20% [189]. In Amazon, every 100ms of latency costs 1% in sales [188]. In trading systems, a broker could lose as much as $4 million per millisecond if its platform is 5ms behind the competition [228]. In the future, the pressure for low latency is even predicted to reach the sub-ms level as faster devices will be a commodity [72, 97, 174, 230]. As a result, today's SLOs include latency percentiles [69, 99, 212], *e.g.*, "<100ms at the 99th-percentile (or 'p99' for short)," implying that 99% of the requests must finish in less than 100ms.

However, achieving such strict SLOs is challenging due to the *"tail latency problem."* The primary nemesis is *resource contention*; for example, a supposedly fast operation can be slowed down by 10-100x if it is contending with other large operations or heavy background activities. The literature has shown how resource contention happens in *many* resource management layers and induces long tail latencies: for example, in *SSDs* due to garbage collection (GC) [130, 140, 161, 270], in *disks* due to disk seeks [134, 229, 272] or cleaning overhead [110]. Beyond that, we also see an increasing sign of hardware that is still running and functional but in a degraded mode, slower than its expected performance. For example, disk throughput can drop by three orders of

magnitude to 100 KB/s due to vibration; SSD operations can stall for seconds due to firmware bugs; memory cards can degrade to 25% of normal speed due to loose NVDIMM connection. All the issues above directly affect storage services and cause performance instability that leads to violations of SLOs, degrading user experience and impacting revenues negatively [77, 242].

Despite a growing number of studies that point out the performance instability in large-scale systems [64, 99, 100, 183, 244] and a vast amount of proposed solutions [90, 150, 177, 178, 280, 284], the instability issue still widely lingers [72], largely due to the growing internal complexity of the devices. Modern devices behave like an "operating system," managing all of its internal resources with background operations. While important, these are the kinds of operations that pose a threat to latency predictability [81, 129, 133, 140, 199, 206, 270, 280], which is still a fresh problem faced by many storage industries in recent years [8, 142, 203, 219].

This prevalent and notorious performance instability presents a significant challenge: *How should we build fast and stable next-generation data and storage systems? Specifically, how should these systems be architected to generically achieve rapid responses of low latency even in the most turmoil scenarios?* To answer this challenging question, we first take a data approach and analyze the performance variance in industrial data and storage systems to understand its landscape (Section 1.1), then develop multiple solutions at different levels, including hardware-level (Section 1.2.1), OS-level (Section 1.2.2), and ML-for-system (Section 1.2.3) approaches. We discuss these parts in the next two sections.

1.1 Analysis of Storage Performance Instability

It is important to have a thorough and scaled understanding of real-world issues with increasing complicacy to help us pinpoint the potential crux and solutions. Here, we focus on studying the prevalence and root causes of performance instability in storage devices.

A growing body of literature studies the general problem of performance instability in large-scale systems, specifically calling out the impact of stragglers on tail latencies [64, 99, 100, 183,

244, 253, 264, 269, 277]. Stragglers often arise from contention for shared local resources (*e.g.*, CPU, memory) and global resources (*e.g.*, network switches, back-end storage), background daemons, scheduling, power limits and energy management, and many others. These studies are mostly performed at server, network, or remote (cloud) storage levels.

To date, we find no systematic, *large-scale studies* of performance instability in *storage devices* such as disks and SSDs. Yet, mounting anecdotal evidence of disk and SSD performance instability in the field continue to appear in various forums. Such ad-hoc information is unable to answer quantitatively key questions about drive performance instability, questions such as: How much slowdown do drives exhibit? How widespread is it? What are the potential root causes?

To answer these questions, we have performed the largest empirical analysis of storage performance instability [130]. Collecting *hourly* performance logs from customer deployments of 458,482 disks and 4,069 SSDs spanning on average 87-day periods, we have amassed a dataset that covers 857 million hours of disk and 7 million hours of SSD field performance data. Uniquely, our data includes drive-RAID relationships, which allows us to compare the performance of each drive to that of peer drives in the same RAID group.

We find that storage performance instability is not uncommon: 0.2% of the time, a disk is more than 2x slower than its peer drives in the same RAID group (and 0.6% for SSD). Slowdown is widespread in the drive population; our study shows 26% of disks and 29% of SSDs have experienced at least one slowdown occurrence. Consequently, stable latencies at 99.9^{th} percentile are hard to achieve in today's storage drives. Slowdowns can also be extreme (*i.e.*, long tails); we observe several slowdown incidents as large as 2-4 orders of magnitude.

As a consequence, disk and SSD-based RAIDs experience at least one slow drive (*i.e.*, storage tail) 1.5% and 2.2% of the time. We observe that storage tails can adversely impact RAID performance, causing a throughput degradation, especially for full-stripe workloads that require to access all drives in the RAID.

To understand the root causes, we correlate slowdowns with other metrics (workload I/O rate

and size, drive event, age, and model). Surprisingly, we find that drive slowdowns should not be blamed on unbalanced workloads (*e.g.*, a drive is busier than others). Overall, the primary cause of slowdowns are the internal characteristics and idiosyncrasies of modern disk and SSD drives.

All of the observations above point out that storage systems are now faced with more responsibilities. Not only must they handle well-known faults such as latent sector errors and corruptions, now they must mask storage tail latencies as well. Therefore, there is an opportunity to create "tail tolerant" systems that can mask storage tail latencies online in deployment.

1.2 Building Fast and Stable Storage Systems

Our analysis highlights the severity of performance instability in our storage and the urgency of handling it. Considering the complicacy of modern storage stacks, we take multiple approaches from different angles. In this section, we first introduce how we cut performance variability "at the source" with hardware-level approach (Section 1.2.1), then describe how we advocate the principle of transparency and support millisecond tail tolerance for data-parallel applications using OS-level approach (Section 1.2.2), and eventually elaborate how this principle can be further extended to adapt to the exploding complexity of the systems using ML-for-system approach (Section 1.2.3).

1.2.1 Tiny-Tail Flash Devices

We start developing our solutions from hardware, where performance variance originates. Here we focus on flash storage devices, which has become the mainstream destination for storage users. From the users' side, they demand fast and stable latencies [101, 123]. However, SSDs do not always deliver the performance that users expect [39]. Some even suggest that flash storage "may not save the world" (due to the tail latency problem) [27].

The core problem of flash performance instability is the well-known and "notorious" *garbage collection* (GC) process. A GC operation causes long delays as the SSD cannot serve (blocks) incoming I/Os. Due to an ongoing GC, read latency variance can increase by 100× [27, 99]. In

the last decade, there is a large body of work that *reduces* the number of GC operations with a variety of novel techniques [128, 159, 161, 162, 178, 202, 268]. However, we find almost no work in literature that attempts to *eliminate* the *blocking* nature of GC operations and deliver steady SSD performance in long runs.

We address this urgent issue with "tiny-tail" flash drive (TINYTAILFLASH [270]), a GC-tolerant SSD that can deliver and guarantee stable performance. TINYTAILFLASH is a "tiny-tail" flash drive (SSD) that eliminates GC-induced tail latencies by circumventing GC-blocked I/Os with four novel strategies: plane-blocking GC, rotating GC, GC-tolerant read, and GC-tolerant flush. It is built on three SSD internal advancements: powerful controllers, parity-based RAIN, and capacitor-backed RAM, but is dependent on the use of intra-plane copyback operations. TINYTAILFLASH comes significantly close to a "no-GC" scenario. Specifically, between 99–99.99th percentiles, TINYTAILFLASH is only 1.0 to 2.6× slower than the no-GC case, while a base approach suffers from 5–138× GC-induced slowdowns. We also show that TINYTAILFLASH is more stable than state-of-the-art approaches that reduce GC impacts such as preemptive GC [32, 179]. In summary, by leveraging modern SSD internal technologies in a unique way, we have successfully built novel features that provide a robust solution to the critical problem of GC-induced tail latencies.

1.2.2 Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface

Our hardware-level solution is effective, but it requires specialized hardware and is ignorant about the upper-layer software. Meanwhile, in modern storage servers the stack is deep, not only just the hardware side, but also the many layers of software. In this context, to accommodate various hardware settings, many storage applications choose to cut tail latency in software, with "wait-then-speculate" as the most popular way. For example, some companies apply *"hedged requests"* [99], where a duplicate request is sent after the first request is outstanding for more than, say, the 95th-percentile expected latency. Such methods have proven to be highly effective coarse-grained jobs

(tens to hundreds of seconds) [100], where there is sufficient time to wait, observe, and launch extra speculative tasks if necessary. However, for real-time applications that generate large numbers of small requests, each expected to finish in milliseconds, techniques that "wait-then-speculate" are ineffective, as the time to detect a problem is comparable to the delay caused by it.

The techniques discussed above attempt to minimize tail in the *absence* of information about underlying resource busyness. While the OS layer may have such information, it is *hidden* and *unexposed*. A prime example is the `read()` interface that returns either success or error. Currently, the OS does not have a direct way to indicate that a request may take a long time, nor is there a way for applications to indicate they would like "to know the OS is busy."

We believe that this is an opportunity that can be exploited. As now we are in the datacenter-era, where data has replicas, applications do not need to wait for a server to finish data requests on its best, which will be slow under heavy resource contention. Instead, the OS on the server can quickly inform the application about a long service latency if the target resource is busy. Upon that, applications can choose *not* to wait, for example performing an *instant* failover to another replica or taking other corrective actions.

Here, we advocate a new philosophy: *the OS should be aware of application SLOs and quickly reject IOs with unmet SLOs* (due to resource busyness). To this end, we introduce MITTOS [129], an OS that employs a fast rejecting SLO-aware interface to support millisecond tail tolerance. In a nutshell, MITTOS provides an SLO-aware read interface, "`read(...,slo)`," such that applications can attach SLOs to their IO operations (*e.g.*, "`read()` should not take more than 20ms"). If the SLO cannot be satisfied (*e.g.*, long disk queue), MITTOS immediately rejects the IOs and returns `EBUSY` (*i.e.*, no wait), hence allowing the application to quickly failover (retry) to another node.

We implement MITTOS design in four different OS subsystems: the disk noop scheduler (MITTNOOP), CFQ scheduler (MITTCFQ), SSD management (MITTSSD), and OS cache management (MITTCACHE). Collectively, they cover the major components that can affect an IO request latency. We evaluate our MITTOS-powered MongoDB in a 20-node cluster with YCSB

workloads and the EC2 noise distribution. We compare MittOS with three other standard practices (basic timeout, cloning, and hedged requests). Compared to hedged requests (the most effective among the three), MittOS reduces the completion time of individual IO requests by 23-26% at p95 and 6-10% on average, showing how operating system support to cut millisecond-level tail latencies for data-parallel applications.

1.2.3 Predictability on Unpredictable Flash Storage with a Light Neural Network

Though MittOS has shown the effectiveness of performance transparency, its applicability suffers from the complexity of implementing prediction mechanisms for different resources. For example, MittCFQ, which conducts prediction for disk drives – the simplest type of devices in major storage media, takes 1810 LOC as we need to reverse-engineer the I/O scheduler. As modern storage devices grow more complex, this cost will only go higher and become unaffordable.

To make our solution both effective and generic, we turn to machine learning. The last decade has witnessed significant growth in the development and application of artificial intelligence (AI) and machine learning (ML). Many industries, including storage, are either applying or planning to use AI/ML techniques to address their respective problem domains. In the literature, we see an increase of research that leverages machine learning for solving system problems, such as management of networking [80, 106, 117, 148, 173, 187], CPU/GPU [84, 85], memory [195, 285], energy [107, 204, 205], code analysis/compilation [186, 208, 241], and many forms of distributed systems [75, 82, 95, 113, 116, 144, 198, 254]. We strongly believe there is an enormous opportunity to explore the use of machine learning exhaustively in the storage stack.

As an effort, we develop LinnOS [131], an operating system that has the capability of learning and inferring *per-I/O* speed for SSDs with high accuracy and minimal overhead using a lightweight neural network. By profiling the latency of millions of I/Os submitted to the device, converting the hard latency inference problem into a simple binary inference, and utilizing aggregate input

features including queue lengths and history latencies, LinnOS is as effective and fine-grained as MittOS to mitigate every slow I/O, and supports black-box devices and real production traces without requiring any extra input from users. Moreover, as a learning-based method, LinnOS can auto-tune its numerous parameters and does not require to implement complex heuristics for predictions. Our evaluation shows that LinnOS outperforms industrial approaches such as pure hedging and beats simple and "advanced" heuristics that we design. Compared to these methods, LinnOS, complemented by hedging based on the learning outcome, further improves the average I/O latencies by 9.6-79.6% with 87-97% accuracy and only 4-6µs inference overhead for every I/O. Overall, we show that it is plausible to adopt machine learning methods for operating systems to learn black-box storage devices, expanding the horizon in ML-for-storage research.

1.3 Thesis Organization

The rest of this dissertation is organized as follows:

- Background: Chapter 2 provides a background in anecdotes on performance instability in our storage devices and existing tail-tolerance mechanisms, including both heuristic-based and ML-based ones.
- Problems: Chapter 3 introduces a large-scale study on performance variance in industrial storage devices, revealing multiple key observations and insights.
- Major solutions: The next two chapters introduce our major solutions at OS and ML-for-system levels. Chapter 4 describes the design of MittOS – our OS-level solution, highlighting how performance transparency can help cut millisecond-level tail latencies for data-parallel applications. Chapter 5 presents LinnOS – our ML-for-system level solution, demonstrating a way to utilize machine learning to support achieving rapid responses at microsecond-level in black-box scenarios, further stretching the power of exploiting transparency.

- Other related work: Chapter 6 briefly mentions our other work on cloud and storage systems, including our hardware-level solution on achieving stable performance.
- Conclusion and Future Work: Chapter 7 concludes this dissertation and discusses potential future research directions on learning-based storage systems.

CHAPTER 2

BACKGROUND AND MOTIVATIONS

This chapter introduces the background and motivations for the major works in this dissertation. Specifically, we start by illustrating the performance instability in our storage (Section 2.1) and the necessity of conducting a large-scale quantitative study on its landscape. Next, we explain the conventional mechanisms to handle this instability and why they cannot catch up with the growing speed of modern storage devices (Section 2.2). Finally, we compare the philosophy behind existing solutions (heuristics vs. machine learning) and highlight the potential of applying "ML-for-OS" solutions in real-time scenarios (Section 2.3).

2.1 Performance Instability in Storage Devices

Understanding fault models is an important criteria of building robust systems. Decades of research has developed mature failure models such as fail-stop [59, 123, 194, 221, 235], fail-partial [70, 225, 234], fail-transient [155], faults as well as corruption [71, 104, 114, 236] and byzantine failures [91].

Besides these well-studied failure models, there is an under-studied "new" failure type: *fail-slow hardware*, hardware that is still running and functional but in a degraded mode, slower than its expected performance. Many major hardware components can exhibit fail-slow faults. For example, disk throughput can drop by three orders of magnitude to 100 KB/s due to vibration, SSD operations can stall for seconds due to firmware bugs. In the past several years, we have collected facts and anecdotal evidence of storage "limpware" [103, 122] from literature, online forums supported by various storage companies, and conversations with large-scale datacenter operators as well as product teams. We found many reports of storage performance problems due to various faults, complexities and idiosyncrasies of modern storage devices.

Disk: Magnetic disk drives can experience performance faults from various root causes such as

mechanical wearouts (*e.g.*, weak head [26]), sector re-reads due to media failures such as corruptions and sector errors [28], overheat from broken cooling fans [29], gunk spilling from actuator assembly and accumulating on disk head [36], firmware bugs [238], RAID controller defects [103, 247], and vibration from bad disk drive packaging, missing screws, earthquakes, and constant "noise" in data centers [109, 132]. All these problems can reduce disk bandwidth by 10-80% and increase latency by seconds. While the problems above can be considered as performance "faults", current generation of disks begin to induce performance instability "by default" (*e.g.*, with adaptive zoning and Shingled-Magnetic Recording technologies [55, 110, 175]).

SSD: The pressure to increase flash density translates to more internal SSD complexities that can induce performance instability. For example, SSD garbage collection, a well-known culprit, can increase latency by a factor of 100 [99]. Programming MLC cells to different states (*e.g.*, 0 vs. 3) may require different numbers of iterations due to different voltage thresholds [260]. The notion of "fast" and "slow" pages exists within an SSD; programming a slow page can be 5-8x slower compared to programming fast page [119]. As the device wears out, breakdown of gate oxide will allow charge moves across the gate easily, resulting in faster programming (10-50%), but also higher chance of corruption [118]. ECC correction, read disturb, and read retry are also factors of instability [111]. Finally, SSD firmware bugs can cause significant performance faults (*e.g.*, 300% bandwidth degradation in a Samsung firmware problem [252]).

While fail-slow hardware arguably did not surface frequently in the past, today, as systems are deployed at scale, along with many intricacies of large-scale operational conditions, the probability that a fail-slow hardware incident can occur increases. Furthermore, as hardware technology continues to scale (smaller and more complex), today's hardware development and manufacturing will only exacerbate the problem. A handful of prior papers already hinted at the urgency of this problem; many different terms have been used such as "fail-stutter" [68], "gray failure" [141], and "limp mode" [103, 122, 156]. However, we counted roughly only 8 stories per paper of fail-slow hardware mentioned in these prior papers, which is probably not sufficient enough to convince the

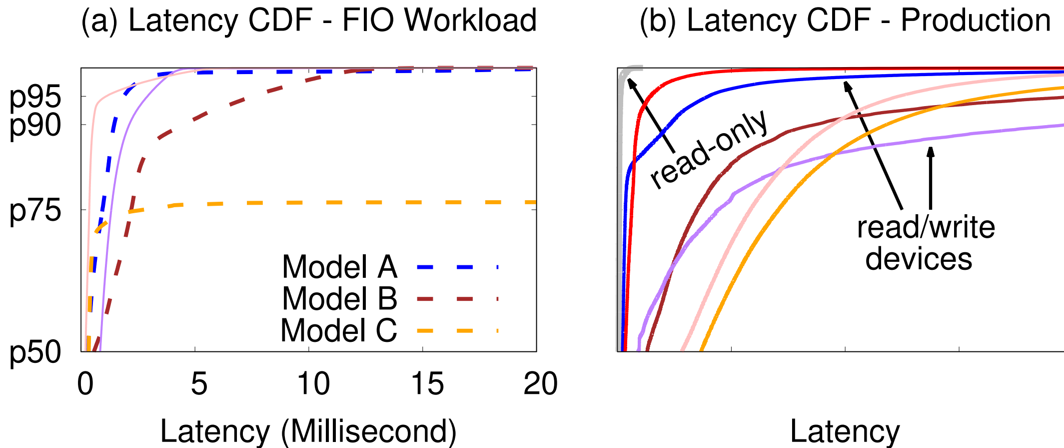

Figure 2.1: **Latency unpredictability.** *The figures show CDFs of block-level read latencies, as discussed in Section 2.1.1. For the left figure, we ran one FIO workload on five different SSD models (the 5 CDF lines). For the right figure, we plot the latencies of 7 block-level traces obtained from 4 read-write servers (colored lines) and 2 read-only servers (bold gray lines) in Azure, Bing, and Cosmos clusters. The x-axis is anonymized for proprietary reasons. The traces are available from Microsoft with NDA.*

systems community of this urgent problem.

Although the facts and anecdotes above are crucial, they do not provide empirical evidence that can guide the design of future storage systems. For this reason, we conduct TAILATSTORE – a systematic, large-scale study of performance instability in storage devices aiming at answer quantitatively key questions about drive performance instability, questions such as: How much slowdown do drives exhibit? How often does slowdown occur? How widespread is it? Does slowdown have temporal behavior? How long can slowdown persist? What are the potential root causes? What is the impact of tail latencies from slow drives to the RAID layer? Answers to these questions could inform a wealth of storage systems research and design.

2.1.1 *Extended Motivation – Read Latency Instability in Production*

Here we show an example of performance instability in production. The colored lines in Figure 2.1a show read latency instability in a read-write workload running on six different SSD models ranging from consumer SATA and NVMe SSDs to new data-center ones. Model A delivers fast and

stable latencies up to about “p98” (the 98th percentile), but models B and C exhibit larger latency tails starting at p90 and p75, respectively. However, when the write operations are converted into read I/Os, the performance becomes highly stable without much latency tail (not shown in the figure). Figure 2.1b also confirms this in real production scenarios in Microsoft SSD-backed servers. The colored lines show block-level read latencies of read-write servers (more variability), and the gray lines for read-only servers (more stability). All of these confirm how write-triggered garbage collection (GC), buffer flushing, and other internal operations are contending with user read I/Os. We only address read performance instability because we found write latencies to be (surprisingly) mostly stable as they are absorbed by the internal buffer, hence not affected by internal contentions such as garbage collection. Write latency spikes only happen when the buffer is full (rarely happened due to internal periodic flush).

The observed performance instability comes from the internal complexities that factor into latency behavior. For example, I/Os contend with each other if they fall into the same chip or channel, which depends on the hidden striping and partitioning logic; two user I/Os that go to separate channels might have different fates when one channel is occupied by GC data transfers between the chips in the channel. Our internal findings show that SSDs can have wide layouts (*e.g.*, 32 channels with four chips per channel) or deep layouts (*e.g.*, four channels with 16 chips per channel), where the latter will cause more channel contention. Some SSDs employ large write buffers from 256MB to as small as 12 MB and can periodically flush from every 3ms to as high as 1 second. As shown in Figure 2.1, all this contention happens from 1% to 25% of the time.

2.2 Tail-Tolerance Mechanisms

Early efforts to mitigate the “tail latency problem”, a typical sign of performance instability, focused on coarse-grained jobs (tens to hundreds of seconds) [100], where there is sufficient time to wait, observe, and launch extra speculative tasks if necessary. Such a “wait-then-speculate” method has proven to be highly effective; many variants of the technique have been proposed and

put into widespread use [62, 243, 277]. More challenging are applications that generate large numbers of small requests, each expected to finish in milliseconds. For these, "wait-then-speculate" is ineffective, as the time to detect a problem is comparable to the delay caused by it.

One approach to this challenging problem is *cloning*, where every request is cloned to multiple replicas and the first to respond is used [62, 264]; this proactive speculation however *doubles* the IO intensity. To reduce extra load, applications can delay the duplicate request and cancel the clone when a response is received (a *"tied requests"*) [99]; to achieve this, IO queueing and revocation management must be *built in* the application layer [76]. A more conservative option is *"hedged requests"* [99], where a duplicate request is sent after the first request is outstanding for more than, for example, the 95th-percentile expected latency; but the slow requests (5%) must *wait* before being retried. Finally, *"snitching"* [3, 244] – monitoring request latency and picking the fastest replica – can be employed; however, such techniques are *ineffective* if noise is bursty.

All of the techniques discussed above attempt to minimize tail in the *absence* of information about underlying resource busyness. While the OS layer may have such information, it is *hidden* and *unexposed*. A prime example is the `read()` interface that returns either success or error. However, when resources are busy (disk contention from other tenants, device garbage collection, etc.), a `read()` can be stalled inside the OS for some time. Currently, the OS does not have a direct way to indicate that a request may take a long time, nor is there a way for applications to indicate they would like "to know the OS is busy."

To solve this problem, we propose MITTOS , which advocates a new philosophy: *the OS should support performance transparency, for example, be aware of application SLOs and quickly reject IOs with unmet SLOs, exposing resource busyness*. The OS arguably knows "everything" about its resources, including which resources suffer from contention. If the OS can quickly inform the application about a long service latency, applications can better manage impacts on tail latencies. If advantageous, they can choose *not* to wait, for example performing an *instant* failover to another replica or taking other corrective actions.

	Def. TT	TO Val.	Failover	Clone	Hedged/Tied
Cassandra	×	12s	✓	×	×
Couchbase	×	75s	×	×	×
HBase	×	60s	✓	✓	×
MongoDB	×	30s	×	×	×
Riak	×	10s	×	×	×
Voldemort	×	5s	✓	✓	×

Table 2.1: **Tail tolerance in NoSQL.** *(As explained in Section 2.2.1).*

2.2.1 Extended Motivation – No "TT" in NoSQL

The goal of this subsection is to highlight that not all NoSQL systems have sufficient tail-tolerance mechanisms ("no 'TT' in NOSQL"). We analyzed six popular NoSQL systems (listed in Table 2.1), each ran on 4 nodes (1 client and 3 replicas), generated thousands of 1KB reads with YCSB [94], and emulated a severe IO contention for one second in a rotating manner across the three replica nodes (to emulate IO burstiness), and finally analyzed if there is any timeout/failover.

Table 2.1 summarizes our findings. First, the "Def. TT" column suggests that all of them (in their default configurations) does not failover from the busy replica to the less-busy ones; Cassandra employs snitching but is not effective with 1sec rotating burstiness. Second, the "TO Val." column provides the reason; by default, the timeout values are very coarse-grained (tens of seconds), thus an IO can stall for a long time without being retried. Third, to exercise the timeout, we set it to 100ms and surprisingly we observed that three of them do *not* failover on a timeout (the "Failover" column); undesirably, the users receive read errors even though less-busy replicas are available. Finally, we analyzed if more advanced techniques are supported and found that only two employ cloning and none of them employ hedged/tied requests (the last two columns).

2.3 Heuristic-based and ML-based Approaches

Heuristic-based approaches. Besides MittOS, a vast amount of research has been devoted to mask the performance instability using "White-box" approaches that re-architect device internals

[90, 150, 153, 158, 190, 245, 262, 270]. "White-box" approaches are powerful but face a high barrier to adoption unless SSD vendors implement the recommendations.

In the middle ground, "gray-box" methods suggest partial device-level modification combined with OS or application-level changes working together in taming the latency instability [169–171, 237, 280, 284]. However, they also depend on the vendors' willingness to modify the device interface.

Finally, more adoptable "black-box" techniques attempt to mask the instability without modifying the underlying hardware and its level of abstraction. Some of them optimize the file systems or storage applications specifically for SSD usage [92, 164, 172, 177, 178, 209, 240, 263, 266], while some others simply use speculative execution [2, 15] but pay the cost of extra I/Os due to being oblivious to storage behaviors. There are existing works on probing black-box SSDs and using the findings for optimization purposes, for example, deconstructing the write buffer size and flush policy [166], detecting the chunk size, interleaving degrees and mapping policies [83], and detecting the size of read/write/erase unit and type of NAND memory used [160]. These works indeed show that deconstructing black-box elements from SSDs is valuable. The downside is that each solution typically probes one or a few elements of the SSD internals and the process is a manual process.

Learning-based approaches. We take a new approach: let the device be the device (black-box) and do not redesign the file systems or applications, but *learn* the device behavior (*i.e.*, not be storage oblivious). The key to our approach is learning. Can we learn the behavior of the underlying device in a black-box way and use the results of the learning to increase predictability? This is a domain that machine learning can likely help. Here, we introduce LINNOS, an operating system that has the capability of learning and inferring *per-I/O* speed with high accuracy and minimal overhead using a lightweight neural network.

We believe ML solutions will fit this problem well for two reasons. First, SSD internals are complex; I/O latencies can be affected by many factors such as chip/channel-level contention,

garbage collection, wear leaving, scheduling policies (*e.g.*, prioritized read, GC preemption), and retries due to wearouts [99, 126, 140]. It is hard to guess latency based only on simple metrics. For example, one might assume that a long IO queue length might imply longer latencies, but for SSDs, due to all the factors above, such a simple correlation cannot be made (we found a low correlation between queue length and I/O latency).

Second, most SSDs are black-box devices where applications and OSs do not have visibility to the SSD internals. However, the OS perhaps can learn about them given the history of all or recent I/Os. Moreover, there are arguably hundreds of different SSD models from mobile, PC, to datacenter versions, hence a general learning solution is more appropriate to heuristic-based ones.

There are some other works that use ML for estimating I/O performance such as average response time based on workload characteristics [139, 185, 257]. A major limitation of these works is that they only predict at "aggregate" level (*e.g.*, is this machine busy within in the next five minutes?). However, real-world deployments exhibit unpredictable, bursty patterns where contentions come and go in millisecond intervals [129, Section 6]. In this case, predicting at coarse average level does not suffice (as tail latencies are about the per-I/O latencies).

2.3.1 Extended Motivation – Rarity of Research in the "ML for OS" area

Systems and machine learning have remarkably facilitated each other in recent years. On one hand, there is the "systems for ML" research branch where systems design continues to evolve to improve ML usage, such as works that build ML frameworks [54, 207], optimize compilers for ML [86, 149], and use accelerators [87, 115]. On the other hand, there is "ML for systems" where machine learning has been widely adopted in systems research, such as works that use ML techniques to improve resource management [95], scheduling policies [191], configuration tuning [173], and request optimization [73, 135] in cloud, edge, and mobile systems. Table 2.2 summarizes the result of our literature study within the intersection of ML and systems [14]. The first and second rows highlight the many publications in these two topics in major conferences and

Topics	Major sys conf.	ML/sys workshops
Systems for ML	59	163
ML for systems	35	20
ML for HW conf.	6	4
ML for OS	4	1

Table 2.2: **#Papers in ML/systems.** *The table shows the numbers of papers in the intersection of ML and systems in recent years. The full data of our literature study can be found in [14]. The numbers in the "Systems for ML" row exclude works that cover a larger scope that automatically covers optimization for ML workloads (e.g., improving stream processing for general workloads including but not specifically for ML ones). The "ML for HW configuration" works (3rd row) usually uses kernel/HW-level statistics and modify the HW configuration within a privilege setting, but not within the OS.*

workshops in recent years.

During this study, we also asked *"are there works that deploy ML techniques for/within the OS layer?"*, *e.g.*, for low-level process or IO scheduling. Surprisingly, we only found a small number of works in this area [93, 105, 106, 112, 233] as shown in the last row of the table. Note that by "ML for OS", our criteria is that the learning outcomes are used by the OS layer. Works that use kernel statistics for learning but not use the outcomes for OS policies are not considered in this category. While it is possible that we missed some ML-for-OS papers or miscategorized other papers, the sheer contrast between the numbers in the table will likely stand.

Our finding echoes what premier ML/system researchers have conveyed to the community. Both Jeffrey Dean and Michael I. Jordan mentioned in their talks that traditional low-level systems such as the operating system "does not make extensive use of ML today", but "learning should be used throughout our computing systems" [98, 151]. The possibility is there. OS components like process scheduler, memory management and file systems, are mostly filled with heuristics that work well in general cases but might not adapt to actual patterns of usage as they might not take more available context into consideration. In this context, we present LinnOS [131] as an attempt to further explore the potential benefit of ML-for-OS.

CHAPTER 3

THE TAIL AT STORE: A REVELATION FROM MILLIONS OF HOURS OF DISK AND SSD DEPLOYMENTS

In TAILATSTORE, we perform the largest empirical analysis of storage performance instability. Collecting *hourly* performance logs from customer deployments of 458,482 disks and 4,069 SSDs spanning on average 87-day periods, we amass a dataset that covers 857 million hours of disk and 7 million hours of SSD field performance data.

Uniquely, our data includes drive-RAID relationships, which allows us to compare the performance of each drive (D_i) to that of peer drives in the same RAID group $(i = 1..N)$. The RAID and file system architecture in our study (Section 3.1.1) expects that the performance of every drive (specifically, hourly average latency L_i) is similar to peer drives in the same RAID group.

Our primary metric, *drive slowdown ratio* (S_i), the fraction of a drive's latency (L_i) over the median latency of the RAID group $(median(L_{1..N}))$, captures deviation from the assumption of homogeneous drive performance. Assuming that most workloads are balanced across all the data drives, a normal drive should not be much slower than the other drives. Therefore, we define "**slow**" (unstable) drive hour when $S_i \geq 2$ (and "stable" the otherwise). Throughout the paper, we use 2x and occasionally 1.5x slowdown threshold to classify drives as slow.

In the following segment, we briefly summarize the findings from our large-scale analysis.

(i) Slowdown occurrences (Section 3.2.1): Disks and SSDs are slow $(S_i \geq 2)$ for 0.22% and 0.58% of *drive hours* in our study. With a tighter $S_i \geq 1.5$ threshold, disks and SSDs are slow for 0.69% and 1.27% of disk hours respectively. Consequently, stable latencies at 99.9th percentile are hard to achieve in today's storage drives. Slowdowns can also be extreme (*i.e.*, long tails); we observe several slowdown incidents as large as 2-4 orders of magnitude.

(ii) Tail hours and RAID degradation (Section 3.2.1): A slow drive can often make an entire RAID perform poorly. The observed instability causes RAIDs to suffer 1.5% and 2.2% of *RAID*

hours with at least one slow drive (*i.e.*, *"tail hours"*). Using 1.5x slowdown threshold, the numbers are 4.6% and 4.8%. As a consequent, stable latencies at 99th percentile (or 96th with 1.5x threshold) are impossible to guarantee in current RAID deployments. Workload performance (especially full-stripe balanced workload) will suffer as a consequence of RAID tails. In our dataset, we observe that RAID throughput can degrade during stable to tail hours (Section 3.2.4).

(iii) Slowdown temporal behavior and extent (Section 3.2.1, Section 3.2.1): We find that slowdown often persists; 40% and 35% of slow disks and SSDs respectively remain unstable for more than one hour. Slowdown periods exhibit temporal locality; 90% of disk and 85% of SSD slowdowns occur on the same day of the previous occurrence. Finally, slowdown is widespread in the drive population; our study shows 26% of disks and 29% of SSDs have experienced at least one slowdown occurrence.

(iv) Workload analysis (Section 3.2.2): Drive slowdowns are often blamed on unbalanced workloads (*e.g.*, a drive is busier than others). Our findings refute this, showing that more than 95% of slowdown periods *cannot* be attributed to I/O size or rate imbalance.

(v) "The fault is (likely) in our drives": We find that older disks exhibit more slowdowns (Section 3.2.3) and MLC flash drives exhibit more slowdowns than SLC drives (Section 3.2.3). Overall, evidence suggests that most slowdowns are caused by internal characteristics of modern disk and SSD drives.

In summary, drive performance instability means the homogeneous performance assumption of traditional RAID is no longer accurate. Drive slowdowns can appear at different times, persist, disappear, and recur again. Their occurrence is "silent"—not accompanied by observable drive events (Section 3.2.3). Most importantly, workload imbalance is not a major root cause (Section 3.2.2). Replacing slow drives is not a popular solution (Section 3.2.4-Section 3.2.4), mainly because slowdowns are often transient and drive replacement is expensive in terms of hardware and RAID rebuild costs.

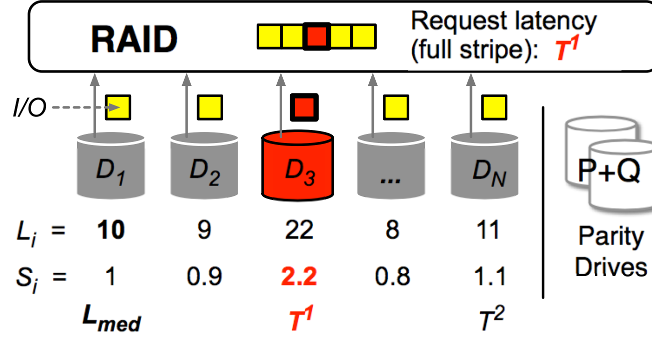

Figure 3.1: **Stable and slow drives in a RAID group.**

(vi) The need for tail-tolerant RAID: All of the reasons above point out that file and RAID systems are now faced with more responsibilities. Not only must they handle well-known faults such as latent sector errors and corruptions, now they must mask storage tail latencies as well. Therefore, there is an opportunity to create "tail tolerant" RAID that can mask storage tail latencies online in deployment.

In the following sections, we present our methodology (Section 3.1), the main results (Section 3.2), an opportunity assessment of tail-tolerant RAID (Section 3.3), discussion (Section 3.4), related work (Section 3.5) and conclusion (Section 3.6).

3.1 Methodology

In this section, we describe the RAID systems in our study (Section 3.1.1), the dataset (Section 3.1.2), and the metrics we use to investigate performance instability (Section 3.1.3). The overall methodology is illustrated in Figure 3.1.

3.1.1 RAID Architecture

RAID group: Figure 3.1 provides a simple illustration of a RAID group. We study disk- and SSD-based RAID groups. In each group, disk or SSD devices are directly attached to a proprietary RAID controller. All the disk or SSD devices within a RAID group are homogeneous (same model,

	Disk	**SSD**
RAID groups	38,029	572
Data drives per group	3-26	3-22
Data drives	458,482	4,069
Duration (days)	1-1470	1-94
Drive hours	857,183,442	7,481,055
Slow drive hours (Section 3.2.1)	1,885,804	43,016
Slow drive hours (%)	0.22	0.58
RAID hours	72,046,373	1,072,690
Tail hours (Section 3.2.1)	1,109,514	23,964
Tail hours (%)	1.54	2.23

Table 3.1: **Dataset summary.**

size, speed, etc.); deployment age can vary but most of them are the same.

RAID and file system design: The RAID layer splits each RAID request to per-drive I/Os. The size of a *per-drive I/O* (a square block in Figure 3.1) can vary from 4 to 256 KB; the storage stack breaks large I/Os to smaller I/Os with a maximum size of the processor cache size. Above the RAID layer runs a proprietary file system (not shown) that is highly tuned in a way that makes most of the RAID I/O requests cover the full stripe; most of the time the drives observe balanced workload.

RAID configuration: The RAID systems in our study use small chunk sizes (*e.g.*, 4 KB). More than 95% of the RAID groups use a custom version of RAID-6 where the parity blocks are not rotated; the parity blocks live in two separate drives (P and Q drives as shown in Figure 3.1). The other 4% use RAID-0 and 1% use RAID-4. We only select RAID groups that have at least three data drives ($D_1..D_N$ where $N \geq 3$ in Figure 3.1), mainly to allow us measure the relative slowdown compared to the median latency. Our dataset contains RAID groups with 3-26 data drives per group. Figure 3.2a shows the RAID width distribution (only data drives); wide RAID (*e.g.*, more than 8 data drives) is popular.

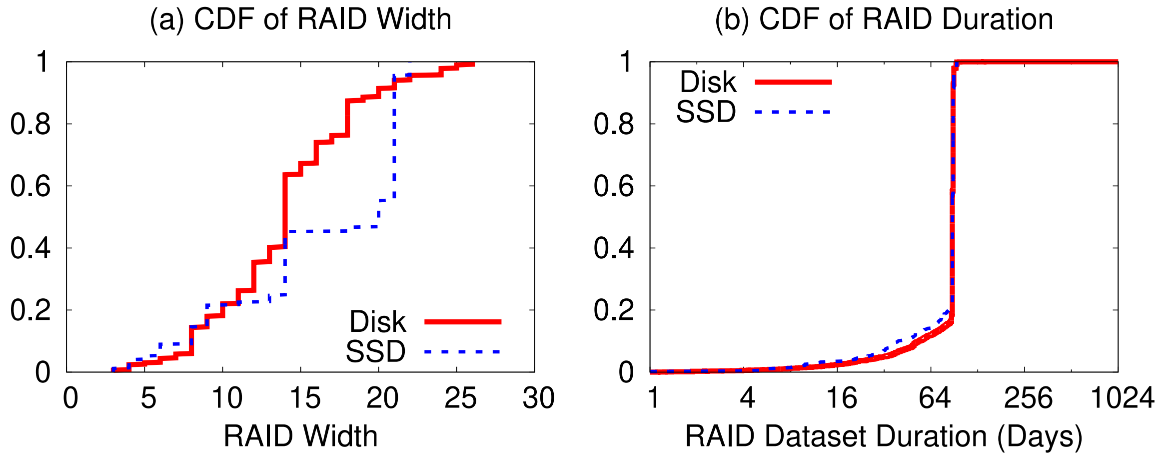

Figure 3.2: **RAID width and dataset duration.**

3.1.2 About the Dataset

Scale of dataset: A summary of our dataset is shown in Table 3.1. Our dataset contains 38,029 disk and 572 SSD groups within deployment duration of 87 days on average (Figure 3.2b). This gives us 72 and 1 million disk and SSD RAID hours to analyze respectively. When broken down to individual drives, our dataset contains 458,482 disks and 4069 SSDs. In total, we analyze 857 million and 7 million disk and SSD drive hours respectively.

Data collection: The performance and event logs we analyze come from production systems at customer sites. When the deployed RAID systems "call home", an auto-support system collects *hourly performance metrics* such as: average I/O latency, average latency per block, and number of I/Os and blocks received every hour. All these metrics are collected at the RAID layer. For each of these metrics, the system separates read and write metrics. In addition to performance information, the system also records drive events such as response timeout, drive not spinning, unplug/replug events.

Label	Definition
	Measured metrics:
N	Number of *data* drives in a RAID group
D_i	Drive number within a RAID group; $i = 1..N$
L_i	Hourly average I/O latency observed at D_i
	Derived metrics:
L_{med}	Median latency; $L_{med} = Median\ of(L_{1..N})$
S_i	Latency slowdown of D_i compared to the median; $S_i = L_i/L_{med}$
T^k	The k-th largest slowdown ("k-th longest tail"); $T^1 = Max\ of\ (S_{1..N})$, $T^2 = 2nd\ Max\ of\ (S_{1..N})$, and so on
Stable	A stable drive hour is when $S_i < 2$
Slow	A slow drive hour is when $S_i \geq 2$
Tail	A tail hour implies a RAID hour with $T_i \geq 2$

Table 3.2: **Primary metrics.** *The table presents the metrics used in our analysis. The distribution of N is shown in Figure 3.2a. L_i, S_i and T^k are explained in Section 3.1.3.*

3.1.3 Metrics

Below, we first describe the metrics that are measured by the RAID systems and recorded in the auto-support system. Then, we present the metrics that we derived for measuring tail latencies (slowdowns). Some of the important metrics are summarized in Table 3.2.

Measured Metrics

Data drives (N): This symbol represents the number of data drives in a RAID group. Our study only includes data drives mainly because read operations only involve data drives in our RAID-6 with non-rotating parity. Parity drives can be studied as well, but we leave that for future work. In terms of write operations, the RAID small-write problem is negligible due to the file system optimizations (Section 3.1.1).

Per-drive hourly average I/O latency (L_i): Of all the metrics available from the auto-support system, we at the end only use the hourly average I/O latency (L_i) observed by every data drive (D_i) in every RAID group (i=1..N), as illustrated in Figure 3.1. We initially analyzed "throughput" metrics as well, but because the support system does not record per-IO throughput average,

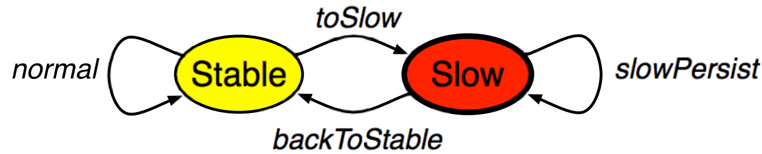

Figure 3.3: **Conceptual drive slowdown model.**

we cannot make an accurate throughput analysis based on hourly average I/O sizes and latencies.

Other metrics: We also use other metrics such as per-drive hourly average I/O rate (R_i) and size (Z_i), time of day, drive age, model, and events (replacements, unplug/replug, etc.), which we correlate with slowdown metrics to analyze root causes and impacts.

Derived Metrics

Slowdown (S_i): To measure tail latencies, RAID is a perfect target because it allows us to measure the relative slowdown of a drive compared to the other drives in the same group. Therefore, as illustrated in Figure 3.1, for every hour, we first measure the median group latency L_{med} from $L_{1..N}$ and then measure the hourly *slowdown* of a drive (S_i) by comparing its latency with the median latency (L_i/L_{med}). The total number of S_i is essentially the "#drive hours" in Table 3.1. Our measurement of S_i is reasonably accurate because most of the workload is balanced across the data drives and the average latencies (L_i) are based on per-drive I/Os whose size variance is small (see Section 3.1.1).

Stable vs. slow drive hours: Assuming that most workload is balanced across all the data drives, a "stable" drive should not be much slower than other drives. Thus, we use a slowdown threshold of **2x** to differentiate slow drive hours ($S_i \geq 2$) and stable hours ($S_i < 2$). We believe 2x slowdown threshold is tolerant enough, but conversations with several practitioners suggest that a conservative 1.5x threshold will also be interesting. Thus, in some of our findings, we show additional results using 1.5x slowdown threshold.

Conceptually, drives appear to behave similar to a simple Markov model in Figure 3.3. In

a given hour, a drive can be stable or slow. In the next hour, the drive can stay in the same or transition to the other condition.

Tails (T^k): For every hourly $S_{1..N}$, we derive the k-th largest slowdown represented as T^k. In this study, we only record the three largest slowdowns (T^1, T^2 and T^3). $\underline{T^1}$ represents the "longest tail" in a given RAID hour, as illustrated in Figure 3.1. The total number of T^1 is the "#RAID hours" in Table 3.1. The differences among T^k values will provide hints to the potential benefits of tail-tolerant RAID.

Tail hours: A "tail hour" implies a RAID hour that observes $T^1 \geq 2$ (*i.e.*, the RAID group observes at least one slow drive in that hour). This metric is important for full-stripe balanced workload where the performance will follow the longest tail (*i.e.*, the entire RAID slows down at the rate of T^1).

From the above metrics, we can further measure other metrics such as slowdown intervals, extents, and repetitions. Overall, we have performed an in-depth analysis of all the measured and derived metrics. In many cases, due to space constraints, we aggregate some results whenever the sub-analysis does not show different behaviors. For example, we merge read and write slowdowns as I/O slowdown. In some graphs, we break down the slowdowns (*e.g.*, to 2-4x, 4-8x, 8-16x) if their characterizations are different.

3.2 Results

We now present the results of our study in four sets of analysis: slowdown and tail distributions and characteristics (Section 3.2.1), correlations between slowdowns and workload-related metrics (Section 3.2.2) and other available metrics (Section 3.2.3), and post-slowdown analysis (Section 3.2.4).

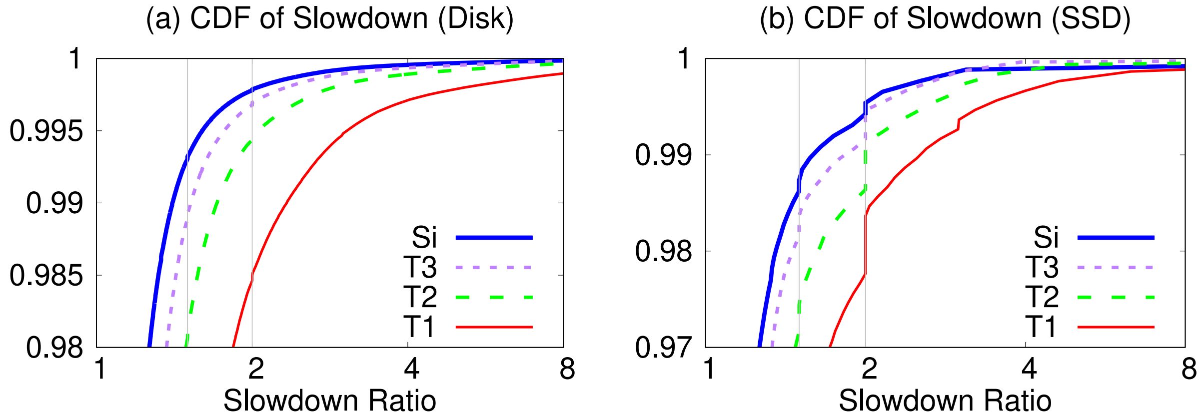

Figure 3.4: **Slowdown (S_i) and Tail (T^k) distributions (Section 3.2.1-Section 3.2.1).** *The figures show distributions of disk (top) and SSD (bottom) hourly slowdowns (S_i), including the three longest tails (T^{1-3}) as defined in Table 3.2. The y-axis range is different in each figure and the x-axis is in log_2 scale. We plot two gray vertical lines representing 1.5x and 2x slowdown thresholds. Important slowdown-percentile intersections are listed in Table 3.3.*

3.2.1 Slowdown Distributions and Characteristics

In this section, we present slowdown and tail distributions and their basic characteristics such as temporal behaviors and the extent of the problem.

Slowdown (S_i) Distribution

We first take all S_i values and plot their distribution as shown by the thick (blue) line in Figure 3.4 (steeper lines imply more stability). Table 3.3 details some of the slowdown and percentile intersections.

Finding #1: *Storage performance instability is not uncommon.* Figure 3.4 and Table 3.3b show that there exists 0.22% and 0.58% of drive hours (99.8^{th} and 99.4^{th} percentiles) where some disks and SSDs exhibit at least 2x slowdown ($S_i \geq 2$). With a more conservative 1.5x slowdown threshold, the percentiles are 99.3^{th} and 98.7^{th} for disk and SSD respectively. These observations imply that user demands of stable latencies at 99.9^{th} percentile [101, 244, 269] (or 99th with 1.5x threshold) are not met by current storage devices.

Disk and SSD slowdowns can be high in few cases. Table 3.3a shows that at four and five

(a)

Y:	90th	95	99	99.9	99.99	99.999
Slowdown (S_i) at Y^{th} percentile						
Disk	1.1x	1.2	1.4	2.7	9	30
SSD	1.1x	1.2	1.7	3.1	10	39
Greatest slowdown (T^1) at Y^{th} percentile						
Disk	1.3x	1.5	2.4	9	29	229
SSD	1.3x	1.5	2.5	20	37	65

(b)

X:	1.2x	1.5x	2x	4x
Percentile at S_i=X				
Disk	97.0^{th}	99.3	99.78	99.96
SSD	95.9^{th}	98.7	99.42	99.92
Percentile at T^1=X				
Disk	83.3^{th}	95.4	98.50	99.72
SSD	87.0^{th}	95.2	97.77	99.67

Table 3.3: **Slowdown and percentile intersections.** *The table shows several detailed points in Figure 3.4. Table (a) details slowdown values at specific percentiles. Table (b) details percentile values at specific slowdown ratios.*

nines, slowdowns reach ≥9x and ≥30x respectively. In some of the worst cases, 3- and 4-digit disk slowdowns occurred in 2461 and 124 hours respectively, and 3-digit SSD slowdowns in 10 hours.

Tail (T^k) Distribution

We next plot the distributions of the three longest tails (T^{1-3}) in Figure 3.4. Table 3.3 details several T^1 values at specific percentiles.

Finding #2: *Storage tails appear at a significant rate.* The T^1 line in Figure 3.4 shows that there are 1.54% and 2.23% "tail hours" (*i.e.*, RAID hours with at least one slow drive). With a conservative 1.5x threshold, the percentiles are 95.4^{th} and 95.2^{th} for disk and SSD respectively. These numbers are alarming for full-stripe workload because the whole RAID will appear to be slow if one drive is slow. For such workload, stable latencies at 99th percentile (or 96th with 1.5x threshold) cannot be guaranteed by current RAID deployments.

The differences between the three longest tails shed light on possible performance improvement from tail-tolerant RAID. If we reconstruct the late data from the slowest drive by reading from a

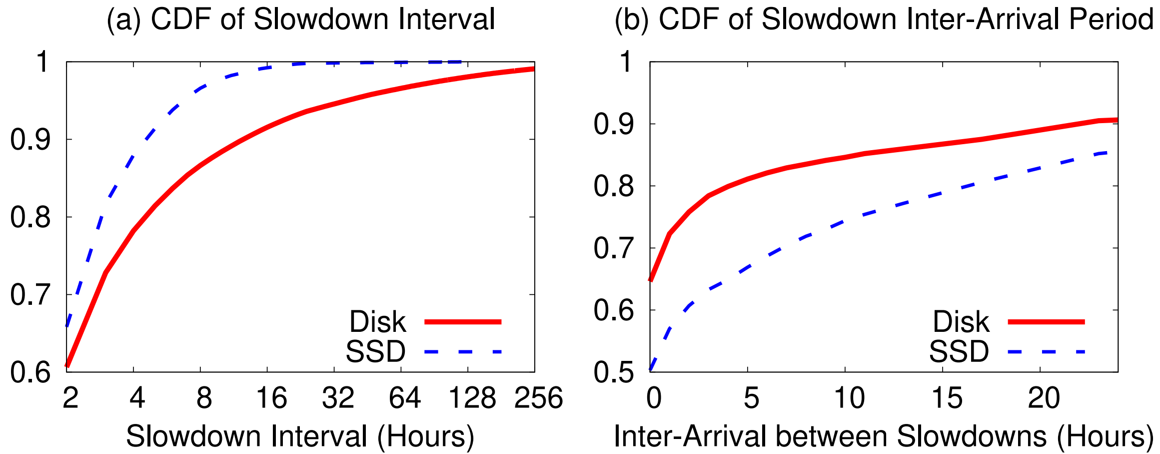

Figure 3.5: **Temporal behavior (Section 3.2.1).** *The figures show (a) the CDF of slowdown intervals (#hours until a slow drive becomes stable) and (b) the CDF of slowdown inter-arrival rates (#hours between two slowdown occurrences).*

parity drive, we can cut the longest tail. This is under an assumption that drive slowdowns are independent and thus reading from the parity drive can be faster. If two parity blocks are available (*e.g.*, in RAID-6), then tail-tolerant RAID can read two parity blocks to cut the last two tails.

Finding #3: *Tail-tolerant RAID has a significant potential to increase performance stability.* The T^1 and T^2 values at x=2 in Figure 3.4a suggests the opportunity to reduce disk tail hours from 1.5% to 0.6% if the longest tail can be cut, and furthermore to 0.3% (T^3) if the two longest tails can be cut. Similarly, Figure 3.4b shows that SSD tail hours can be reduced from 2.2% to 1.4%, and furthermore to 0.8% with tail-tolerant RAID.

The T^1 line in Figure 3.4b shows several vertical steps (*e.g.*, about 0.6% of T^1 values are exactly 2.0). To understand this, we analyze S_i values that are exactly 1.5x, 2.0x, and 3.0x. We find that they account for 0.4% of the entire SSD hours and their corresponding hourly and median latencies (L_i and L_{med}) are exactly multiples of 250 μs. We are currently investigating this further with the product groups to understand why some of the deployed SSDs behave that way.

Temporal Behavior

To study slowdown temporal behaviors, we first measure the *slowdown interval* (how many consecutive hours a slowdown persists). Figure 3.5a plots the distribution of slowdown intervals.

Finding #4: *Slowdown can persist over several hours.* Figure 3.5a shows that 40% of slow disks do not go back to stable within the next hour (and 35% for SSD). Furthermore, slowdown can also persist for a long time. For example, 13% and 3% of slow disks and SSDs stay slow for 8 hours or more respectively.

Next, we measure the *inter-arrival period* of slowdown occurrences from the perspective of each slow drive. Figure 3.5b shows the fraction of slowdown occurrences that arrive within X hours of the preceding slowdown; the arrival rates are binned by hour.

Finding #5: *Slowdown has a high temporal locality.* Figure 3.5b shows that 90% and 85% of disk and SSD slowdown occurrences from the same drive happen within the same day of the previous occurrence respectively. The two findings above suggest that history-based tail mitigation strategies can be a fitting solution; a slowdown occurrence should be leveraged as a good indicator for the possibility of near-future slowdowns.

Slowdown Extent

We now characterize the *slowdown extent* (*i.e.*, fraction of drives that have experienced slowdowns) in two ways. First, Figure 3.6a plots the fraction of all drives that have exhibited at least one occurrence of at least X-time slowdown ratio as plotted on the x-axis.

Finding #6: *A large extent of drive population has experienced slowdowns at different rates.* Figure 3.6a depicts that 26% and 29% of disk and SSD drives have exhibited ≥2x slowdowns at least one time in their lifetimes respectively. The fraction is also relatively significant for large slowdowns. For example, 1.6% and 2.5% of disk and SSD populations have experienced ≥16x slowdowns at least one time.

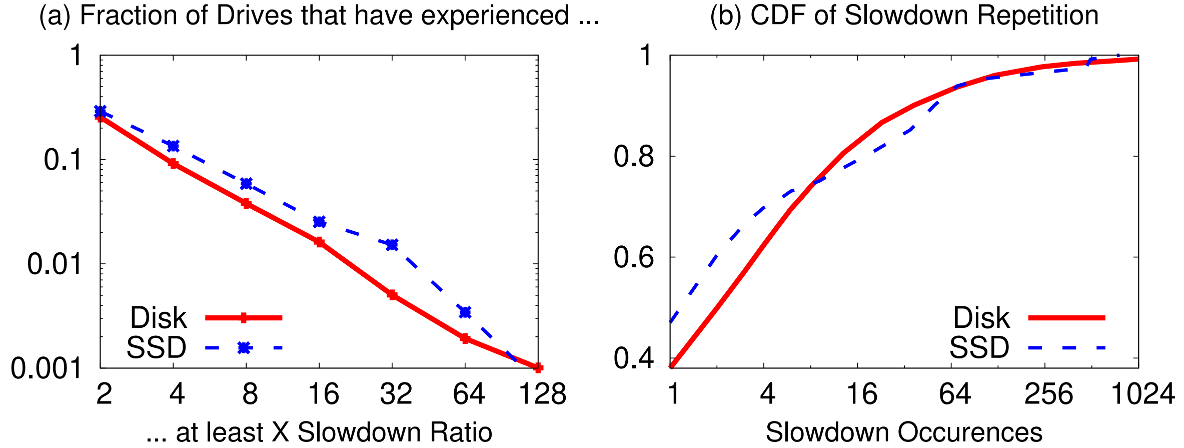

Figure 3.6: **Slowdown extent (Section 3.2.1).** *Figure (a) shows the fraction of all drives that have experienced at least one occurrence of X-time slowdown ratio as plotted on the x-axis; the y-axis is in log_{10} scale. Figure (b) shows the fraction of slow drives that has exhibited at least X slowdown occurrences.*

Next, we take only the population of slow drives (26% and 29% of the disk and SSD population) and plot the fraction of slow drives that has exhibited at least X slowdown occurrences, as shown in Figure 3.6b.

Finding #7: *Few slow drives experience a large number of slowdown repetitions.* Figure 3.6b shows that that around 6% and 5% of slow disks and SSDs exhibit at least 100 slowdown occurrences respectively. The majority of slow drives only incur few slowdown repetitions. For example, 62% and 70% of slow disks and SSDs exhibit only less than 5 slowdown occurrences respectively. We emphasize that frequency of slowdown occurrences above are *only within the time duration of 87 days on average* (Section 3.1.2).

3.2.2 Workload Analysis

The previous section presents the basic characteristics of drive slowdowns. We now explore the possible root causes, starting with workload analysis. Drive slowdowns are often attributed to unbalanced workload (*e.g.*, a drive is busier than other drives). We had a hypothesis that such is not the case in our study due to the storage stack optimization (Section 3.1.2). To explore our hypothesis, we correlate slowdown with two workload-related metrics: size and rate imbalance.

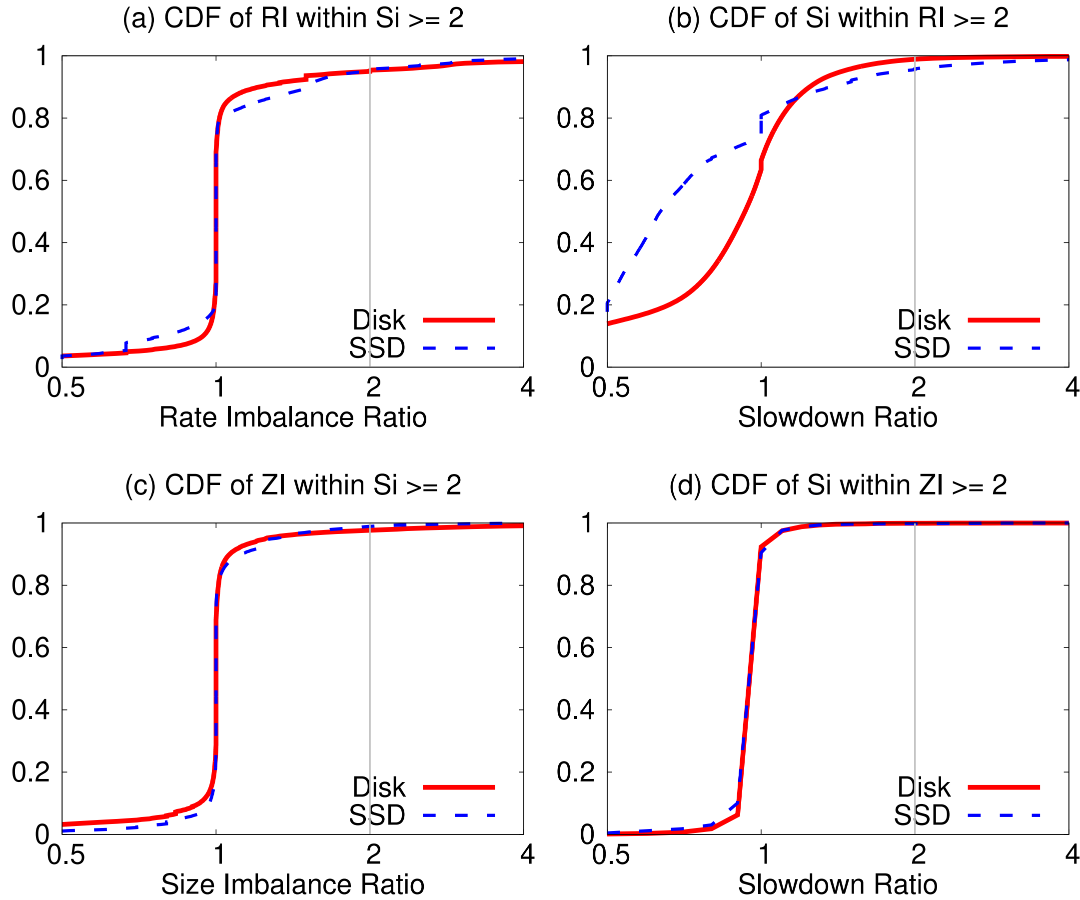

Figure 3.7: **CDF of size and rate imbalance (Section 3.2.2).** *Figure (a) plots the rate imbalance distribution (RI_i) within the population of slow drive hours ($S_i \geq 2$). A rate imbalance of X implies that the slow drive serves X times more I/Os, as plotted in the x-axis. Reversely, Figure (b) plots the slowdown distribution (S_i) within the population of rate-imbalanced drive hours ($RI_i \geq 2$). Figures (c) and (d) correlate slowdown and size imbalance in the same way as Figures (a) and (b).*

Slowdown vs. Rate Imbalance

We first measure the hourly I/O count for every drive (R_i), the median (R_{med}), and the rate imbalance ($RI_i = R_i/R_{med}$); this method is similar to the way we measure S_i in Table 3.2. *If* workload is to blame for slowdowns, then we should observe a high correlation between slowdown (S_i) and rate imbalance (R_i). That is, slowdowns happen in conjunction with rate imbalance, for example, $S_i \geq 2$ happens during $R_i \geq 2$.

Figure 3.7a shows the rate imbalance distribution (RI_i) *only within* the population of slow

drive hours. A rate imbalance of X (on the x-axis) implies that the slow drive serves X times more I/Os. The figure reveals that *only* 5% of slow drive hours happen when the drive receives 2x more I/Os than the peer drives. 95% of the slowdowns happen in the absence of rate imbalance (the rate-imbalance distribution is mostly aligned at x=1).

To strengthen our conjecture that rate imbalance is not a factor, we perform a reverse analysis. To recap, Figure 3.7a essentially shows how often slowdowns are caused by rate imbalance. We now ask the reverse: how often does rate imbalance cause slowdowns? The answer is shown in Figure 3.7b; it shows the slowdown distribution (S_i) *only within* the population of "overly" rate-imbalanced drive hours ($RI_i \geq 2$). Interestingly, rate imbalance has negligible effect on slowdowns; only 1% and 5% of rate-imbalanced disk and SSD hours experience slowdowns. From these two analyses, we conclude that rate imbalance is not a major root cause of slowdown.

Slowdown vs. Size Imbalance

Next, we correlate slowdown with size imbalance. Similar to the method above, we measure the hourly average I/O size for every drive (Z_i), the median (Z_{med}), and the size imbalance ($ZI_i = Z_i/Z_{med}$). Figure 3.7c plots the size imbalance distribution (ZI_i) only within the population of slow drive hours. A size imbalance of X implies that the slow drive serves X times larger I/O size. The size-imbalance distribution is very much aligned at x=1. Only 2.5% and 1.1% of slow disks and SSDs receive 2x larger I/O size than the peer drives in their group. Reversely, Figure 3.7d shows that only 0.1% and 0.2% of size-imbalanced disk and SSD hours experience more than 2x slowdowns.

Finding #8: *Slowdowns are independent of I/O rate and size imbalance.* As elaborated above, the large majority of slowdown occurrences (more than 95%) cannot be attributed to workload (I/O size or rate) imbalance.

3.2.3 Other Correlations

As workload imbalance is not a major root cause of slowdowns, we now attempt to find other possible root causes by correlating slowdowns with other metrics such as drive events, age, model and time of day.

Drive Event

Slowdown is often considered as a "silent" fault that needs to be monitored continuously. Thus, we ask: are there any *explicit* events surfacing near slowdown occurrences? To answer this, we collect *drive events* from our auto-support system.

Finding #9: *Slowdown is a "silent" performance fault.* A large majority of slowdowns are not accompanied with any explicit drive events. Out of the millions slow drive hours, we only observe hundreds of drive events. However, when specific drive events happen (specifically, "disk is not spinning" and "disk is not responding"), 90% of the cases lead to slowdown occurrences. We rarely see storage timeouts (*e.g.*, SCSI timeout) because timeout values are typically set coarsely (*e.g.*, 60 seconds). Since typical latency ranges from tens of microseconds to few milliseconds, a slowdown must be five orders of magnitude to hit a timeout. Thus, to detect tail latencies, storage performance should be monitored continuously.

Drive Age

Next, we analyze if drive age matters to performance stability. We break the the slowdown distribution (S_i) by different ages (*i.e.*, how long the drive has been deployed) as shown in Figure 3.8.

For disks, the bold lines in Figure 3.8a clearly show that older disks experience more slowdowns. Interestingly, the population of older disks is small in our dataset and yet we can easily observe slowdown prevalence within this small population (the population of 6-10 year-old disks

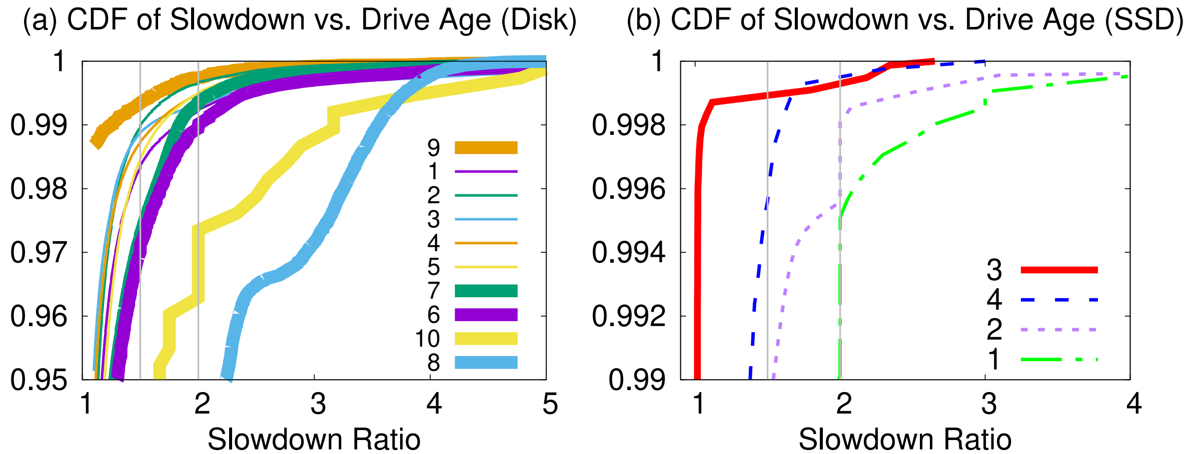

Figure 3.8: **Drive age (Section 3.2.3).** *The figures plot the slowdown distribution across different (a) disk and (b) SSD ages. Each line represents a specific age by year. Each figure legend is sorted from the left-most to right-most lines.*

ranges from 0.02-3% while 1-5 year-old disks ranges from 8-33%). In the worst case, the 8th year, the 95th percentile already reaches 2.3x slowdown. The 9th year (0.11% of the population) seems to be an outlier. Performance instability from disk aging due to mechanical wear-out is a possibility (Section 2.1).

For SSD, we do not observe a clear pattern. Although Figure 3.8b seemingly shows that young SSDs experience more slowdowns than older drives, it is hard to make such as a conclusion because of the small old-SSD population (3-4 year-old SSDs only make up 16% of the population while the 1-2 year-old is 83%).

Finding #10: Older disks tend to exhibit more slowdowns. For SSDs, no high degree of correlation can be made between slowdown and drive age.

Drive Model

We now correlate slowdown with drive model. Not all of our customers upload the model of the drives they use. Only 70% and 86% of customer disks and SSDs have model information. Thus, our analysis in this section is based on partial population.

We begin by correlating SSD model and slowdown. The SSD literature highlights the pressure

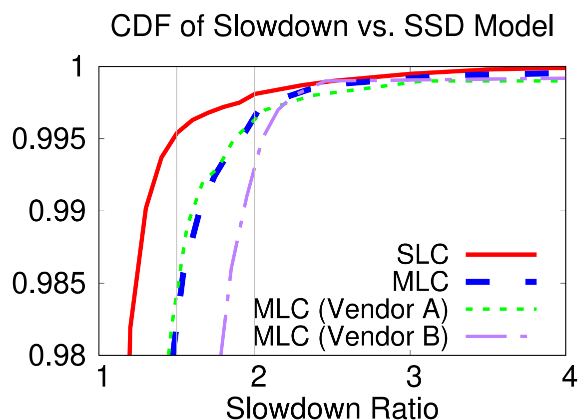

Figure 3.9: **SSD models (Section 3.2.3).** *The figure plots the slowdown distribution across different SSD models and vendors.*

to increase density, which leads to internal idiosyncrasies that can induce performance instability. Thus, it is interesting to know the impact of different flash cell levels to SSD performance stability.

Finding #11: *SLC slightly outperforms MLC drives in terms of performance stability.* As shown in Figure 3.9, at 1.5x slowdown threshold, MLC drives only reaches 98.2^{th} percentile while SLC reaches 99.5^{th} percentile. However, at 2x slowdown threshold, the distribution is only separated by 0.1%. As MLC exhibits less performance stability than SLC, future comparisons with TLC drives will be interesting.

Our dataset contains about 60:40 ratio of SLC vs. MLC drives. All the SLC drives come from one vendor, but the MLC drives come from two vendors with 90:10 population ratio. This allows us to compare vendors.

Finding #12: *SSD vendors seem to matter.* As shown by the two thin lines in Figure 3.9, one of the vendors (the 10% MLC population) has much less stability compared to the other one. This is interesting because the instability is clearly observable even within a small population. At 1.5x threshold, this vendor's MLC drives already reach 94.3^{th}percentile (out of the scope of Figure 3.9).

For disks, we use different model parameters such as storage capacity, RPM, and SAN interfaces (SATA, SAS, or Fibre Channel). However, we do not see any strong correlation.

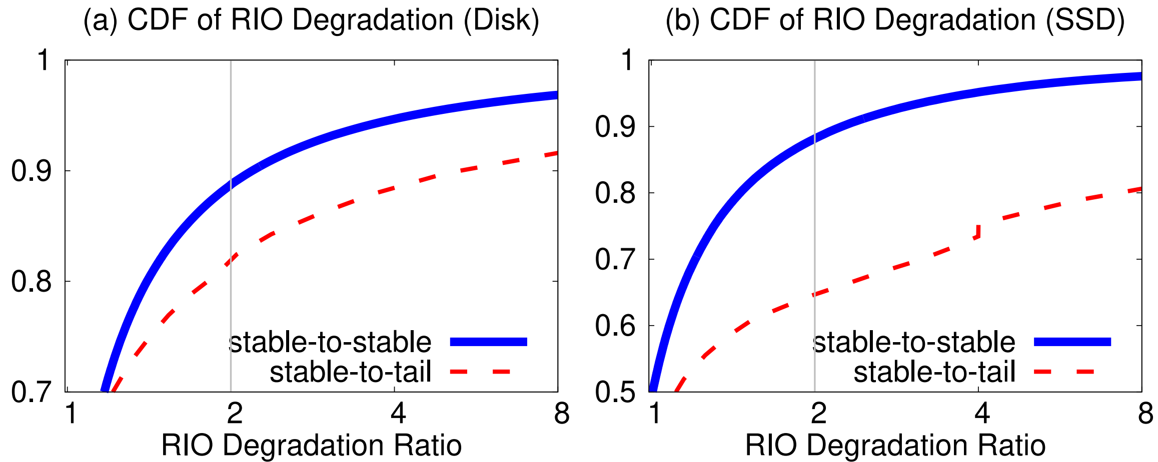

Figure 3.10: **RAID I/O degradation (Section 3.2.4).** *The figures contrast the distributions of RIO degradation between stable-to-stable and stable-to-tail transitions.*

Time of Day

We also perform an analysis based on time of day to identify if night-time background jobs such as disk scrubbing cause slowdowns. We find that slowdowns are uniformly distributed throughout the day and night.

3.2.4 Post-Slowdown Analysis

We now perform a post-mortem analysis: what happens *after* slowdown occurrences? We analyze this from two angles: RAID performance degradation and unplug/replug events.

RAID Performance Degradation

A slow drive has the potential to degrade the performance of the entire RAID, especially for full-stripe workload common in the studied RAID systems (Section 3.1.1), it is reasonable to make the following hypothesis: during the hour when a drive slows down, the RAID aggregate throughput will drop as the workload's intensity will be throttled by the slow drive. Currently, we do not have access to throughput metrics at the file system or application levels, and even if we do, connecting metrics from different levels will not be trivial. We leave cross-level analysis as future work, but

meanwhile, given this constraint, we perform the following analysis to explore our hypothesis.

We derive a new metric, ***RIO*** (hourly RAID I/O count), which is the aggregate number of I/Os per hour from all the data drives in every RAID hour. Then, we derive ***RIO degradation*** (RAID throughput degradation) as the ratio $\text{RIO}_{lastHour}$ to $\text{RIO}_{currentHour}$. If the degradation is larger than one, it means the RAID group serves less I/Os than the previous hour.

Next, we distinguish *stable-to-stable* and *stable-to-tail* transitions. Stable RAID hour means all the drives are stable ($Si < 2$). Tail RAID hour implies at least one of the drives is slow. In stable-to-stable transitions, RIO degradation can naturally happen as workload "cools down". Thus, we first plot the distribution of stable-to-stable RIO degradation, shown by the solid blue line in Figure 3.10. We then select only the stable-to-tail transitions and plot the RIO degradation distribution, shown by the dashed red line in Figure 3.10.

Finding #13: *A slow drive can significantly degrade the performance of the entire RAID.* Figure 3.10 depicts a big gap of RAID I/O degradation between stable-to-stable and stable-to-tail transitions. In SSD-based RAID, the degradation impact is quite severe. Figure 3.10b for example shows that only 12% of stable-to-stable transitions observe ≥2x RIO degradation (likely from workload cooling down). However, in stable-to-slow transitions, there is *23% more chance* (the vertical gap at x=2) that RIO degrades by more than 2x. In disk-based RAID, RIO degradation is also felt with 7% more chance. This finding shows the real possibilities of workload throughput being degraded and stable drives being under-utilized during tail hours, which again motivates the need for tail-tolerant RAID.

We note that RAID degradation is felt more if user requests are casually dependent; RIO degradation only affects I/Os that are waiting for the completion of previous I/Os. Furthermore, since our dataset is based on hourly average latencies, there is no sufficient information that shows every I/O is delayed at the same slowdown rate. We believe these are the reasons why we do not see a complete collapse of RIO degradation.

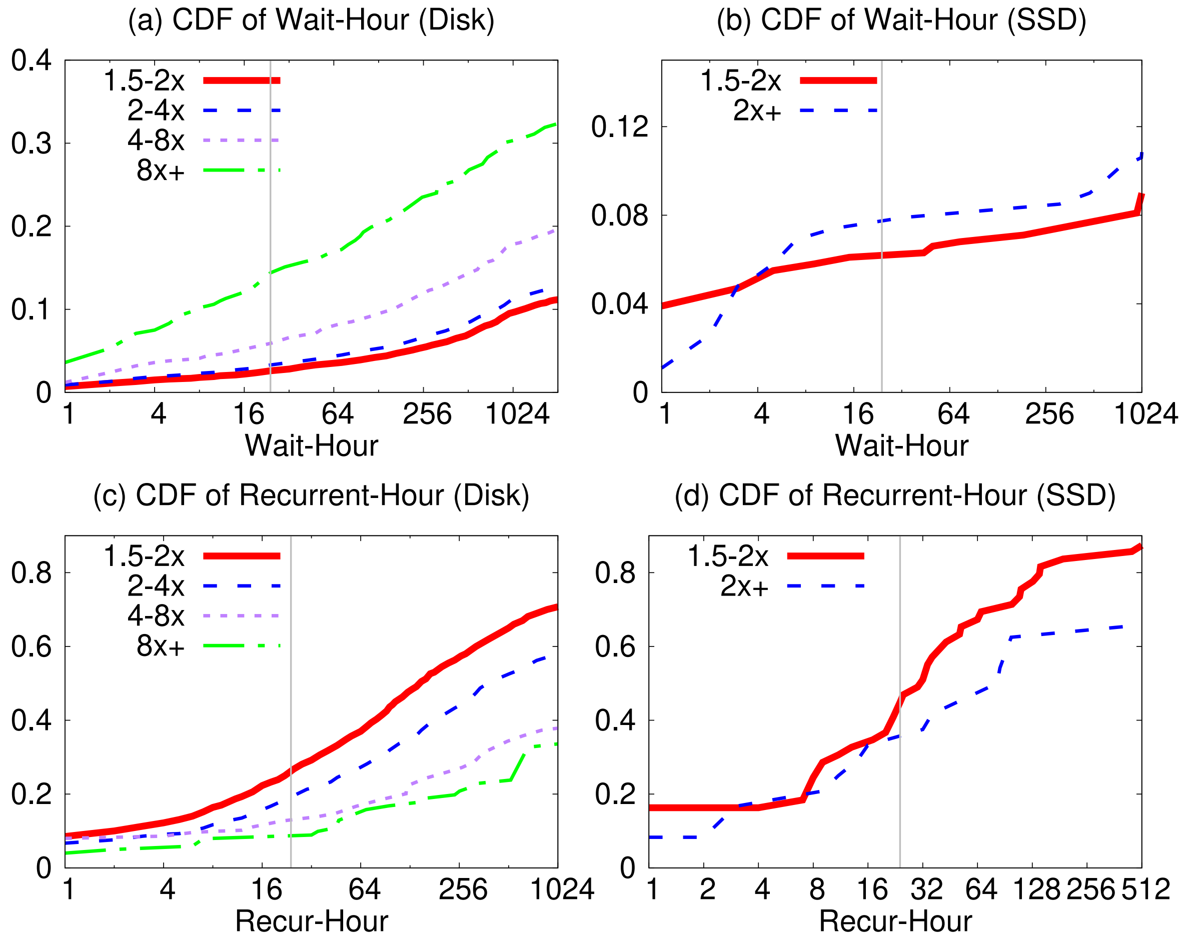

Figure 3.11: **Unplug/replug events (Section 3.2.4-Section 3.2.4).** *The figures show the relationships between slowdown occurrences and unplug/replug events. The top and bottom figures show the distribution of "wait-hour" and "recur-hour" respectively.*

Unplug Events

When a drive slows down, the administrator might unplug the drive (*e.g.*, for offline diagnosis) and later replug the drive. Unplug/replug is a manual administrator's process, but such events are logged in our auto-support system. To analyze unplug patterns, we define *wait-hour* as the number of hours between a slowdown occurrence and a subsequent unplug event; if a slowdown persists in consecutive hours, we only take the first slow hour. Figures 3.11a-b show the wait-hour distribution within the population of slow disks and SSDs respectively.

Finding #14: *Unplug events are common.* Figures 3.11a-b show that within a day, around 4% and 8% of slow (≥2x) disks and SSDs are unplugged respectively. For "mild" slowdowns (1.5-

2x), the numbers are 3% and 6%. Figure 3.11a also shows a pattern where disks with more severe slowdowns are unplugged at higher rates; this pattern does not show up in SSD.

Replug Events

We first would like to note that unplug is not the same as drive replacement; a replacement implies an unplug without replug. With this, we raise two questions: What is the replug rate? Do replugged drives exhibit further slowdowns? To analyze the latter, we define *recur-hour* as the number of hours between a replug event and the next slowdown occurrence. Figures 3.11c-d show the recur-hour distribution within the population of slow disks and SSDs respectively.

Finding #15: *Replug rate is high and slowdowns still recur after replug events.* In our dataset, customers replug 89% and 100% of disks and SSDs that they unplugged respectively (not shown in figures). Figures 3.11c-d answer the second question, showing that 18% and 35% of replugged disks and SSDs exhibit another slowdown within a day. This finding points out that administrators are reluctant to completely replace slow drives, likely because slowdowns are transient (not all slowdowns appear in consecutive hours) and thus cannot be reproduced in offline diagnosis and furthermore the cost of drive replacement can be unnecessarily expensive. Yet, as slowdown can recur, there is a need for online tail mitigation approaches.

In terms of unplug-replug duration, 54% of unplugged disks are replugged within 2 hours and 90% within 10 hours. For SSD, 61% are replugged within 2 hours and 97% within 10 hours.

3.2.5 Summary

It is now evident that storage performance instability at the drive level is not uncommon. One of our major findings is the little correlation between performance instability and workload imbalance. One major analysis challenge is the "silent" nature of slowdowns; they are not accompanied by explicit drive events, and therefore, pinpointing the root cause of each slowdown occurrence is still an open problem. However, in terms of the overall findings, our conversations with product teams

and vendors [36] confirm that many instances of drive performance faults are caused by drive anomalies; there are strong connections between our findings and some of the anecdotal evidence we gathered (Section 2.1). As RAID deployments can suffer from storage tails, we next discuss the concept of tail-tolerant RAID.

3.3 Tail-Tolerant RAID

With drive performance instability, RAID performance is in jeopardy. When a request is striped across many drives, the request cannot finish until all the individual I/Os complete (Figure 3.1); the request latency will follow the tail latency. As request throughput degrades, stable drives become under-utilized. Tail-tolerant RAID is one solution to the problem and it brings two advantages.

First, slow drives are masked. This is a simple goal but crucial for several reasons: stringent SLOs require stability at high percentiles (*e.g.*, 99% or even 99.9% [101, 244, 249, 264]); slow drives, if not masked, can create cascades of performance failures to applications [103]; and drive slowdowns can falsely signal applications to back off, especially in systems that treat slowdowns as hints of overload [120].

Second, tail-tolerant RAID is a cheaper solution than drive replacements, especially in the context of transient slowdowns (Section 3.2.1) and high replug rates by administrators (Section 3.2.4). Unnecessary replacements might be undesirable due to the hardware cost and the expensive RAID re-building process as as drive capacity increases.

Given these advantages, we performed an opportunity assessment of tail-tolerant strategies at the RAID level. We emphasize that the main focus of this paper is the large-scale analysis of storage tails; the initial exploration of tail-tolerant RAID in this section is only to assess the benefits of such an approach.

3.3.1 Tail-Tolerant Strategies

We explore three tail-tolerant strategies: *reactive*, *proactive*, and *adaptive*. They are analogous to popular approaches in parallel distributed computing such as speculative execution [100] and hedging/cloning [62, 99]. To mimic our RAID systems (Section 3.1.2), we currently focus on tail tolerance for RAID-6 with *non-rotating* parity (Figure 3.1 and Section 3.1.1). We name our prototype ToleRAID, for simplicity of reference.

Currently, we only focus on full-stripe read workload where ToleRAID can cut "read tails" in the following ways. In normal reads, the two parity drives are unused (if no errors), and thus can be leveraged to mask up to two slow data drives. For example, if one data drive is slow, ToleRAID can issue an extra read to one parity drive and rebuild the "late" data.

Reactive: A simple strategy is reactive. If a drive (or two) has not returned the data for STx (slowdown threshold) longer than the median latency, reactive will perform an extra read (or two) to the parity drive(s). Reactive strategy should be enabled by default in order to cut extremely long tails. It is also good for mostly stable environment where slowdowns are rare. A small ST will create more extra reads and a large ST will respond late to tails. We set $ST = 2$ in our evaluation, which means we still need to wait for roughly an additional 1x median latency to complete the I/O (a total slowdown of 3x in our case). While reactive strategies work well in cluster computing (*e.g.*, speculative execution for medium-long jobs), they can react too late for small I/O latencies (*e.g.*, hundreds of microseconds). Therefore, we explore proactive and adaptive approaches.

Proactive: This approach performs extra reads to the parity drives concurrently with the original I/Os. The number of extra reads can be one (P drive) or two (P and Q); we name them PROACTIVE_1 and PROACTIVE_2 respectively. Proactive works well to cut short tails (near the slowdown threshold); as discussed above, reactive depends on ST and can be a little bit too late. The downside of proactive strategy is the extra read traffic.

Adaptive: This approach is a middle point between the two strategies above. Adaptive by default

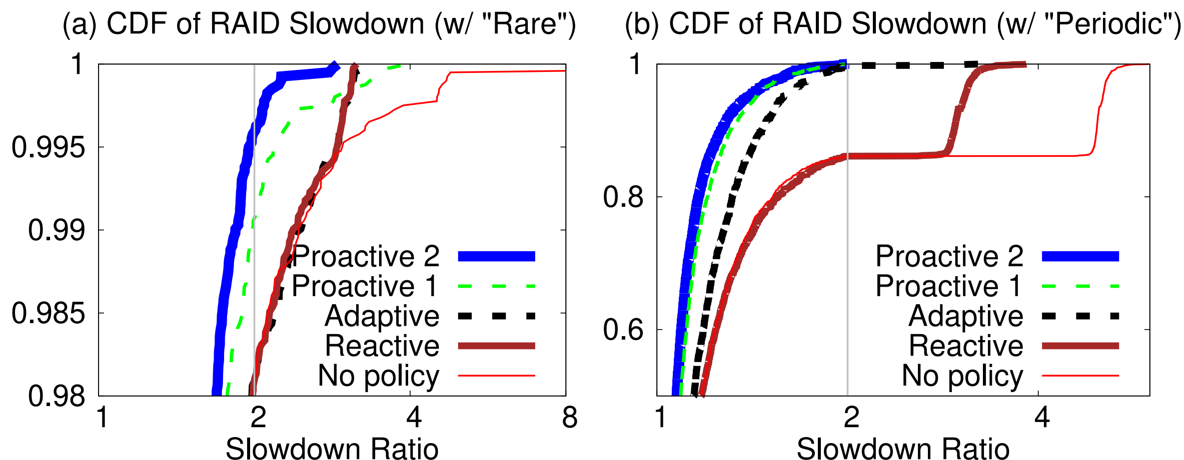

Figure 3.12: **ToleRAID evaluation.** *The figures show the pros and cons of various ToleRAID strategies based on two slowdown distributions: (a) Rare and (b) Periodic. The figures plot the T^1 distribution (i.e., the RAID slowdown). T^1 is essentially based on the longest tail latency among the necessary blocks that each policy needs to wait for.*

runs the reactive approach. When the reactive policy is triggered repeatedly for SR times (slowdown repeats) on the same drive, then ToleRAID becomes proactive until the slowdown of the offending drive is less than ST. If two drives are persistently slow, then ToleRAID runs PROACTIVE$_2$. Adaptive is appropriate for instability that comes from persistent and periodic interferences such as background SSD GC, SMR log cleaning, or I/O contention from multi-tenancy.

3.3.2 Evaluation

Our user-level ToleRAID prototype stripes each RAID request into 4-KB chunks (Section 3.1.2), merge consecutive per-drive chunks, and submit them as direct I/Os. We insert a delay-injection layer that emulates I/O slowdowns. Our prototype takes two inputs: block-level trace and slowdown distribution. Below, we show ToleRAID results from running a block trace from Hadoop Wordcount benchmark, which contains mostly big reads. We perform the experiments on 8-drive RAID running IBM 500GB SATA-600 7.2K disk drives.

We use two slowdown distributions: (1) *Rare* distribution, which is uniformly sampled from our disk dataset (Figure 3.4a). Here, tails (T^1) are rare (1.5%) but long tails exist (Table 3.3). (2)

Periodic distribution, based on our study of Amazon EC2 ephemeral SSDs (not shown due to space constraints). In this study, we rent SSD nodes and found a case where one of the locally-attached SSDs periodically exhibited 5x read slowdowns that lasted for 3-6 minutes and repeated every 2-3 hours (2.3% instability period on average).

Figure 3.12 shows the pros and cons of the four policies using the two different distributions. In all cases, PROACTIVE$_1$ and PROACTIVE$_2$ always incur roughly 16.7% and 33.3% extra reads. In Figure 3.12a, REACTIVE can cut long tails and ensure RAID only slows down by at most 3x, while only introducing 0.5% I/O overhead. PROACTIVE$_2$, compared to PROACTIVE$_1$, gives slight benefits (*e.g.*, 1.8x vs. 2.3x at 99th percentile). Note also that PROACTIVE$_1$ does not use REACTIVE, and thus PROACTIVE$_1$ loses to REACTIVE within the 0.1% chance where two disks are slow at the same time. ADAPTIVE does not show more benefits in non-persistent scenarios. Figure 3.12b shows that in periodic distribution with persistent slowdowns, ADAPTIVE works best; it cuts long tails but only incurs 2.3% I/O overhead.

Overall, ToleRAID shows potential benefits. In separate experiments, we have also measured Linux Software RAID degradation in the presence of storage tails (with `dmsetup` delay utilities). We are integrating ToleRAID to Linux Software RAID and extending it to cover more policies and scenarios (partial reads, writes, etc.).

3.4 Discussions

We hope our work will spur a set of interesting future research directions for the larger storage community to address. We discuss this in the context of performance-log analysis and tail mitigations.

Enhanced data collection: The first limitation of our dataset is the hourly aggregation, preventing us from performing micro analysis. Monitoring and capturing fine-grained data points will incur high computation and storage overhead. However, during problematic periods, future monitoring systems should capture detailed data. Our ToleRAID evaluation hints that realistic slowdown

distributions are a crucial element in benchmarking tail-tolerant policies. More distribution benchmarks are needed to shape the tail-tolerant RAID research area. The second limitation is the absence of other metrics that can be linked to slowdowns (*e.g.*, heat and vibration levels). Similarly, future monitoring systems can include such metrics.

Our current SSD dataset is two orders of magnitude smaller than the disk dataset. As SSD becomes the front-end storage in datacenters, larger and longer studies of SSD performance instability is needed. Similarly, denser SMR disk drives will replace old generation disks [55, 110]. Performance studies of SSD-based and SMR-based RAID will be valuable, especially for understanding the ramifications of internal SSD garbage-collection and SMR cleaning to the overall RAID performance.

Further analyses: Correlating slowdowns to latent sector errors, corruptions, drive failures (*e.g.*, from SMART logs), and application performance would be interesting future work. One challenge we had was that not all vendors consistently use SMART and report drive errors. In this paper, we use median values to measure tail latencies and slowdowns similar to other work [99, 183, 264]. We do so because using median values will not hide the severity of long tails. Using median is exaggerating if (N-1)/2 of the drives have significantly higher latencies than the rest; however, we did not observe such cases. Finally, we mainly use 2x slowdown threshold, and occasionally show results from a more conservative 1.5x threshold. Further analyses based on average latency values and different threshold levels are possible.

Tail mitigations: We believe the design space of tail-tolerant RAID is vast considering different forms of RAID (RAID-5/6/10, etc.), different types of erasure coding [224], various slowdown distributions in the field, and diverse user SLA expectations. In our initial assessment, ToleRAID uses a black-box approach, but there are other opportunities to cut tails "at the source" with transparent interactions between devices and the RAID layer. In special cases such as materials trapped between disk head and platter (which will be more prevalent in "slim" drives with low heights), the file system or RAID layer can inject random I/Os to "sweep" the dust off. In summary, each root

cause can be mitigated with specific strategies. The process of identifying all possible root causes of performance instability should be continued for future mitigation designs.

3.5 Related Work

Large-scale storage studies at the same scale as ours were conducted for analysis of whole-disk failures [222, 235], latent sector errors [70, 194], and sector corruptions [71]. Many of these studies were started based on anecdotal evidence of storage faults. Today, as these studies had provided real empirical evidence, it is a common expectation that storage devices exhibit such faults. Likewise, our study will provide the same significance of contribution, but in the context of performance faults.

Krevat *et al.* [175] demonstrate that disks are like "snowflakes" (same model can have 5-14% throughput variance); they only analyze throughput metrics on 70 drives with simple microbenchmarks. To the best of our knowledge, our work is the first to conduct a large-scale performance instability analysis at the drive level.

Storage performance variability is typically addressed in the context of storage QoS (*e.g.*, mClock [121], PARDA [120], Pisces [239]) and more recently in cloud storage services (*e.g.*, C3 [244], CosTLO [264]). Other recent work reduces performance variability at the file system (*e.g.*, Chopper [134]), I/O scheduler (*e.g.*, split-level scheduling [272]), and SSD layers (*e.g.*, Purity [92], Flash on Rails [240]). Different than ours, these sets of work do not specifically target drive-level tail latencies.

Finally, as mentioned before, reactive, proactive and adaptive tail-tolerant strategies are lessons learned from the distributed cluster computing (*e.g.*, MapReduce [100], dolly [62], Mantri [64], KMN [253]) and distributed storage systems (*e.g.*, Windows Azure Storage [138], RobuSTore [265]). The applications of these high-level strategies in the context of RAID will significantly differ.

3.6 Conclusion

We have "transformed" anecdotes of storage performance instability into large-scale empirical evidence. Our analysis so far is solely based on last generation drives (few years in deployment). With trends in disk and SSD technology (*e.g.*, SMR disks, TLC flash devices), the worst might be yet to come; performance instability can be more prevalent in the future, and our findings are perhaps just the beginning. File and RAID systems are now faced with more responsibilities. Not only must they handle known storage faults such as latent sector errors and corruptions [71, 124, 125, 234], but also now they must mask drive tail latencies as well. Lessons can be learned from the distributed computing community where a large body of work has been born since the issue of tail latencies became a spotlight a decade ago [100]. Similarly, we hope "the tail at store" will spur exciting new research directions within the storage community.

CHAPTER 4

MITTOS: SUPPORTING MILLISECOND TAIL TOLERANCE WITH FAST REJECTING SLO-AWARE OS INTERFACE

Here, we introduce MITTOS (pronounced "mythos"), an OS that employs a fast rejecting SLO-aware interface to support millisecond tail tolerance. We materialize this concept within the storage software stack, primarily because storage devices are a major resource of contention [76, 130, 161, 200, 239, 248, 272]. In a nutshell, MITTOS provides an SLO-aware read interface, "`read(...,slo)`," such that applications can attach SLOs to their IO operations (*e.g.*, "`read()` should not take more than 20ms"). If the SLO cannot be satisfied (*e.g.*, long disk queue), MITTOS immediately rejects the IOs and returns `EBUSY` (*i.e.*, no wait), hence allowing the application to quickly failover (retry) to another node.

The biggest challenge in supporting a fast rejecting interface is the development of *latency prediction* used to determine whether the IO request should be accepted or rejected (returning `EBUSY`). Such prediction requires understanding the nature of contention and queueing discipline of the underlying resource (*e.g.*, disk spindles vs. SSD channels/chips, FIFO vs. priority). Furthermore, latency prediction must be fast; the computing effort to produce good predictions should be negligible to maintain high request rates. Finally, prediction must be accurate; vendor variations and device idiosyncrasies must be incorporated.

We demonstrate that these challenges can be met; we will describe our MITTOS design in four different OS subsystems: the disk noop scheduler (MITTNOOP), CFQ scheduler (MITTCFQ), SSD management (MITTSSD), and OS cache management (MITTCACHE). Collectively, these cover the major components that can affect an IO request latency. Our discussion will also cover the key design challenges and exemplar solutions.

To examine MITTOS can benefit applications, we study data-parallel storage such as distributed NoSQL systems. Examination shows that many NoSQL systems (*e.g.*, MongoDB) do not adopt tail-tolerance mechanisms (Section 2.2.1), and thus can benefit from MITTOS support.

To evaluate the benefits of MITTOS in a real multi-tenant setting, we collected statistics of memory, SSD, and disk contentions in Amazon EC2, observed from the perspective of a tenant (Section 4.4). Our most important finding is that the "noisy neighbor" problem exhibits sub-second burstiness, hence coa-rse latency monitoring (*e.g.*, snitching) is not effective, but timely latency prediction in MITTOS will help.

We evaluate our MITTOS-powered MongoDB in a 20-node cluster with YCSB workloads and the EC2 noise distribution. We compare MITTOS with three other standard practices (basic timeout, cloning, and hedged requests). Compared to hedged requests (the most effective among the three), MITTOS reduces the completion time of individual IO requests by 23-26% at p95[1] and 6-10% on average. Better, as tail latencies can be amplified by scale (*i.e.*, a user request can comprise S parallel requests and must wait for all to finish), with S=5, MITTOS reduces the completion time of hedged requests up to 35% at p95 and 16-23% on average. The higher the scale factor, the more reduction MITTOS delivers.

In summary, our contributions are: the new concept and principles of MITTOS (Section 4.1), design and working examples of MITTOS design in disk, SSD, and OS cache managements (Section 4.2), the statistics of sub-second IO burstiness in Amazon EC2 (Section 4.4), and demonstration that MITTOS-powered storage systems (MongoDB and LevelDB) can leverage fast IO rejection to achieve significant latency reductions in compared to other advanced techniques (Section 4.5). We close with discussion, related work, and conclusion.

4.1 MITTOS Overview

4.1.1 Deployment Model and Use Case

MITTOS suits the deployment model of data-parallel frameworks running on multi-tenant machines, as illustrated in Figure 4.1. Here, every machine has local storage resources (*e.g.*, disk)

1. We use "pY" to denote the Yth-percentile; for example, p90 implies the 90th-percentile (y=0.9 in CDF graphs).

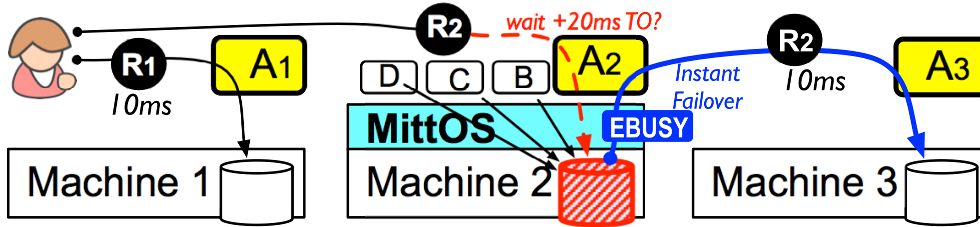

Figure 4.1: **MITTOS Deployment Model (Section 4.1.1).**

directly managed by the *host OS*. On top, different *tenants/applications* ($A...D$) share the same machine. Let us consider a single data-parallel storage (*e.g.*, MongoDB) deployed as applications $A_1 - A_3$ across machines #1-3 and the data (key-values) will be replicated three times across the three machines. Imagine a user sending two parallel requests R_1 to A_1 and R_2 to A_2, each supposedly takes only 10ms (the term "*user*" implies the application's users). If the disk in machine #2 is busy because other tenants ($B/C/D$) are busy using the disk, ideally MongoDB should quickly retry the request R_2 to another replica A_3 on machine #3.

In *wait*-and-speculate approaches, request R_2 is only retried after some time has elapsed (*e.g.*, 20ms), resulting in R_2's completion time of roughly 30ms, a tail latency 3x longer than R_1's latency. In contrast, MITTOS will instantly return EBUSY (*no wait* in the application), resulting in a completion time of only 10+e ms; e is a one-hop network overhead.

In the above model, MITTOS is integrated to the *host OS* layer from where applications can get direct notification of resource busyness. However, our model is similar to container- or VM-based multi-tenancy models, where MITTOS can be integrated jointly across the host OS and VMM or container-engine layers. For faster research prototyping, in this paper we mainly focus on direct application-to-host model, but MITTOS can also be extended to the VMM layer. MITTOS principles will remain the same across these models.

4.1.2 Use Case

Figure 4.2 shows a simple use-case illustration of MITTOS. ① The application (*e.g.*, MongoDB) creates an SLO for a user. In this paper, we use *latency deadline* (*e.g.*, <20ms) as a form of SLO,

but more complex forms of SLO such as throughput or deadline with confidence interval can be explored in future work (Section 4.6.1). We use the 95th-percentile latency as the deadline value, which we will discuss more in Sections 4.5.2 and 4.6, to what value a deadline should be set. ② The application then tags `read()` calls with the deadline SLO. To support this, we create a new `read()` system call that can accept application SLO (essentially one extra argument to the existing `read()` system call). ③ As the IO request enters a resource queue in the kernel, MITTOS checks if the deadline SLO can be satisfied. ④ If the deadline SLO will be violated in the resource queue, MITTOS will instantly return `EBUSY` error code to the application. ⑤ Upon receiving `EBUSY`, the application can quickly failover (retry) the request to another replica node.

4.1.3 Goals / Principles

MITTOS advocates the following principles.

- *Fast rejection ("busy is error"):* In the PC era, the OS must be best-effort; returning busy errors is undesirable as PC applications cannot retry elsewhere. However, in tail-critical datacenter applications, best effort interface is insufficient to help applications manage ms-level tails. Datacenter applications inherently run on redundant machines, thus there is no "shame" for the OS to reject IOs. In large-scale deployments, this principle works well, as the probability of all replicas busy at the same time is extremely low (Section 4.4).

- *SLO aware:* Applications should expose their SLOs to the OS, such that the OS only rejects IOs whose SLOs cannot be met (due to resource busyness).

- *Instant feedback/failover:* The sub-ms fast rejection gives ms-level operations more flexibility to failover quickly. Making a system call and receiving `EBUSY` only takes <5µs (③ and ④ in Figure 4.2). Failing over to another machine (⑤ in Figure 4.2) only involves one more network hop (*e.g.*, 0.3ms in EC2 and our testbed or even 10µs with Infiniband [215]).

- *Keep existing OS policies:* MITTOS' simple interface extensions allow existing OS optimizations and policies to be preserved. MITTOS does *not* negate nor replace all prior advance-

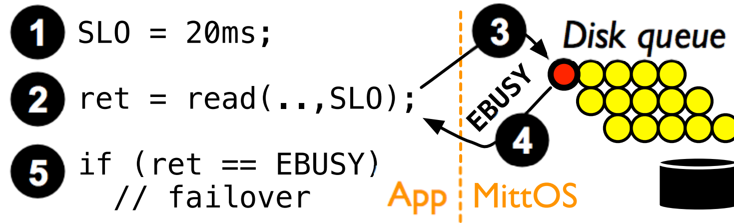

Figure 4.2: **MITTOS use-case illustration (Section 4.1.2).**

ments in the QoS literature. We only advocate that applications get notified when OS-level QoS policies fail to meet user deadlines due to unexpected bursty contentions. For example, even with CFQ fairness [4], IOs from high-priority processes occasionally must wait for lower-priority ones to finish. As another example, in SSDs, even with advanced isolation techniques, garbage collection or wear-level-ing activities can induce a heavy background noise.

- *Keep applications simple:* Advanced tail-tolerance mechanisms such as tied requests and IO revocation are less needed in applications. These mechanisms are now pushed to the OS layer, which then can be reused by many applications. In MITTOS, the rejected request is *not* queued (step ④ in Figure 4.2); it is *automatically cancelled* when the deadline is violated. Thus, applications do not need to wait or revoke IOs, nor they add more contentions to the already-contended resources. MITTOS also keeps application failover logic simple and sequential (the sequence of ②-⑤ in Figure 4.2).

4.1.4 Design Challenges

The biggest challenge of integrating MITTOS to a target resource and its management is the EBUSY prediction (*i.e.*, whether the arriving IO should be accepted or rejected). There are three major challenges: **(1)** We must understand the contention nature and queueing discipline of the target resource. For example, in disks, the spindle is the resource of contention, but in SSDs, parallel chips/channels exhibit independent queueing delays. Furthermore, the target resource can be managed by different queuing disciplines (noop/FIFO, CFQ [4], anticipatory [145], etc.). Thus, EBUSY prediction will vary across different resources and schedulers. **(2)** In terms of performance

overhead, latency prediction should ideally be $O(1)$ for every arriving IO. $O(N)$ prediction that iterates through N pending IOs is not desirable. **(3)** In terms of accuracy, different device types/vendors have different latency characteristics (*e.g.*, varying seek costs across disks, page-level latency variability within an SSD).

4.2 Case Studies

The goal of this section is to demonstrate that MITTOS principles can be integrated to many resource managements such as the disk noop (Section 4.2.1) and CFQ (Section 4.2.2) IO schedulers, SSD (Section 4.2.3) and OS cache (Section 4.2.4) managements. In each integration, we describe how we address the three challenges (understanding the resource contention nature and fast and accurate latency prediction).

*4.2.1 Disk Noop Scheduler (*MITTNOOP*)*

Our first and simplest integration is to the noop scheduler. The use of noop for disk is actually discouraged, but the goal of our description below is to explain the basic mechanisms of MITTOS, which will be re-used in subsequent sections.

Resource and deadline checks: In noop, arriving IOs are put to a FIFO dispatch queue whose items will be absorbed to the disk's device queue. The logic of MITTNOOP is relatively simple: when an IO arrives, it calculates the IO's wait time (T_{wait}) given all the pending IOs in the dispatch and device queues. If $T_{wait} > T_{deadline} + T_{hop}$, then EBUSY is returned; T_{hop} is a constant of 0.3ms one-hop failover in our testbed.

Performance: A naive $O(N)$ way to perform a deadline check is to sum all the N pending IOs' processing times. To make deadline check $O(1)$, MITTNOOP keeps track the disk's next free time ($T_{nextFree}$), as explained below. The arriving IO's wait time is simply the difference of the current and next free time ($T_{wait} = T_{nextFree} - T_{now}$). If the disk is currently free ($T_{nextFree} < T_{now}$), the IO is submitted directly.

Accuracy: When an IO is accepted, MITTNOOP adds the next free time with the predicted processing time to serve the new IO ($T_{nextFree}$+=$T_{processNewIO}$). To make $T_{nextFree}$ accurate, $T_{processNewIO}$ must be precise. To achieve that, we must profile the disk's read/write latency, specifically the relationships between IO sizes, jump distances, and latencies. In a nutshell, $T_{processNewIO}$ is a function of the size and offset of the current IO, the last IO completed, and all the IOs in the device queue. We defer the details to Appendix Section 4.9. Our one-time profiling takes 11 hours (disk is slow).

$T_{nextFree}$ will automatically be calibrated when the disk is idle ($T_{nextFree}$=T_{now} +$T_{processNewIO}$). However, under no-idle period, a slight error in $T_{nextFree}$+=$T_{processNewIO}$ will accumulate over time as thousands/millions of IOs are submitted. To calibrate more accurately, we attach $T_{processNewIO}$ and the IO's start time to the IO descriptor, such that upon IO completion, we can measure the "diff" of the actual and predicted processing time (T_{diff}=$T_{processActual}$ −$T_{processNewIO}$) and then calibrate the next free time ($T_{nextFree}$+=T_{diff}).

*4.2.2 Disk CFQ Scheduler (*MITTCFQ*)*

Next, we build MITTCFQ within the CFQ scheduler [4], the default and most sophisticated IO scheduler in Linux. We first describe the structure of CFQ and its policy.

Unlike noop, CFQ manages groups with time slices proportional to their weights. In every group, there are three *service trees* (RealTime/BestEffort/Idle). In every tree, there are *process nodes*. In every node, there is a *red-black tree* for sorting the process' pending IOs based on their on-disk offsets. Using `ionice`, applications can declare IO types (RealTime/BestEffort/Idle and 0-7 priority level). CFQ policy always picks IOs from the RealTime tree first, and then from BestEffort and Idle. In the chosen tree, it picks a node in round robin style, proportional to its time slice (0-7 priority level). Then, CFQ dispatch some or all the requests from the node's red-black tree. The requests will be put to a FIFO dispatch queue and eventually to the device queue.

Resource and deadline checks: When an IO arrive MITTCFQ needs to identify to which

group, service tree, and process node, the IO will be attached to. This is for predicting the IO wait time, which is the sum of the wait times of the current pending IOs in the device and dispatch queues as well as the IOs in other CFQ queues that are *in front of* the priority of the new IO's node. This raises the following challenges.

Performance: To avoid $O(N)$ complexity, MITTCFQ keeps track the predicted total IO time of each process node. This way, we reduce $O(N)$ to $O(P)$ where P is the number of processes with pending IOs in the CFQ queues. In our most-intensive test with 128 IO-intensive threads, iterating through all pending IOs' descriptors in the naive $O(N)$ method costs about 10-20µs. Our optimizations above (and more below) bring the overhead down to <5µs per IO prediction.

Accuracy: With the method above, MITTCFQ can reject IOs before they enter the CFQ queues. However, due to the nature of CFQ, some IOs can be accepted initially, but if soon new higher-priority IOs arrive, the deadlines of the earlier IOs can be violated as they are "bumped to the back." To cancel such IOs, the $O(P)$ technique above is not sufficient because a single process (*e.g.*, MongoDB) can have different IOs with different deadlines (from different users). Thus, MITTCFQ adds a hash table where the `key` is a tolerable time range and the `values` are the IOs with the same tolerable time (grouped by 1ms). For example, a recently-accepted IO (supposedly 6ms without noise) has a 25ms deadline but only a 10ms wait time, hence its tolerable time is 9ms. If a new higher-priority IO arrive with 6ms predicted processing time, the prior IO is not cancelled, but its `key` changes from 9ms to 3ms. If another 6ms higher priority IO arrives, the tolerable time will be negative (-3ms); all IOs with negative tolerable time are rejected with `EBUSY`.

4.2.3 SSD Management (MITTSSD)

Latency variability in SSD is an ongoing problem [27, 37, 99]. Read requests from a tenant can be queued behind writes by other tenants, or the GC implications (more read-write page movements and erases). A 4KB read can be served in 100µs while a write and an erase can take up to 2ms and 6ms, respectively. While there are ongoing efforts to achieve a more stable latency (GC impact

reduction [133, 270] or isolation [140, 161]), none of them cover all possible cases. For example, under write bursts or no idle period, read requests can still be delayed significantly [270, Section 6.6]. Even with isolation, occasional wear-leveling page movements will introduce a significant noise [140, Section 4.3].

Fortunately, not all SSDs are busy at the same time (Section 4.4), a situation that empowers MITTSSD. A read-mostly tenant can set a deadline of $<$1ms; thus, if the read is queued behind writes or erases then the tenant can retry elsewhere.

Resource and deadline checks: There are two initial challenges in building MITTSSD. First, CFQ optimizations are not applicable as SSD parallelizes IO requests without seek costs; the use of noop is suggested [25]. While we cannot reuse MITTCFQ, MITTNOOP is also not reusable. This is because unlike disks where a spindle (a single queue) is the contended resource [60, 193], an SSD is composed of multiple parallel channels and chips. Calculating IO serving time in the block-level layer will be inaccurate (*e.g.*, ten IOs going to ten separate channels do not create queueing delays). Thus, MITTSSD must keep track of outstanding IOs to *every* chip, which is impossible without white-box knowledge of the device (in commodity SSDs, only the firmware has full knowledge of the internal complexity).

Fortunately, host-managed/software-defined flash [216] is gaining popularity and publicly available (*e.g.*, Linux LightNVM [74] on OpenChannel SSDs [18]). Here, all SSD internal channels, chips, physical blocks and pages are all exposed to the host OS, which also manages all SSD managements (FTL, GC, wear leveling, etc.). With this new technology, MITTSSD in the OS layer is possible.

As an additional note, a large IO request can be striped to sub-pages to different channels/chips. If any sub-IO violates the deadline, `EBUSY` is returned for the entire request; all sub-pages are not submitted to the SSD.

Performance: Similar to MITTNOOP's approach, MITTSSD maintains the next available time of *every* chip (as explained below), thus the wait-time calculation is $O(1)$. For every IO, the

overhead is only 300 ns.

Accuracy: Making MITTSSD accurate involves solving two more challenges. First, MITTSSD needs to know the chip-level read/write latency as well as the channel speed, which can be obtained from the vendor's NAND specification or profiling. For measuring chip-level queueing delay, our profiler injects concurrent page reads to a single chip and for channel-level queueing delay, concurrent reads to multiple chips behind the same channel. As a result, for our OpenChannel SSD: $T_{chipNextFree}$+= 100μs per new page read. That is, a page (16KB) read takes 100μs (chip read and channel transfer); $>$16KB multi-page read to a chip is automatically chopped to individual page reads. Thus, $T_{wait} = T_{now} - T_{chipNextFree} + (60\mu s \times \#IO_{SameChannel})$. That is, the IO wait time involves the target chip's next available time plus the number of outstanding IOs to other chips in the same channel, where 60μs is the channel queueing delay (consistent with the 280 MBps channel bandwidth in the vendor specification). If there is an erase, $T_{chipNextFree}$+=$6ms$.

Second, while read latencies are uniform, write latencies (flash programming time) vary across different pages. Pages that are mapped to upper bits of MLC cells incur 2ms programming time, while those mapped to lower bits only incur 1ms. To differentiate upper and lower pages, one-time profiling is sufficient. Our profiled write time of the 512 pages of every NAND block is "`11111121121122...2112`." That is, 1ms write time is needed for pages #0-6, 2ms for page #7, 1ms for pages #8-9, and the middle pages ("...") have a repeating pattern of "`1122`." The pattern is the same for every block (consistent with the vendor specification); hence, the profiled data can be stored in an 512-item array.

To summarize, unlike disks, SSD internal complexity is arguably more complex (in terms of address mapping and latency variability). Thus, accurate prediction of SSD performance requires white-box knowledge of the device.

4.2.4 OS Cache (MITTCACHE)

A user with accesses to a small working set might expect a high cache hit ratio, hence the use of a small deadline. However, under memory space contention, MITTCACHE can inform the application of swapped-out data and and to retry elsewhere (not wait) while the data is fetched from the disk.

Resource and deadline checks: For applications that read OS-cached data via `read(..., deadline)`, MITTCACHE only adds a slight extension. First, MITTCACHE checks if the data is in the buffer cache. If not, it simply propagates the deadline to the underlying IO layer (Section 4.2.1-4.2.3), where if the deadline is less than the smallest possible IO latency (the user expects an in-memory read), `EBUSY` is returned. Else, the IO layer will process the request as explained in previous sections.

The next challenge is for `mmap()`-ed file, which skips the `read()` system call. For example, a database file can be mmap-ed to the heap (`myDB[]`). If some of the pages are not memory resident, an instruction such as "`return myDB[i]`" can stall due to a page fault. Since no system call is involved, the OS cannot signal `EBUSY` easily.

We explored some possible solutions including restartable special threads and `EBUSY` callbacks. In the former, MITTCACHE would restart the page-faulting thread and inform the application's master thread to retry elsewhere. In the latter, the page-faulting thread would still be stalled, but the application's main thread must register a callback to MITTCACHE in order to be notified. These solutions keep the `mmap()` semantic but require heavy restructuring of the application.

We resort to a simpler, practical solution: adding an "`addr check()`" system call. Before dereferencing a pointer to an mmap-ed area, the application can make a quick system call (*e.g.*, `addrcheck(&myDB[i], size, deadline)`, which walks through the process' page table and checks the residency of the corresponding page(s).

Performance: We find `addrcheck()` an acceptable solution for three reasons. First, `mmap()` sometimes is used only for the simplicity of traversing the database file (*e.g.*, traversing B-tree

on-disk pointers in MongoDB), but not necessarily for performance. Second, in storage systems, users typically read a large data range (>1KB); thus, no system call is needed for every byte access. Third, existing buffer cache and page table managements are already efficient; `addrcheck` traverses existing hash tables in $O(1)$. With these reasons combined, using `addrcheck()` only adds a negligible overhead (82ns per call); network latency (0.3ms) still dominates.

One caveat in MITTCACHE is that OS cache should be treated differently than low-level storage. For fairness, MITTCACHE should continue swapping in the data in the background, even after `EBUSY` is already returned. Otherwise, the OS cache is less populated with data from applications that expect memory residency.

Accuracy: As MITTCACHE only looks up the buffer/page tables, there is no accuracy issues. One other caveat to note, MITTCACHE should return `EBUSY` to signal memory space contention (*i.e.*, swapped-in pages are swapped out again), but not for first-time accesses. Note that MITTCACHE does not modify existing page-eviction policies (Section 4.1.3). The OS can be hinted not to swap out the pages that are being checked, to avoid false positives.

4.2.5 Implementation Complexity

MITTOS is implemented in 3440 LOC in Linux v4.10 (MITTNOOP +MITTCFQ, MITTSSD, and MITTCACHE in 1810, 1450, and 130 lines respectively, and an additional 50 lines for propagating deadline SLO through the IO stack).

4.3 Applications

Any application that employs data replication can leverage MITTOS with small modifications. In this paper, we focus on data-parallel storage such as distributed NoSQL.

- **MITTOS-Powered MongoDB:** Our first application is MongoDB; being written in C++, MongoDB enables fast research prototyping of new system calls usage. The following is a series of

our modifications. **(1)** MongoDB can create one deadline for every user, which can be modified anytime. A user can set the deadline value to the 95th-percentile (p95) expected latency of her workload (Section 4.5.2). For example, if the workload mostly hits the disk, the p95 latency can be $>$10ms. In contrast, if the dataset is small and mostly hits the buffer cache, the p95 latency can be $<$0.1ms. **(2)** MongoDB should use MITTOS' `read()` or `addrcheck()` system calls to attach the desired deadline. **(3)** If the first two tries return `EBUSY`, the last retry (currently) disables the deadline ($Prob(3NodesBusy)$ is small; Section 4.4). Having MITTOS return `EBUSY` with wait time, to allow a 4th retry to the least busy node (out of the three), is a possible extension. **(4)** Finally, one last specific change in MongoDB is adding an "exceptionless" retry path. Many systems (Java/C++) use exceptions to catch errors and retry. In MongoDB, C++ exception handling adds 200 µs, thus we must make a direct exceptionless retry path.

The core modification in MongoDB to leverage MITTOS support is only 50 LOC, which mainly involves adding user's deadlines and calling `addrcheck` (MongoDB by default uses `mmap()` to read data file). For testing MITTOS' `read()` interface, we also add `read`-based method to MongoDB in 40 LOC. For making one-hop, exceptionless retry path, we add 20 more lines of code. Finally, to evaluate MITTOS with other advanced techniques, we have added cloning and hedged-request features to MongoDB for another 210 LOC.

• **MITTOS-Powered LevelDB+Riak**: To show that MITTOS is applicable broadly, we also integrated MITTOS to LevelDB. Unlike MongoDB, LevelDB is not a replicated system, it is only a single-machine database engine for a higher-level replicated system such as Riak. Thus, we perform a two-level integration: we first modify LevelDB to use MITTOS system calls, and then the returned `EBUSY` is propagated to Riak where the read failover takes place. All of the modifications are only 50 LOC additions.

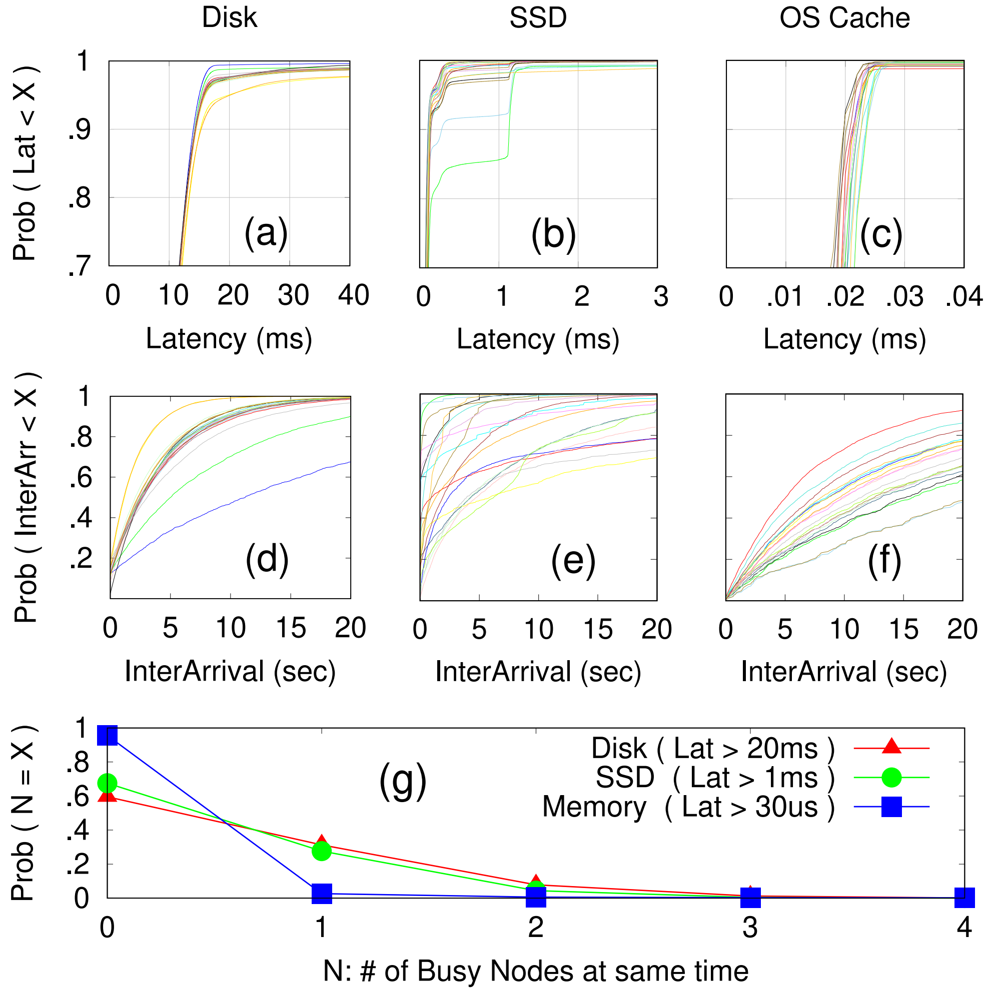

Figure 4.3: **Millisecond-level latency dynamism in EC2.** *The figures are explained in Section 4.4. The 20 lines in Figure (a)-(f) represent the latencies observed in 20 EC2 nodes.*

4.4 Millisecond Dynamism

For our evaluation (Section 4.5), we first present the case of ms-level latency dynamism in multi-tenant storage due to the noisy neighbor problem. To the best of our knowledge, no existing work shows ms-level dynamism at the local resource level; NAS/SAN latency profiles (*e.g.*, for S3/EBS [108, 264]) exist but the deep NAS/SAN stack prevents the study of dynamism at the resource level. Device-level hourly aggregated latency data [130] also exists but prevents ms-level study.

We ran three sets of data collections for disk, SSD, and OS cache in EC2 instances with directly-attached "instance storage," shared by multiple tenants per machine. Many deployments

use local storage for lower latency and higher bandwidth [227]. The instance types are `m3.medium` for SSD and OS cache and `d2.xlarge` for disk experiments (none of "`m`" instances have disk). For each experiment, we spawn **20 nodes** for **8 hours** on a weekday (9am-5pm). For disk, in each node, 4KB data is read randomly every 100ms; for SSD, 4KB data is read every 20ms; for OS cache, we pre-read 3.5GB file (fit in the OS cache) and read 4KB random data every 20ms. Their latencies without noise are expected to be 6-10ms (disk), 0.1ms (SSD), and 0.02ms (OS cache).

We ran more experiments to verify data accuracy. We repeat each experiment 3x on different weekdays (and still obtain highly similar results). To verify the absence of self-inflicted noises, $\geq$20ms `sleep` is used, otherwise "no-sleep" instances will hit VCPU limit and occasionally freeze. An "idle" instance (no data-collection IOs) only reads/writes 10 MB over 8 hours (0.4 KB/s). Finally, the 3.5GB file always fits the OS cache.

Observation #1: *Long tail latencies are consistently observed.* Figures 4.3a-c show the latency CDFs from the disk, SSD, and cache experiments, where each line represents a node (20 lines in each graph). Roughly, tail latencies start to appear around p97 (>20ms) for disks, p97 (>0.5ms) for SSD, and p99 (>0.05ms) for OS cache; the last one implies the cached data is swapped out for other tenants (a VM ballooning effect [255]). The tails can be long, more than 70ms, 2ms, and 1ms at p99 for for disk, SSD, and OS cache, respectively. Sometimes, some nodes are more busy than others (*e.g.*, >0.5ms deviation at p85 and p90 in Figure 4.3b).

A small resource-level variability can easily be amplified by scale. If P fraction of the requests to a node observe tail latencies, and a user request must collect N parallel sub-requests to such nodes, probabilistically, $1-(1-P)^N$ of user requests will observe tail latencies [99]. We will see this scale amplification later (Section 4.5.3).

Observation #2: *Contentions exhibit bursty arrivals.* As a machine can host a wide variety of tenants with different workloads, noise timings are hard to predict. Figures 4.3d-f show the CDF of noise inter-arrival times (one line per node). We define "noisy period", bucketed to windows of 20ms (SSD/cache) or 100ms (disk), when the observed latency is above 20ms, 1ms, and 0.05ms

for disk, SSD, and OS cache, respectively. If noises exhibit a high temporal locality, all lines would have a vertical spike at x=0. However, the figures show that noises come and go at various intervals. Noise inter-arrival distributions also vary across nodes.

Observation #3: *Mostly only 1-2 nodes (out of 20) are busy simultaneously.* This is the most important finding that motivates MITTOS. Figure 4.3 shows the probability of X nodes busy simultaneously (across the 20 nodes), which diminishes rapidly as X increases. For example, only 1 node is busy in 25% of the time and only 2 nodes are busy in 5% of the time. Thus, almost all the time, there are less-busy replicas to failover to.

4.5 Evaluation

We now evaluate MITTCFQ, MITTSSD, and MITTCACHE (with the data set up in the disk, SSD, and OS buffer cache, respectively). We use YCSB [94] to generate 1KB key-value `get()` operations, create a noise injector to emulate noisy neighbors, and deploy 3 MongoDB nodes for microbenchmarks, 20 nodes for macrobenchmarks, and the same number of nodes for the YCSB client nodes. Data is always replicated across 3 nodes; thus, every `get()` request has three choices. For MITTCFQ and MITTCACHE, each node runs on an Emulab d430 machine (two 2.4GHz 8-core E5-2630 Haswell with 64GB DRAM and 1TB SATA disk). For MITTSSD, we only have one machine with an OpenChannel SSD (4GHz 8-core i7-6700K with 32GB DRAM and 2TB OpenChannel SSD with 16 internal channels and 128 flash chips).

All the latency graphs in Figures 4.4 to 4.12 show the latencies obtained from the client `get()` requests. In the graphs, "`NoNoise`" denotes no noisy neighbors, "`Base`" denotes vanilla MongoDB running on vanilla Linux with noise injections, and "`MittOS`" or "`Mitt`" prefix denotes our modified MongoDB running on MITTOS with noise injections. In most of the graphs, even in `NoNoise`, there are tail latencies at p99.8-p100 with a max of 50ms; our further investigation shows three causes: around 0.03% is caused by YCSB (Java) stack, 0.08% by Emulab network contention, and 0.09% by disk being slow (all of which we do not control at this point).

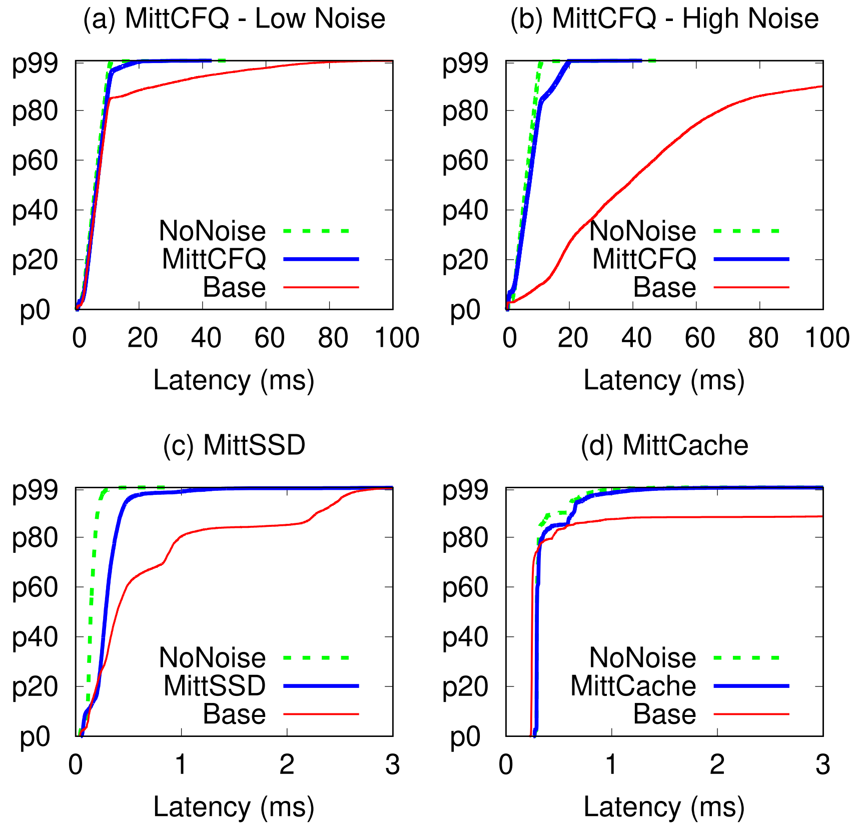

Figure 4.4: **Latency CDFs from microbenchmarks.** *The figures are explained in Section 4.5.1.*

4.5.1 Microbenchmark Results

The goal of the following experiments is to show that MITTOS can successfully detect the contention, return `EBUSY` instantly, and allow MongoDB to failover quickly. We setup a 3-node MongoDB cluster and run our noise injector on one replica node. All `get()` requests are initially directed to the noisy node.

MITTCFQ: Figure 4.4a shows the results for MITTCFQ in three lines. First, without contention (`NoNoise`), almost all `get()` requests can finish in <20ms. Second, with the noise, vanilla MongoDB+Linux (`Base`) experiences the noise impacts (tail latencies starting at p80); the noise injector runs 4 threads of 4KB random reads, but with less priority than MongoDB. Third, still with the same noise but now with MITTCFQ, MongoDB receives `EBUSY` (with a deadline of 20ms) and retries quickly to another replica, cutting tail latencies towards the `NoNoise` line.

Figure 4.4b uses the same setup above but now the noise IOs have a higher priority than MongoDB's. The performance of vanilla MongoDB (`Base`) is severely impacted (a deviation starting at p0). However, MITTCFQ detects that the in-queue MongoDB's IOs are often less picked than the new high-priority IOs, hence quickly notifying MongoDB of the disk busyness.

MITTSSD: Figure 4.4c shows the results for MITTSSD (note that we use our lab machine for this one with a local client). First, SSD can serve the requests in $<$0.2ms (`NoNoise`). Second, when read IOs are queued behind write IOs (the noise), the latency variance is high (`Base`); the noise injector runs a thread of 64KB writes. Third, with MITTSSD, MongoDB instantly reroutes the IOs that cannot be served in 2ms (the small gap between `Base` and `MittSSD` lines is the cost of software failover).

MITTCACHE: Figure 4.4d shows the results for MITTCACHE. First, in-memory read can be done in µs, but the network hop takes 0.3ms (`NoNoise`). Second, as we throw away about 20% of the cached data (with `posix_fadvise`), some memory accesses trigger page faults and must wait for disk access (tail latencies at p80). Finally, with MITTCACHE, MongoDB can first check whether the to-be-accessed memory region is in the cache and rapidly retry elsewhere if the data is not in the cache, removing the 20% long tail.

4.5.2 MITTCFQ *Results with EC2 Noise*

Our next goal is to show the potential benefit of MITTOS in a real multi-tenant cluster. We note that MITTOS is targeted for deployment at the host OS (and VMM) level for full visibility of resource queues. For this reason, we do not run experiments on EC2 as there is no access to the host OS level (running MITTOS as a guest OS will not be effective as MITTOS cannot observe the contention from other VMs). Instead, to mimic a multi-tenant cluster, in this evaluation section, we apply EC2 noise distributions (Section 4.4) to our testbed. Later (Section 4.5.8), we will also inject noises with macrobenchmarks and production workloads.

Methodology: We deploy a 20-node MongoDB disk-based cluster, with 20 concurrent YCSB clients sending `get()` requests across all the nodes. For the noise, we take a 5-minute timeslice from the EC2 disk latency distribution across the 20 nodes (Figure 4.3a). We run a multi-threaded *noise injector* (in every node) whose job is to emulate busy neighbors at the right timing. For example, if in node n at time t, the EC2 data shows a 30ms latency (while no noise is around 6ms), then the noise injector will add IO noises that will make the disk busy for 24ms (*e.g.*, by injecting two concurrent 1MB reads, where each will add 12ms delay).

Other techniques compared: Figure 4.5a shows the comparisons of MITTCFQ with other techniques such as hedged requests, cloning, and application timeout. **Base:** As usual, we first run vanilla MongoDB+Linux under a typical noise condition; we will use *13ms*, the *p95* latency (Figure 4.5a) for deadline and timeout values below. **Hedged requests:** This is a strategy where a secondary request is sent after "the first request [try] has been outstanding for more than the 95th-percentile expected latency, [which] limits the additional load to approximately 5% while substantially shortening the latency tail" [99]. More specifically, if the first try does not return in 13ms, MongoDB will make a 2nd try to another replica and take the first one to complete (the first try is *not* cancelled). **Cloning:** Here, for every user request, MongoDB duplicates the request to two random replica nodes (out of three choices) and picks the first response. **Application timeout (TO):** Here, if the first try does not finish in 13ms, MongoDB will cancel the first try and make a second try, and so on. With MITTCFQ and application timeout, the third try (rare, as alluded in Figure 4.3g) disables the timeout; otherwise, users can undesirably get IO errors.

Results: We discuss Figure 4.5a from right-to left-most lines. First, as expected, `Base` suffers from long tail latencies (>40ms at p98), as occasionally the requests are "unlucky" and hit a busy replica node.

Second, application timeout (`AppTO` line) must *wait* at least 13ms delay before reacting. The 2nd try will take at least another few ms for a disk read, hence `AppTO` still exhibits around >20ms tail latencies above p95.

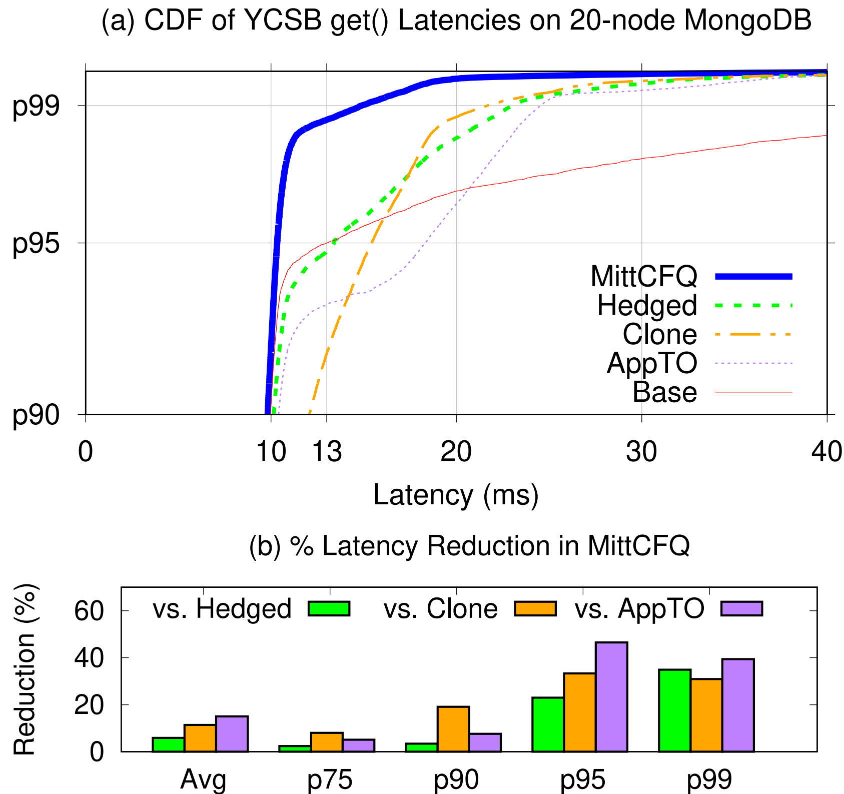

Figure 4.5: **MITTCFQ results with EC2 noise.** *The figures are explained in Section 4.5.2.*

Third, cloning is better than timeout (`Clone` vs. `AppTO`) but only above p95. This is because cloning can pick the faster of the two concurrent requests. However, below p93 to p0, cloning is worse. This is because cloning increases the load by 2x, hence creating a self-inflicting noise in common cases.

Fourth, hedged strategy proves to be effective. It does not significantly increase the load (below p95, `Hedged` and `Base` are similar), but it effectively cuts the long tail (the wide horizontal gap between `Hedged` and `Base` lines above p95). However, we can still observe that hedged's additional load slightly delays other requests (`Hedged` is slightly worse than `Base` between p92 and p95).

Finally, MITTCFQ *is shown to be more effective.* Our most fundamental principle is that the first try does *not* need to wait if the OS cannot serve the deadline. As a result, there is a significant latency reduction above p95. To quantify our improvements, the bar graph in Figure 4.5b shows (at specific percentiles) the % of latency reduction that MITTCFQ achieved compared

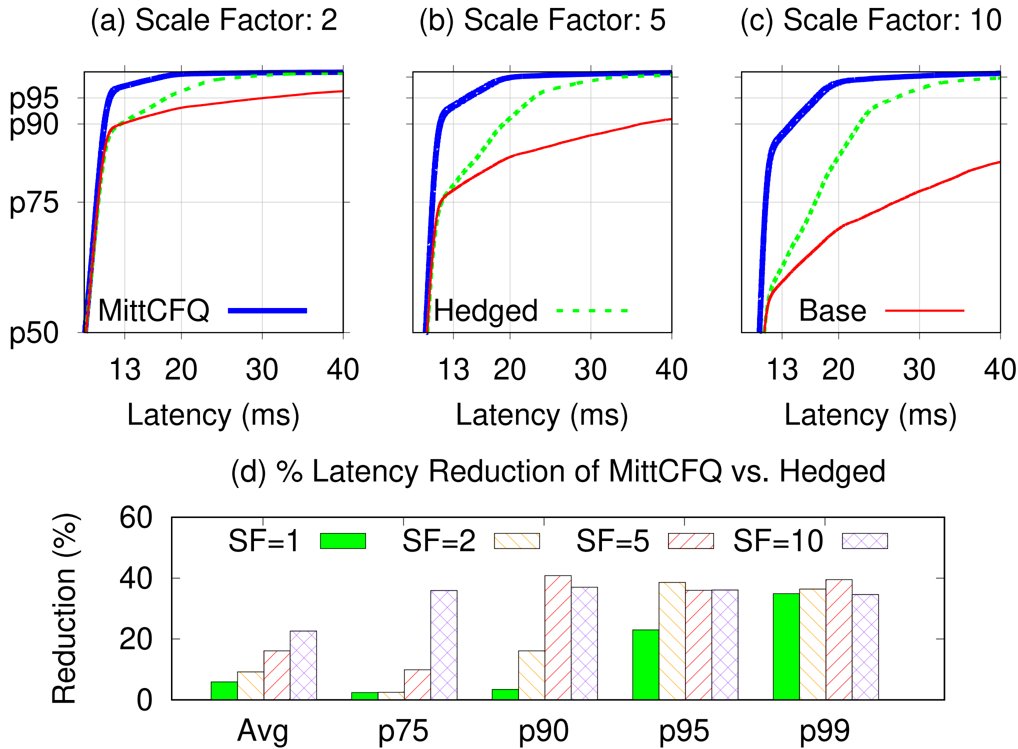

Figure 4.6: **Tail amplified by scale (MITTCFQ vs. Hedged).** *The figures are explained in Section 4.5.3.*

to the other techniques.[2] For example, at p95, MITTCFQ reduces the latency of `Hedged`, `Clone`, and `AppTO` by 23%, 33%, and 47%, respectively. There is also a pattern that the higher the percentiles, MITTCFQ's latency reductions are more significant.

4.5.3 *Tail Amplified by Scale (MITTCFQ)*

The previous section only measures the latency of every individual IO, which reflects the "component-level variability" in every disk/node. In other words, so far, a user request is essentially a single `get()` request. However, component-level variability can be amplified by scale [99]. For example, a user request might need to fetch N data items by submitting S parallel `get()` requests and then must wait until all the S items are fetched.

To show latency tail amplified by scale, we introduce "SF," a scale factor of the parallel re-

2. *% of Latency Reduction* = $(T_{Other} - T_{MittCFQ})$ / T_{Other}

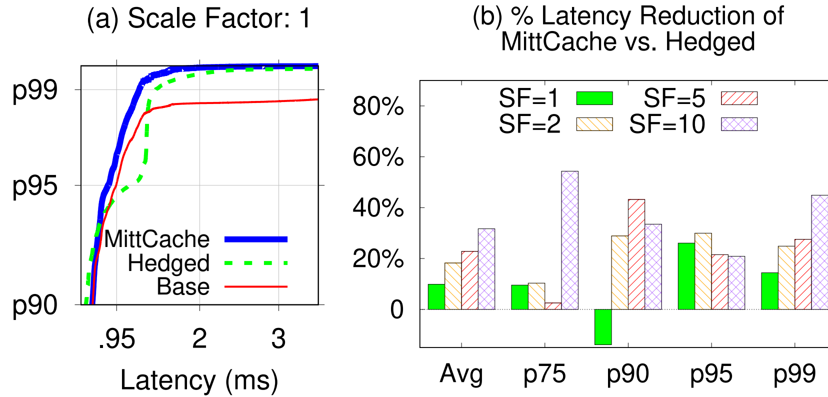

Figure 4.7: **MITTCACHE vs. Hedged.** *The figures are explained in Section 4.5.4. The left figure shows the same CDF plot as in Figure 4.5a and the right figure the same %reduction bar plot as in Figure 4.6b.*

quests; for example, SF=5 means a user request is composed of 5 parallel `get()` requests to different nodes. Figure 4.5a in the previous section essentially uses SF=1. Figures 4.6a-c show the same figures as in Figure 4.5a, but now with scaling factors of 2, 5, and 10. Since, hedged requests are more optimum than cloning or application timeout, we only compare MITTCFQ with `Hedged`. We make two important observations from Figure 4.6.

First, again, hedged requests must wait before reacting. If before, with SF=1 (Figure 4.5a), there are 5% (p95) of requests that must wait for 13ms, now with SF=2, there are roughly 10% (p90) of requests that cannot finish by 13ms (the `Hedged` line in Figure 4.6a). Similarly, with SF=5 and 10, there are around 25% (p75) and 40% (p60) of requests that must wait for 13ms, as shown in Figures 4.6b-c.

Second, MITTCFQ initially already cuts the tail latencies of the individual IOs significantly; with SF=1, only 1% (p99) of the IOs are above 13ms (Figure 4.5a). Thus, with scaling factors of 5 and 10, only 5% and 10% of IOs are above 13ms (`MittCFQ` lines in Figures 4.6b-c).

The bar graph in Figure 4.6d summarizes the % of latency reduction achieved by MITTCFQ from the `Hedged` strategy across the three scaling factors. With SF=10, MITTCFQ already reduces the latency by 36% starting at *p75*. Thus, MITTCFQ reduces the overall average latency of `Hedged` by 23% (x=`Avg`, SF=10 bar); note that "average" denotes the average latencies of *all* IOs (not

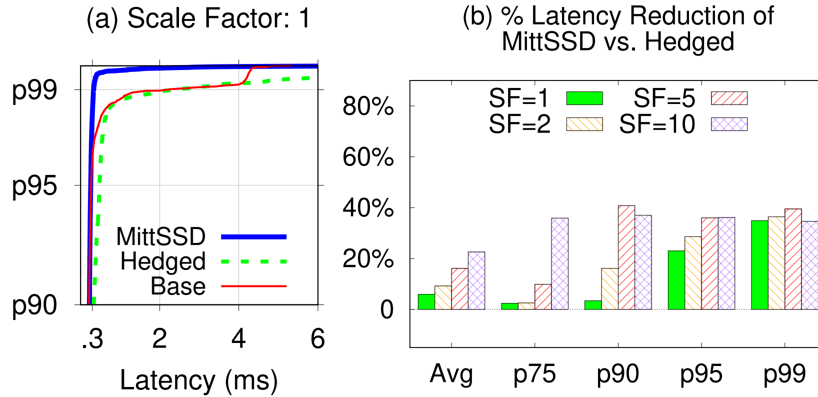

Figure 4.8: **MITTSSD vs. Hedged.** *The figures are explained in Section 4.5.5. The left figure shows the same plot as in Figure 4.5a and the right figure the same %reduction bar plot as in Figure 4.6b.*

just the high-percentile ones).

4.5.4 MITTCACHE *Results with EC2 Noise*

Similarly, Figure 4.7 shows the success of MITTCACHE (20-node results). All the data were originally in memory, but we swapped out $P\%$ of the cached data, where P is based on the cache-miss rate in Figure 4.3c. Another method to inject cache misses is by running competing IO workloads, but such setup is harder to control in terms of achieving a precise cache miss rate. For this reason, we perform manual swapping. In terms of the deadline, we use a small value such that `addrcheck` returns `EBUSY` when the data is not cached. In Figure 4.7b, at p90 and SF=1, our reduction is negative; our investigation shows that this is from the uncontrollable network latency (which dominates the request completion time). Similarly, in Figure 4.7a, between p95 and p98, `Hedged` is worse than `Base` due to the same networking reason.

4.5.5 MITTSSD *Results with EC2 Noise*

Unlike in prior experiments where we use 20 nodes, for MITTSSD, we can only use our single OpenChannel SSD in one machine with 8 core-threads. We carefully (a) partition the SSD

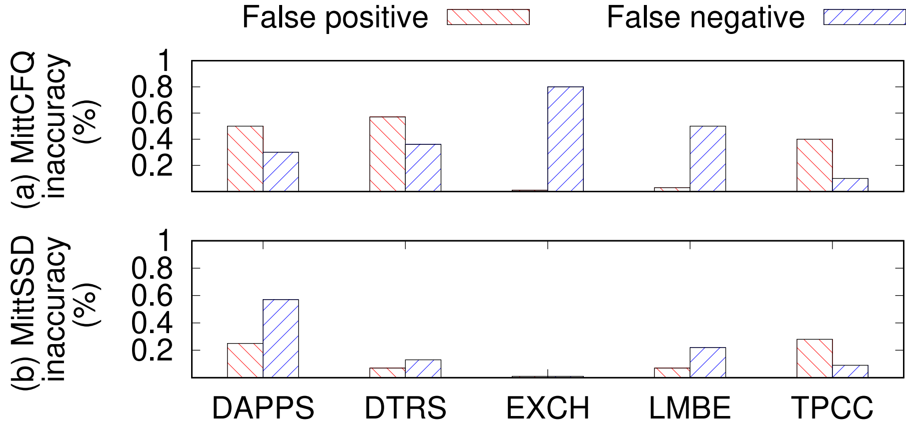

Figure 4.9: **Prediction inaccuracy.** *(As explained in Section 4.5.6).*

into 6 partitions with no overlapping channels, hence no contention across partitions, (b) set up 6 MongoDB nodes/processes on a single machine serving only 6 concurrent client requests, each mounted on one partition, (c) pick noise distributions only from 6 nodes in Figure 4.3b, and (d) set the deadline to the p95 value, which is 0.3ms (as there is no network hop).

While latency is improved with MITTOS (the gap between `MittSSD` and `Base` in Figure 4.8a), we surprisingly found that hedge (`Hedged` line) is worse than the baseline. After debugging, we found another limitation of hedge (in MongoDB architecture). In MongoDB, the server creates a request handler for every user, thus 18 threads are created (for 6 clients connecting to 3 replicas). In stable state, only 6 threads are busy all the time. But for 5% of the requests (after the timeout expires), the workload intensity doubles, making 12 threads busy simultaneously (note that SSD is fast, thus processes are not IO bound). These hedge-induced CPU contentions (12 threads on a 8-thread machine) cause the long tail. Figure 4.8b shows the resulting % of latency reduction.

4.5.6 Prediction Accuracy

Figure 4.9 shows the results of MITTCFQ and MITTSSD accuracy tests. For a more thorough evaluation, we use 5 real-world block-level traces from Microsoft Windows Servers (the details are publicly available [157, Section III][21]), choose the busiest 5 minutes, and replay them on just one machine. For a fairer experiment, as the traces were disk-based, we re-rate the trace 128x more

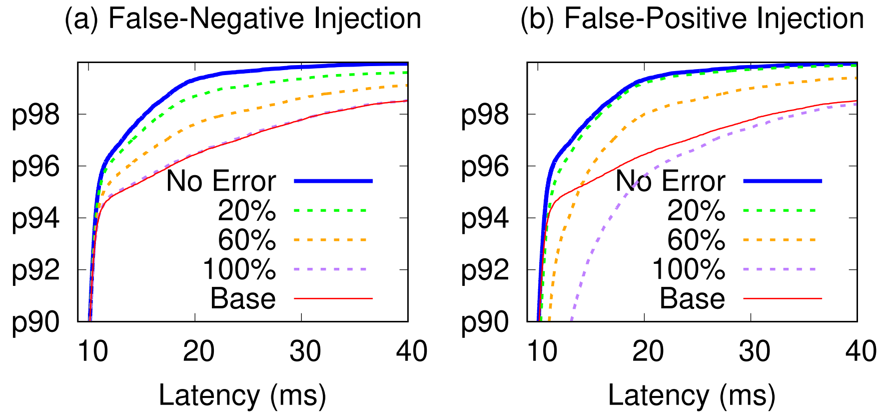

Figure 4.10: **Tail sensitivity to prediction error.** *The figures are described in Section 4.5.7.*

intensive (128 chips) for SSD tests. For each trace, we always use the p95 value for the deadline.

The % of inaccuracy includes: false positives (EBUSY is returned, but $T_{processActual} \leq T_{deadline}$) and false negatives (EBUSY is not returned, but $T_{processActual} > T_{deadline}$). During accuracy tests, EBUSY is actually *not* returned; if error is returned, the IO is not submitted to the device, hence the actual IO completion time cannot be measured, which is also the reason why we cannot report accuracy numbers in real experiments. Instead, we attach EBUSY flag to the IO descriptor, thus upon IO completion, the accuracy can be measured.

Figure 4.9 shows the % of false positives and negatives over all IOs. In total, MittCFQ inaccuracy is only 0.5-0.9%. Without our precision improvements (Section 4.2.2), its inaccuracy can be as high as 47%. MittSSD inaccuracy is also only up to 0.8%. Without the improvements (Section 4.2.3), its inaccuracy can rise up to 6% (no hard-to-predict disk seek time). The next question is how far our predictions are off *within* the inaccurate IO population. We found that all the "diff"s are <3ms and <1ms on average, for disk and SSD respectively. We leave further optimizations as future work.

4.5.7 Tail Sensitivity to Prediction Error

High prediction accuracy depends on detailed and complex device performance model. This raises the question whether a simpler model (but lower accuracy) can also be effective. To investigate this, we use the same MITTCFQ experiment in Section 4.5.2 but now we vary MITTCFQ's prediction accuracy by injecting false-negative and false-positive errors: **(a)** False-negative injection implies that when MITTOS decides to cancel an IO and return `EBUSY`, it has $E\%$ chance to let the IO continue and not return `EBUSY`. Figure 4.10a shows the latency implications when the false-negative rate is varied (E=20-100%). "No error" denotes the current MITTCFQ's accuracy and 100% false negative reflects the absence of MITTOS (*i.e.*, similar to `Base`). **(b)** False-positive injection implies that when IO actually can meet the deadline, MITTOS instead will return `EBUSY` at $E\%$ rate. Figure 4.10b shows the latency results with varying false-positive rates.

Overall, both figures show that higher accuracy leads to shorter latency tail. False-negative errors only affect slow requests, thus extreme error rates in this category will not make MITTOS worse than `Base`. On the other hand, false-positive errors matter more as they will trigger unnecessary failovers. For example, in Figure 4.10b with 100% false-positive rate, all IOs will be retried, creating worse latency tail than `Base`.

4.5.8 Other evaluations

Workload Mix

We ran experiments that colocate MITTOS + MongoDB with filebench and Hadoop Facebook workloads [88]. We deployed filebench's fileserver, varmail, and webserver macrobenchmarks on different nodes (creating different levels of noise) and the first 50 Hadoop jobs from the Facebook 2010 benchmark [88]. Figure 4.11a shows the resulting performance of MITTCFQ, `Hedged` and `Base`. The `Base` line shows that almost 15% of the IOs experience long tail latencies (x>40ms above p85). `Hedged` shortens the latency tail, but MITTCFQ is still more effective.

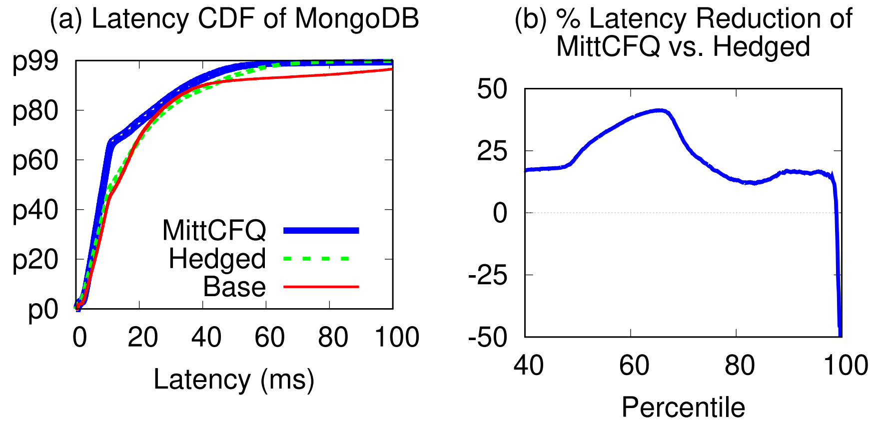

Figure 4.11: **MITTCFQ with macrobenchmarks and production workloads.** *The figures are discussed in Section 4.5.8.*

The y-axis of Figure 4.11b shows the % of latency reduction achieved by MITTCFQ compared to `Hedged` at every percentile (*i.e.*, an interpose layout of Figure 4.11a). The reduction is positive in overall (up to 41%), but above p99, `Hedged` is faster. This is because the intensive workloads make our MongoDB perform 3rd retries (with deadline disabled) in 1% of the IOs, but the 3rd choices were busier than the first two. On the other hand, in the `Hedged` case, it only tries to the 2nd replica, and in this 1% case, the 2nd choices were less busy than the 3rd ones. This problem can be addressed by extending MITTOS interface to return the expected wait time, with which MongoDB can choose the shortest wait time when all replicas return `EBUSY`.

vs. Tied Requests

One evaluation that we could not fully perform is the comparison with the "tied requests" approach [99, pg77]. In this approach, a user request is cloned to another server with a small delay and both requests are tagged with the identity of the other server ("tied"). When one of them "*begins execution*," it sends a cancellation message to the other one. This approach was found very effective for a cluster-level distributed file system [99].

We attempted to build a similar mechanism, however MongoDB does not manage any IO queues (unlike the distributed file system mentioned above). All incoming requests are submitted

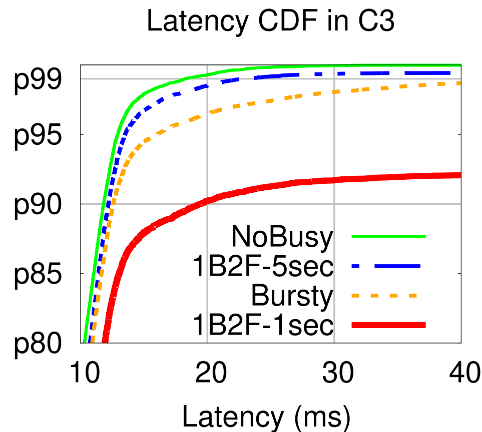

Figure 4.12: **C3 and bursty noises.** *The figure is described in Section 4.5.8.*

directly to the OS and subsequently to the device queue, which raises three complexities. First, we found that most requests do not "linger" in the OS-level queues (Section 4.2.1-4.2.2); instead, the device quickly absorbs and enqueues them in the device queue. As an implication, it is not easy to know when precisely an IO is served by the device (*i.e.*, the "begin execution" time). Device queue is in fact "invisible" to the OS; MITTOS can only record their basic information (size, offset, predicted wait time, etc.). Finally, it is not easy to build a "begin-execution" signal path from the OS/device layer to the application; such signal cannot be returned using the normal IO completion/`EBUSY` path, thus a callback must be built and the application must register a callback. For these reasons, we did not complete the evaluation with tied requests.

vs. Snitching/Adaptivity

Many distributed storage systems employ a "choose-the-fastest-replica" feature. This section shows that such existing feature is not effective in dealing with millisecond dynamism. Specifically, we evaluated Cassandra's snitching [3] and C3's adaptive replica selection [244] mechanisms. As C3 improves upon Cassandra [244], we only show C3 results for graph clarity. Figure 4.12 shows that under sub-second burstiness, unlike MITTOS, neither Cassandra nor C3 can react to the bursty noise (the gap between `NoBusy` and `Bursty` lines). When we create another scenario

where one replica/disk is extremely busy (1B) and two are free (2F) in 1-second rotating manner, the tail latencies become worse (the `1B2F-1sec` line is below p90). We then decrease the noise rotating frequency and found that they only perform well if the busyness is stable (a busy rotation in every 5 seconds), as shown in the `1B2F-5sec` line.

MITTOS-powered LevelDB+Riak

Figure 4.13 shows the result of MITTOS integration to LevelDB+Riak (Section 4.3). For this experiment, we primarily evaluate MITTCFQ with disk-based IOs and use the same EC2 disk noise distribution. Figure 4.13a shows the resulting latency CDF (similar to earlier experiments), showing that MITTCFQ can also help LevelDB+Riak to cut the latency tail. Figure 4.13b depicts the situation over time from the perspective of a single node. In this node, when the noise is high (the high number of outstanding IOs make the deadline cannot be met), MITTOS in this node will return `EBUSY` (the triangle points). But when the noise does not break the deadline, `EBUSY` is not returned.

All in One

Finally, we enable MITTCFQ, MITTSSD, and MITTCACHE in one MongoDB deployment with 3 users whose data are mostly in the disk, SSD (flash cache), and OS cache, with three different deadlines, 20 ms, 2 ms, and 0.1 ms for the three users, respectively. The SSD is mounted as a flash cache (with Linux `bcache`) between the OS cache and the disk, thus our MongoDB still runs on one partition. On one replica node, we simultaneously injected three different noises, disk contentions, SSD background writes, and page swapouts. Put simply, we combine the earlier microbenchmarks (Section 4.5.1) into a single deployment. We obtained results similar to Figure 4.4, showing that all MITTOS resource managements can co-exist.

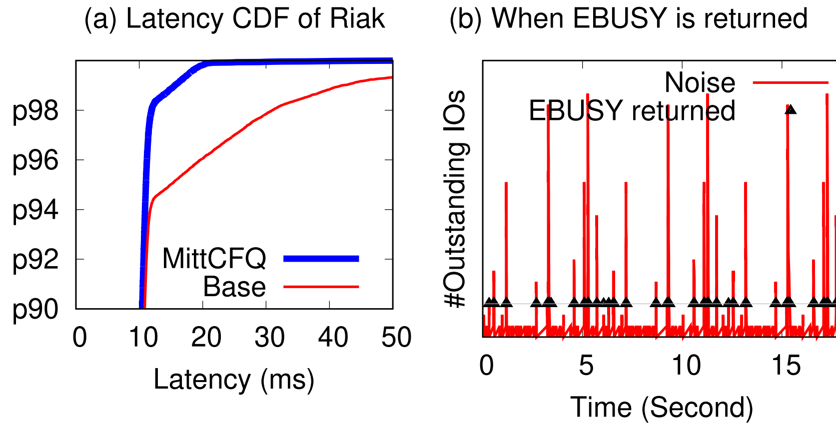

Figure 4.13: **MITTOS-powered Riak+LevelDB.** *The figure is explained in Section 4.5.8.*

Writes Latencies

Our work only addresses read tail latencies for the following reasons. In many storage frameworks (MongoDB, Cassandra, etc.), writes are first buffered to memory and flushed in the background, thus user-facing write latencies are not directly affected by drive-level contention. Even if the application flushes writes, most modern drives employ (capacitor-backed) NVRAM to absorb writes quickly and persistently. We ran YCSB write-only workloads with disk noise and found that the `Base` and `NoNoise` latency lines are very close to each other.

4.6 Discussions

4.6.1 MITTOS *Limitations*

We identify two fundamental MITTOS limitations: **(1)** First, besides hardware/resource-level queueing delays, the software stack can also induce tail latencies. For example, an IO path can traverse a rare code path that triggers a long lock contention or inefficient loops. Such corner-case paths are hard to foresee. **(2)** Second, while rare, hardware performance can degrade over time due to many factors [103, 122, 123], or the other way around, performance can improve as device wears out (*e.g.*, faster SLC programming time as gate oxide weakens [118]). This suggests that

latency profiles must be recollected over time; a sampling runtime method can be used to catch a significant deviation.

In terms of design and implementation, our current version has the following limitations (which can be extended in the future): **(1)** Currently, applications only pass deadline latencies to MITTOS. Other forms of SLO information such as throughput [248] or statistical latency distribution [184] can be included as input to MITTOS. Furthermore, applications must set precise deadline values, which could be a major burden. Automating the setup of deadline/SLO values in general is an open research problem [154]. For example, too many `EBUSY`s imply that the deadline is too strict, but rare `EBUSY`s and longer tail latencies imply that the deadline is too relaxed. The open challenge is to find a "sweet spot" in between, which we leave for future work. **(2)** MITTOS essentially returns a binary information (`EBUSY` or success). However, applications can benefit from richer responses, for example, predicted wait time (Section 4.5.8) or certainty/confidence of how close to or far from the deadline the prediction is. **(3)** Our performance models require white-box knowledge of the devices and resources queueing policies. However, many commodity disks and SSDs do not expose their complex firmware logic. Simpler and more generic device models can be explored, albeit with higher prediction errors (Section 4.5.7).

4.6.2 Beyond the Storage Stack

We believe that MITTOS principles are powerful and can be applied to many other resource managements such as CPU, runtime memory, and SMR drive managements.

In EC2, CPU-intensive VMs can contend with each other. The VMM by default sets a VM's CPU timeslice to 30ms, thus user requests to a frozen VM will be parked in the VMM for tens of ms [267]. With MITTOS, the user can pass a deadline through the network stack, and when the message is received by the VMM, it can reject the message with `EBUSY` if the target VM must still sleep more than the deadline time.

In Java, a simple "`x = new Request()`" can stall for *seconds* if it triggers GC. Worse, *all* threads

on the same runtime must stall. There are ongoing efforts to reduce the delay [196, 211], but we find that the stall cannot be completely eliminated; in the last 3 months, we study the implementations of many Java GC algorithms and find that EBUSY exception cannot be easily thrown for the GC-triggering thread. MITTOS has the potential to transform future runtime memory management.

Similar to GC activities in SSDs, SMR disk drives must perform "band cleaning" operations [110], which can easily induce tail latencies to applications such as SMR-backed key-value stores [197, 223]. MITTOS can be applied naturally in this context, also empowered by the development of SMR-aware OS/file systems [56].

4.6.3 Other Discussions

With MITTOS*, should other tail-tolerant approaches be used?* We believe MITTOS handles a major source of storage tail latencies (*i.e.*, storage device contention). Ideally, MITTOS is applied to all major resources as discussed above. If MITTOS is only applied to a subset of the resources (*e.g.*, storage stack only), then other approaches such as hedged/tied requests are still needed and can co-exist with MITTOS.

Can MITTOS*' fast replica switching cause inconsistencies?* MITTOS encourages fast failover, however many NoSQL systems support eventually consistency and generally attempt to minimize replica switching to ensure monotonic reads. MITTOS-powered NoSQL can be made more conservative abo-ut switching replicas that may lead to inconsistencies (*e.g.*, do not failover until the other replicas are no longer stale).

Can MITTOS *expose side channels?* Some recent works [147, 283] show that attackers can use information exposed by the OS (*e.g.*, CPU scheduling [147], cache behavior [282]). MITTOS can potentially make IO side channels more effective because an attacker can obtain a less noisy signal about the I/O activity of other tenants. We believe that even without MITTOS, attackers can deconstruct the noises by probing IO performance periodically, however more research can be

conducted in this context.

4.7 Related Work

Storage tails: A growing number of work has investigated many root causes of storage latency tail, including multi tenancy [140, 161, 200, 248], maintenance jobs [60, 76, 193], inefficient policies [134, 192, 272], device cleaning/garbage collection [55, 99, 140, 161, 270], and hardware variability [130]. MITTOS does not eliminate these root causes but rather expose the implied busyness to applications.

Storage tail tolerance: Throughout the paper, we discussed solutions such as snitching/adaptivity [3, 244], cloning at various different levels [62, 264], hedged and tied requests [99].

Performance isolation (QoS): A key to reduce performance variability is performance isolation, such as isolation of CPU [281], IO throughput [121, 239, 248], buffer cache [200], and end-to-end resources [65, 89]. QoS-enforcements do not return busy errors when SLOs are not met; they provide "best-effort fairness." MITTOS is *orthogonal* to this class of work (Section 4.1.3).

OS transparency: MITTOS in spirit is similar to other works that advocate more information exposure to applications [67] and first-class supports for interactive applications [273]. MITTOS provides busyness transparency by slightly modifying the user-kernel interfaces (mainly for passing deadlines and returning `EBUSY`).

4.8 Conclusion

Existing application-level tail-tolerant solutions can only gu-ess at resource busyness. We propose a new philosophy: OS-level SLO awareness and transparency of resource busyness, which eases applications' tail and other performance management. In a world where consolidation and sharing are a fundamental reality, busyness transparency, as embodied in the MITTOS principles, should only grow in importance.

4.9 Appendix: Details

This section describes how we compute $T_{nextFree}$ (`freeTime`) as discussed inSection 4.2.1-4.2.2. For the noop scheduler (Section 4.2.1), we model the disk queue `Q` as a list of outstanding IOs {`ABCD...`}. Thus, the `Q`'s wait time (`qTime`) is the total *seek cost* of every consecutive IO pairs in the queue, which will be added to `freeTime`:

```
qTime += (seekCost(A,B) + seekCost(B,C) + ...);
freeTime += qTime;
```

We then model the `seekCost` from IO `X` to `Y` as follows:

```
seekCost(X,Y) =
  seekCostPerGB * (Y.offsetInGB - X.offsetInGB) +
  transferCostPerKB * X.sizeInKB;
```

The `transferCostPerKB` is currently simplified as a constant parameter. The `seekCostPerGB` parameter is more challenging to set due to the complexity of disk geometry. To profile a target disk, we measure the latency (seek cost) of all pairs of random IOs per GB distance. For example, for a 1000 GB disk, we fill a matrix `seekCostPerGB[X][Y]`, where `X` and `Y` range from 1 to 1000, a total of 1 million items in the matrix. We profile the disk with 10 tries and use linear regression for more accuracy.

The next challenge is to model the queueing policy. For noop (FIFO) scheduler, the order of IOs do not change. However, as the IOs are submitted to the disk, the disk has its own disk queue and will reorder the IOs. Existing works already describe how to characterize disk policies [229, 231]. For example, we found that our target disk exhibits SSTF policy. Thus, to make `freeTime` more accurate, `qTime` should be modeled into an SSTF ordering (`sstfTime`), for example if the disk head position (known from the last IO completed) is currently near `D`, then the `freeTime` should be:

```
sstfTime += (seekCost(D,C) + seekCost(C,B) + ...);
freeTime += sstfTime;
```

For CFQ, the modeling is more complex as there are two-level of queues: the CFQ queues (based on priority and fairness) and the disk queue (SSTF). Thus, we predict two queueing wait times `cfqTime` and `sstfTime`. Predicting `cfqTime` is explained in Section 4.2.2.

CHAPTER 5

LINNOS: PREDICTABILITY ON UNPREDICTABLE FLASH STORAGE WITH A LIGHT NEURAL NETWORK

LinnOS is an operating system that leverages a light neural network for inferring SSD performance at a very fine—per-IO—granularity and helps parallel storage applications achieve performance predictability. LinnOS supports black-box devices and real production traces without requiring any extra input from users, while outperforming industrial mechanisms and other approaches. Our evaluation shows that, compared to hedging and heuristic-based methods, LinnOS improves the average I/O latencies by 9.6-79.6% with 87-97% inference accuracy and 4-6µs inference overhead for each I/O, demonstrating that it is possible to incorporate machine learning inside operating systems for real-time decision-making.

The biggest challenge for LinnOS is to be as effective and fine-grained as the popular approach, speculative execution, which can mitigate *every* slow I/O by sending a duplicate I/O to another node or device. Speculative execution's success in increasing predictability comes at the cost of poor resource utilization. The key to avoiding this cost is to know the current activities going on inside the devices and always schedule I/Os to less occupied devices. However, because keeping the abstraction barrier is a fundamental constraint, we need to learn to infer latency and make the inference highly usable. Achieving this requires *learning and inferring on a very fine, per-I/O scale* in a live fashion. To the best of our knowledge, there is no existing learning approach for I/O scheduling that supports such fine-grained learning due to the challenges of achieving per-I/O accuracy and fast online inference. To address this, LinnOS introduces three technical contributions.

First, LinnOS converts the hard latency inference problem into a simple *binary* inference ("fast" or "slow" speed). We take advantage of the typical latency distributions in system deployments, specifically, a behavior that forms a Pareto distribution with a high alpha number. In other words, most of the time (*e.g.*, $>$90%), the latency is very stable, but occasionally (*e.g.*, $<$10% of the

time), the latency exhibits a long-tail behavior [89, 99, 183, 206]. The behavior of flash storage reflects the same distribution [81, 130]. In this simple view where users only want "slow" I/Os to become "fast," inferring the exact latencies is overkill. With this intuition, LinnOS comes with an algorithm that monitors the latency distribution of the current workload running on the flash device and computes a roughly optimal threshold that separates the slow and fast speed ranges.

Second, with the binary model, LinnOS employs a simple admission control for clustered storage applications. LinnOS makes a binary inference on every incoming I/O using a light neural network model that infers the I/O speed in advance without any guide from the device nor application. If the I/O is inferred to be fast, LinnOS will submit it down to the flash device, or else it will revoke the I/O and inform the clustered application. With this timely and straightforward binary information, the storage application can quickly failover the I/O to another node or device that holds the same replica. Furthermore, resources are efficiently utilized because the original slow I/O has been revoked.

Third, LinnOS balances the accuracy and performance of the neural network. High accuracy but high inference time will lead to a significant per-I/O overhead, especially for modern SSDs. On the other hand, lowering inference time by lowering accuracy will lead to many false inferences that make storage performance hard to reason about.

For high accuracy, LinnOS profiles the latency of millions of I/Os submitted to the device (a natural "data lake"), which will be used to train the neural network. Furthermore, as we convert regression to a simple binary classification, the output accuracy is significantly improved (akin to the simplicity of "cat or dog" image classification). The next challenge is to decide the input features that matter most to improving accuracy. We will present our surprising findings. For example, "important-looking" features such as block offsets, read/write flags, or long history of writes do not play a significant role. In the end, the input features become tractable with only two types of information: the latencies of a few recently completed I/Os and the number of pending I/Os when those I/Os and the current, to-be-inferred I/O arrived.

For performance, the challenge is to make an inference (admission decision) in sub-10µs, which is crucial as we target fine-grained live inference for fast storage devices. While using deeper models with more features can improve accuracy, it will hurt inference latency and will be deemed too expensive for usage in the I/O layer. Through several design iterations, we cut the inference time to 4-6µs with minor accuracy loss, achieved with several methods: a 3-layer light neural network, weight quantization, and (optional) 2-threaded/2-core matrix multiplication.

Our evaluation shows that LinnOS supports a wide variety of black-box devices (10 device models tested) and works on real production traces without requiring any extra input from users (*e.g.*, hints about traces/devices or latency deadlines etc.), outperforms industrial approaches such as pure hedging, and beats simple and "advanced" heuristics that we design. Compared to these methods, LinnOS, complemented by hedging based on the learning outcome, further improves the average I/O latencies by 9.6-79.6% with 87-97% accuracy and only 4-6µs inference overhead for every I/O.

Overall, we show that it is plausible to adopt machine learning methods for operating systems to learn black-box devices. We conclude with many interesting discussions to explore in the future.

5.1 Overview

We now give the overview of LinnOS, its usage scenario, architecture, and challenges, followed by its design (Section 5.2).

5.1.1 Usage Scenario

LinnOS is beneficial for parallel, redundant storage such as flash arrays (cluster-based or RAID) that maintain multiple replicas of the same block, as illustrated in Figure 5.1. **(a)** With LinnOS, when a storage application performs an I/O via OS system calls, it can add a one-bit flag, hinting to LinnOS that the I/O is latency-critical (`LC=true`), *e.g.*, for interactive services. Such tagging of critical operations has been proposed many times [273, 280], but in our case, the bit is used

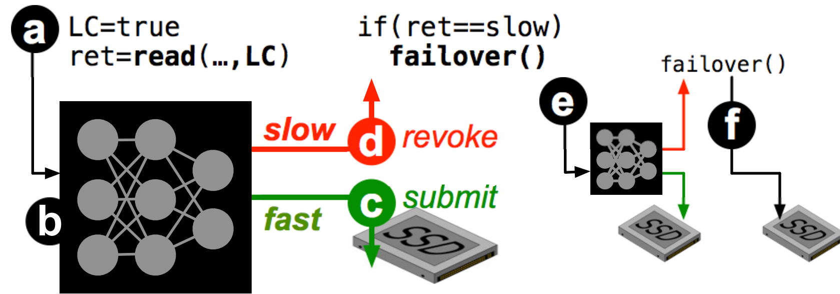

Figure 5.1: **Usage scenario.** *This usage scenario is explained in Section 5.1.1. "LC" implies latency critical.*

to trigger LinnOS to infer the I/O latency. **(b)** Before submitting the I/O to the underlying SSD, LinnOS inputs the I/O information to the neural network model that it has trained, which will make a binary inference: fast or slow. **(c)** If the output is "fast," LinnOS *submits* the I/O down to the device. **(d)** Otherwise, if it is "slow," LinnOS *revokes* the I/O (not entered to the device queue) and returns a "`slow`" error code. **(e)** Upon receiving the error code, the storage application can failover the same I/O to another replica. **(f)** In the worst case where the application must failover to the last replica, this last retry will not be tagged as latency-critical so that the I/O will complete and not be revoked.

5.1.2 Overall Architecture

Figure 5.2 shows LinnOS's overall architecture, which consists of five main components.

(a) The model. At the center of LinnOS is the speedy inference model (Section 5.2.3) with a light neural network. The model's input features are information about the current outstanding I/Os and recently completed ones. The model infers the speed of every incoming I/O individually. The model's output is the binary inference about the I/O (fast/slow).

(b) Tracing. To train the model, LinnOS uses the current live workload that the SSD is serving. To have a rich representative data, this can be done during normal busy hours. The I/O metadata (block offset, size, read/write) and their resulting latencies are recorded using `blktrace`. With millions of I/Os collected, this naturally forms the "data lake" of our model. The training data (the

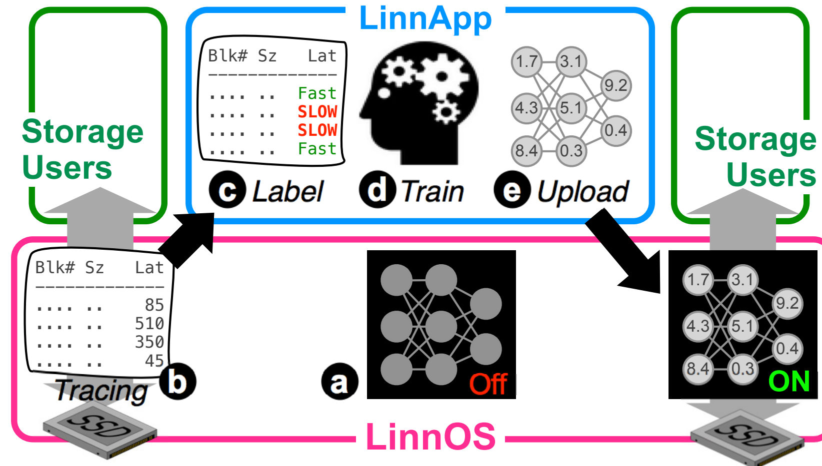

Figure 5.2: **LinnOS architecture.** *The figure displays LinnOS architecture including LinnApp, as summarized in Section 5.1.2. The two SSD pictures represent the same SSD instance; the left one depicts tracing/training and the right one live inference on the SSD.*

collected trace) is expected to be different than the "test data" (the I/Os that will be inferred when the model is activated).

(c) Labeling with inflection point analysis. The collected trace is then supplied to LinnApp, a supporting user-level application. LinnApp has three main jobs: labeling, training, and uploading trained weights to LinnOS. Because the model is designed to produce a binary output, the model must be trained with two labels, "fast" and "slow." Hence, given a latency distribution in the trace, LinnApp runs an algorithm (Section 5.2.2) that finds the "inflection point," a latency value that divides the fast and slow latency ranges.

(d) Training. With this inflection point, LinnApp labels the traced I/Os with "fast" and "slow" labels and proceeds with the training phase (using TensorFlow). We emphasize the labeling is done automatically *without* human input. This training phase can be run anywhere, on GPU or CPU nodes.

(e) Uploading weights. The training phase generates the weights for the neurons in the model that will be uploaded to LinnOS. Because using floating points is not well supported in OS kernel, the weights are converted to integers by quantization. The model is then activated, and LinnOS is

ready to make inferences and revoke “slow” I/Os.

5.1.3 Challenges

Using a machine learning approach for making online, fine-grained inferences on I/O speed requires us to solve the following fundamental challenges.

High accuracy. The inference must be accurate. We should not revoke I/Os that can be served fast (“false revoke”) or submit those that will be slow (“false submit”). Accuracy depends on careful output labeling and input features selection. If the label classification is too complicated, high accuracy is hard to achieve, *e.g.*, we find that classification by linear bucketing (0-10µs, 10-20µs, etc.) or exponential bucketing (0-1µs, 2-4µs, etc.) is hard to make accurate and should remain as a future work. However, the simple two-class approach (fast or slow) simplifies the output into a binary format, which helps the model achieve high accuracy.

Fast inference. For modern SSDs, while the raw NAND read latency is advertised to be below 100µs, we see that for typical production workload on data-center SSDs (Section 5.3), the actual user-perceived latency is above 200µs more than 50% of the time. Given this observation, we believe the challenge is to do decision-making in around 5µs, a $<$3% overhead per I/O. Fast inference depends on input preprocessing, the depth of the layers, neuron complexity, and feature representation. Using deep layers that tend to improve accuracy is not attractive in our problem domain. The input features must be minimized to include only the features that matter. Hence, we must balance accuracy and performance. Moreover, considering that operating systems run on CPUs, the models inside must be CPU-friendly [261].

Anticipating heterogeneity. In flash arrays (RAID or cluster-based), the user load is not always balanced, and all the flash hardware might not be homogeneous. Because this heterogeneity can lead to different latency distributions observed on different devices, we should not use one global latency value (*e.g.*, 1ms inflection point) to differentiate fast and slow speed for all the de-

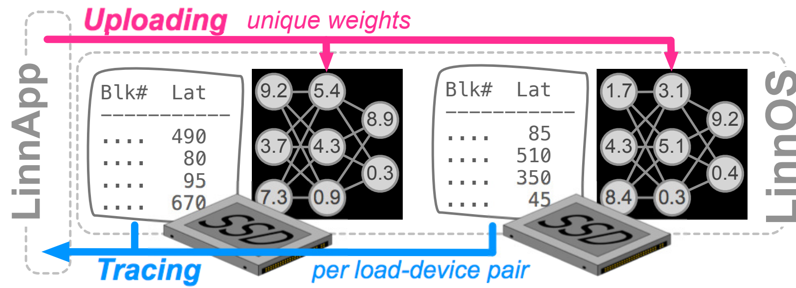

Figure 5.3: **Anticipating heterogeneity.** *The figure shows heterogeneous trained models, as mentioned in Section 5.1.3.*

vices in the array. For example, 3ms perhaps could be considered fast enough on slower SSDs or the ones with heavier user load. While we do not expect that the heterogeneity will be extreme (*e.g.*, a good storage system typically balances the load very well), heterogeneity is still important to address. For this reason, LinnApp collects per-device traces and trains the model for *every* load-device pair in the array (Figure 5.3). After the training phase completes, LinnApp supplies the model weights to all instances of LinnOS in a cluster-based array or to one instance of LinnOS in a RAID-based array. In the latter, LinnOS carries N trained models for the N drives in the RAID. Furthermore, to anticipate workload changes over time, LinnApp occasionally recollects traces (*e.g.*, every few hours) to check if the inflection point has shifted significantly such that the model must be retrained.

5.2 LinnOS Design

In this section, we dive deep into our solution to the challenges mentioned above. To the best of our knowledge, LinnOS is the first operating system that successfully infers I/O speed in *fast*, *accurate*, *live*, *fine-grained*, and *general* fashions. The key to this is the "lightness" of the neural network model that LinnOS employs. This section presents the final design and the principal intuitions about how we get here. We will explain LinnOS design chronologically, from data collection (Section 5.2.1), labeling via inflection point analysis (Section 5.2.2), the model design (Section 5.2.3), and how to improve its accuracy (Section 5.2.4) and performance (Section 5.2.5),

and summarize the advantages again (Section 5.2.6).

5.2.1 Training Data Collection

This project started with a simple question: can we infer the performance of every I/O accurately? Since we use machine learning, accuracy depends on the amount of true-signal data available, the more, the better. Fortunately, I/O systems inherently can collect a large amount of data. Given low-overhead tracing tools and hundreds of KIOPS of workload that modern SSDs can serve, collecting a large amount of data for training is not an issue (a large "I/O data lake").

For every load-SSD pair to model, LinnApp collects traces of the real workload running on the drive. For example, for inferring a production workload performance on a particular SSD in deployment, an online trace will be collected. For every I/O, we collect 5 raw fields, the submission time, block offset, block size, read/write, and most importantly, the I/O completion time. Because the model input (Section 5.2.3) does not necessarily take the same raw fields, in this phase, we also convert the fields to the input feature format.

The main challenge here is to decide how long the trace should be. If the behavior of the training data (the latency distribution) is very different from that of the "test" data (the to-be-inferred I/Os), the inference accuracy will drop. In this work, we take a simple approach where we use a busy-hour trace (*e.g.*, midday). In the evaluation (Section 5.3.2), we show that for production workloads, a busy-hour trace well represents the other hours, *i.e.*, the inflection point does not deviate much. As mentioned above, to anticipate a dramatic shift in workload behavior, retracing and retraining can be done.

5.2.2 Labeling (with Inflection Point)

As we employ a supervised classification approach, the model must be trained with labels. If we label every I/O with the actual µs-level latency, there will be too many labels for our problem domain; a user might not care if the I/O is delayed by 1µs. Another option is to use a linear

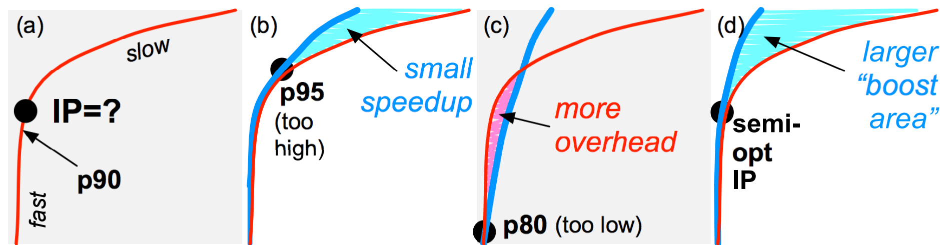

Figure 5.4: **Inflection point (fast/slow threshold).** *The figures show the results of using higher, lower, and semi-optimum inflection point (IP) for fast/slow threshold as explained in Section 5.2.2. The figures format is latency CDF, as in Figure 2.1.*

(0-10µs, 10-20µs, and so on) or exponential labeling (2-4µs, 4-8µs, and so on). While these fit better, the model is still hard to make accurate and fast after many design iterations. The accuracy only reached 60-70% because many times, an I/O that should fall into a specific group (*e.g.*, 128-256µs) is often mis-inferred to the neighbor groups (*e.g.*, 256-512µs)—“a Lhasa Apso dog can easily be misidentified as a Shih Tzu dog.” This is perhaps why prior successes in auto-learning storage performance were only done at a coarse-grained level such as average latency or throughput aggregated for many requests [136, 257, 274].

With all this mind and the understanding of how performance variance behaves in the field [81, 99, 130, 183, 199], we observe that latencies often form a Pareto distribution with a high alpha number [19]. As an example shown in Figure 5.4a, 90% of the time, the latency is likely stable, but in the other 10% of the time, it starts forming a long tail. Such a Pareto distribution clearly contrasts the fast and slow regions. Hence, a simple conjecture can be made that users only worry about the tail behavior, not the precise latency.

To separate the two regions, we need to find the “best” *inflection point* (marked with “**IP**=?” in Figure 5.4a) for maximizing the latency reduction. Setting the inflection point too relaxed (*e.g.*, the p95 latency in Figure 5.4b) will make LinnOS treat the slow I/Os between p90 and p95 as “fast” (no failover), reducing the opportunity for effective retries, hence failing to cut many tail latencies, as highlighted by the small shaded area between the original and projected distributions (more in Section 5.2.2) in Figure 5.4b. On the other hand, setting the inflection point too low (*e.g.*, the p80 latency in Figure 5.4c) will make LinnOS revoke too many I/Os, including those that are supposed

to be fast, which will induce unnecessary retry overhead as shown in Figure 5.4c.

An optimum inflection point implies that for every slow I/O that will be revoked, it can be guaranteed that the other replicas can serve it fast within the same time frame. Likewise, for every fast I/O, it should not be failed over. Finding this optimum point will deliver the maximum gap between the original tail-heavy and tail-free distributions, as shown by the large shaded area in Figure 5.4d. Finding an optimum value however is hard in practice, fundamentally because of the many unknowns: we do not know which replica the request will be failed over to (application dependent); the training data is only an approximation of the future unknown test data; other variability such as CPU or network contention can factor into unknown retry overhead. The next section describes our best-effort algorithm in finding a semi-optimum inflection point for every workload-device pair.

Inflection Point Algorithm

First, during data collection, we collect t workload traces (T_1 to T_t) running on d devices (D_1 to D_d), respectively, where $t{=}{=}d$. Every trace T_i gives us the latency distribution of the workload running on the device (as in Figure 5.4a). To find the unique inflection point (IP) value for every $T_i{-}D_i$ pair, we run a user-space simulation based on random replica selection as follows. For illustrative purposes, we use specific device numbers (*e.g.*, D_1) in our explanation below.

(1) For every $T_i{-}D_i$ pair, we pick a starting IP value where the CDF line (with x-axis and y-axis of same visual length) begins bending at 45 degree (likely entering the tail area). For example, if for D_1, the T_1's 45-degree slope is at $y{=}p90.5$ and $x{=}1ms$, then the IP value is initially set to 1ms. **(2)** For the currently simulated device, D_1, we run a simulation of 1 million I/Os, ($r_{i=0..1000000}$) where each I/O request r_i takes a random latency value from T_1's real latency distribution. We then simulate LinnOS admission control: if the chosen latency is smaller than 1ms (the current IP), the r_i's new latency is set to be the same; else, if it is larger than 1ms, it will be revoked and failed over to another randomly selected node (*e.g.*, D_4) where a new random latency

is picked from its trace, T_4, and the admission control is repeated (submit or revoke). We assume three replicas (configurable), hence a request can only be revoked a maximum of two times. **(3)** The simulation produces the new, optimized latencies for all the r_i in workload trace T_1 that previously went to only one device, D_1, but now can be redirected as if LinnOS admission control is activated. These optimized r_i latencies form the new CDF (as in the bold blue line in Figure 5.4d). Using the original and new CDFs, we can calculate the *area difference* (the shaded "boost area" in Figure 5.4d), which represents the latency gain if $1ms$ ($p90.5$) is used as the IP value. **(4)** Still, for D_1, we repeat all the steps above by moving $^+/\!-0.1$ percentile within the $^+/\!-10$ percentile ranges from the initial IP value. For every new IP value, the simulation gives a new boost area. We now can pick the IP^{max}, the IP value that gives us the *largest (positive) boost area*, which will be used as the fast-slow threshold in training the model for device D_1. **(5)** We repeat all the steps for other devices (D_2, D_3, etc.). At the end, for every $T_i - D_i$ pair, our algorithm generates a unique IP_i^{max} value. All these steps are repeated upon recalibration (Section 5.2.4).

5.2.3 Light Neural Network Model

Before we decided to build a light neural network model, we explored various learning methods such as logistic regression, decision trees, and random forests. We found that the accuracy only ranges from 17-84%, while a basic neural network can reach a better accuracy. Although it is possible to continue optimizing each of these methods to its full potential, we decided to start from an acceptable baseline that our initial neural model delivered. Below, we describe our final model (Figure 5.5), from input features, their representation, to the neural layers. We will emphasize how we use storage intuitions to design the model, as opposed to brute-force.

Input features. To infer the speed of every I/O, our model takes three inputs: **(a)** the number of pending I/Os when an incoming I/O arrives (in the number of 4KB pages, including the incoming I/O), **(b)** the latency of the R most-recently completed I/Os, where we set R as 4, and **(c)** the number of pending I/Os at the time when each of the R completed I/Os arrived. We now reason

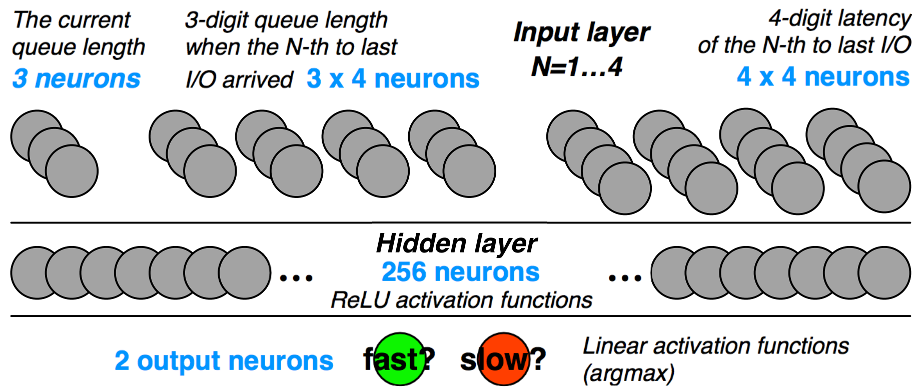

Figure 5.5: **Light neural network.** *The figure depicts LinnOS 3-layer neural network explained in Section 5.2.3.*

about these necessary inputs.

Deciding the first feature is straightforward—an I/O latency typically correlates with how many I/Os are currently pending. The unit we use here is the number of 4KB pending pages, and the reason is that the lowest granularity of striping inside SSDs is typically at the page level and the main contention is at channel and chip level.

While for disks, the first feature might be sufficient for inferring single-spindle performance, for SSDs, the other two features are required. In essence, to speculate whether the SSD is currently busy internally, we need to record a small piece of historical information, the latencies of the last four I/Os, as well as how many pending I/Os existed when those I/Os arrived. Put simply, if recent I/Os experienced a long delay without many pending I/Os, then the model could learn that there is likely an internal contention due to device-level activities such as GC, internal flushing, or wear leveling. In this case, the model will suggest revoking incoming I/Os until the number of pending I/Os drops substantially so that the device can possibly provide fast responses even with heavy internal activities. Once the device resumes serving I/Os, the model can tell whether the device-level contention is over by learning the returned latency values.

Our features above look simple because we have removed unnecessary features after many design iterations. For example, we surprisingly found that important-looking features such as block offsets, read/write flags, or long history of writes do not significantly improve accuracy. We make

several conjectures. First, on read/write flags, although NAND-level read/write latencies differ, almost all medium/high-end SSDs employ write buffering. Thus, the problem of read-behind-write is no longer observable. More likely observable is read-behind-buffer-flush delays, which can be learned from our input features. Second, on block offsets, because we target production workloads and the fact that SSDs typically stripe incoming I/Os uniformly across all channels and chips (or with some bounded partitioning), the workload is likely to be evenly scattered, hence block offsets do not really matter for learning. In other words, scenarios where a batch of incoming I/Os with block offsets that simultaneously hit only one chip rarely happen in the field. Third, on history of writes, internal activities such as GC and buffer flush often happen in a short burst, hence they can be sensed by just observing the speed of the last four I/Os. These are surprising but fortunate findings because using just a small set of features will reduce the model's overhead.

Input format. The next challenge is to choose the right input format to be fed to the neurons. First, for the R value, if accuracy is the only important metric, we should record more completed I/Os (the higher R, the better), but it would prolong inference time as the number of neurons would increase. We found that $R{=}4$ suffices for balancing performance and accuracy.

In another simplification, we format the number of pending I/Os into three decimal digits. For example, the format for 15 pending I/Os is three integers $\{0{,}1{,}5\}$. Three digits suffice as device queue length of over 1,000 is rarely heard of. Similarly, for the latencies of the recent completed I/Os, we break the µs latency value into four digits. For example, a latency of a recent I/O that completed in 240µs will be formatted as four features $\{0{,}2{,}4{,}0\}$. Latencies larger than 9,999µs will be capped to $\{9{,}9{,}9{,}9\}$. In total, our model takes 31 input features, each a one-digit decimal number.

Reformatting the original integers into decimal digits is an effective trade-off. If we use bits and supply every bit to every neuron, there will be too many neurons that increase the model size and hurt inference time. On the other extreme, if every neuron takes a raw integer value, the neurons need to learn over a wide input range, which makes learning/training harder (*e.g.*, latency value

can range from 1µs to over 9,999µs). With decimal digits, we make the neuron learning bounded within a small range of 0 to 9.

The network. The final model is a fully-connected neural network with only three layers ("light"), including one input/preprocess layer, one hidden layer, and one output layer, as shown in Figure 5.5. All the neurons are regular linear neurons ($y{=}wx{+}b$),

The input layer is supplied with the 31 features described above. The raw information from the block layer is converted to the feature format, in an offline way for training and an online way for live inference. For the latter, with some programming optimization, we can achieve O(1) preprocessing overhead. Next, the hidden layer consists of 256 regular neurons. This layer uses RELU activation functions for its low computation cost and ability to support non-linear modeling. More neurons will cause longer inference time and fewer neurons less accuracy. Lastly, the output layer has 2 neurons with linear activation functions. We use an `argmax` operator to convert the output to a binary decision (*e.g.*, $\{0.4{,}0.6\}$ to $\{0{,}1\}$). Overall, this design makes the network lightweight and easy to integrate into the OS, while balancing inference accuracy and performance.

Preceding design iterations. Here we briefly describe how we reach the current design. We started by using the I/O offsets in binary format (32-bit) as the input features since the device FTL mapping basically uses I/O offsets to decide where the I/Os go, which defines the resource contention. This setting allows the learning models to achieve higher accuracies (up to 99% for some traces), however it has a heavy model and high inference overhead, which is impractical for real-time usage. We further trimmed the heavy model but could not find a reasonable tradeoff between generality and inference overhead. As a result, we took a step back from the fine-grained features and switched to a more aggregate one, and finally reached the current design.

5.2.4 Improving Accuracy

To further improve the model accuracy, we perform false-submit reduction via biased training, model recalibration via retracing/retraining, and inaccuracy masking with high-percentile hedging.

Reducing false submits. An accurate inference means LinnOS submits I/Os that will be fast (true negative) and revokes those that will be slow (true positive). Reversely, inaccurate cases can be categorized into **(a)** "false submit" (false negative) wherein the model believes the request will be served fast, making LinnOS submit the request to the device, but the request will take longer than the fast-slow threshold, or **(b)** "false revoke" (false positive) where the I/O is revoked, but in fact, it can be served fast by the device.

Using the same system intuition on typical latency distributions in the field (Section 5.2.2), we found that *reducing false submits* is far more important, while false revokes are more tolerable. The reason for the latter is that when the storage devices of a cluster exhibit similar tail behavior (high-alpha Pareto), the *probability* that peer devices are simultaneously busy is relatively *small*. For example, with three replicas and P% busyness, the probability that all the replicas are busy around the same time is $(P/100)^3$ (*e.g.*, 0.000125 with 5% busyness). Another factor is the increasing network speed where a failover cost can be as low as 1-6µs for flash arrays across PCIe or Fiber Channel or 5-40µs across Ethernet [7] (plus some negligible software overhead).

To summarize, the wrong inference penalty is small for false revokes but high for false submits. In the latter, the I/O will be "stuck" in the device and cannot be revoked. This motivates us to use *biased training* for reducing false submits by allowing more false revokes. We do this by customizing the categorical hinge loss function with a multiplier that puts more penalty weights for false submits, which makes the trained models favor false revokes.

Recalibrating. Another source of inaccuracy happens when the inflection point computed over the training data does not represent the same threshold of the "test" data (the workload during live inference). This can happen under significant workload changes that cause shifts in the latency distributions of the nodes in the cluster. Fortunately, our evaluation of production traces shows that latency distributions do not widely shift across hours (Section 5.3.2). However, to anticipate this scenario, re-tracing and re-computation of inflection point analysis can be done periodically every a few hours. If in the new workload-device pair, the inflection point has shifted by five percentiles,

LinnApp will retrain the model using the newly collected trace and re-upload the new trained weights to the device. Running `blktrace` during the busiest hour in the production workloads we use only generates 300 MB of data (85 KB/s of trace writes) and increases CPU overhead by 0.5% (only relevant parameters are traced).

Masking small inaccuracy. Our methods above managed to increase accuracy up to 98%. Just like other neural networks, achieving 100% accuracy is fundamentally hard and usually implies a lack of generality. Within the small inaccuracy, long latency tail due to false submits still needs to be circumvented. This is where we marry learning and hedging [99]. When the false submit rate[1] (Section 5.3.4) is significant (*e.g.*, >5%), we use the rate as an indicator for the hedging percentile value. For example, if 6% of the inferences produce false submits, then p94 hedging will be applied. For the rare cases where the false submit rate is lower, we use the conventional p95 hedging, which typically achieves a good balance between issuing extra I/Os and tail-cutting. Though sometimes this design issues more extra I/Os, it can bring a further performance improvement (Section 5.3.3).

5.2.5 Improving Inference Time

A large part of deep neural network (DNN) research mainly focuses on how to structure even larger networks to achieve the highest possible accuracy [23]. Strict latency is not a typical constraint for neural networks. However, putting a neural network into the storage layer poses a unique challenge. Our goal is to reach around 5µs of inference time (as discussed in Section 5.1.3), and although the 3-layer design is fundamental to reach the goal, we made further optimizations.

Quantization. First, neuron weights are by default in floating points for improving accuracy, but it is an overkill for our purpose. Some of the major storage functionalities that define contention are striping and partitioning using `mod` operations over integers, which does not require

1. To clarify, different from conventional way of calculating false positive/negative, in this paper, the false submit rate is based on the submit decision and the actual resulting latency.

ultra-high precision. Besides, floating point calculations are expensive and hard to manage inside the OS. Hence, we adopt DNN quantization by maintaining precision of three decimal points; the trained floating-point weights are converted to integers with precision of three decimal points. DNN quantization is a popular technique to reduce the space, power, and computation cost of DNN on mobile-platform and IoT devices, albeit some loss on accuracy [96, 146, 271]. In our case, the accuracy loss from quantization is bounded within 0.1%.

Co-processors. Second, using additional accelerators such as GPUs and TPUs may be possible in the future, but currently, they are optimized more towards throughput and not easy to interact with the host kernel code. If we move the inference to GPUs, the cross-communication would add more overhead. Furthermore, technology trend suggests that 100-200x improvement on inference latency can be foreseen in the near future with more advanced hardware innovations [16], which can make LinnOS future deployment more practical, especially when storage devices are getting faster. However, until this technology arrives, we show that LinnOS can opportunistically use co-processors (if available) to reduce the average inference time (*e.g.*, from 6 to 4µs with 2-threaded optimized matrix multiplication using one additional CPU core).

5.2.6 Summary of Advantages

With all of the techniques, LinnOS delivers advantages in various dimensions, which we show in the evaluation.

- *Performance predictability*. The most important advantage is that LinnOS helps storage applications achieve predictable performance on flash arrays, outperforming other popular methods.
- *Automation*. LinnOS auto-infers unpredictability by learning from millions of I/Os as a natural data lake and automatically trains and produces neuron weights for different workloads and devices. Storage developers do not have to tweak and configure heuristics manually.

- *Generality*. To achieve predictability, LinnOS does not require device-level modification nor a heavy redesign of file systems or applications. Storage applications simply need to tag latency-critical I/Os. Failover/retry logic is already a standard in many storage applications with data replicas.
- *Timeliness*. With fast inference, the application can failover as soon as the `slow` error code is returned, without the need to wait (*i.e.*, a more "timely" decision).
- *Efficiency*. With auto-revocation, LinnOS eliminates duplicate I/Os suffered in hedging. Some production systems do not use hedging for the same reason and instead use a more efficient method such as "tied requests," where clones are sent but when one of them is served, the duplicate is canceled [99]. Similar to this "clone-then-cancel" method, our "(timely) revoke-then-failover" also avoids duplicates. Furthermore, while some implementation of tied requests burdens the application layer [99], LinnOS supports I/O revocation inside the kernel.
- *Simplicity*. We do not require applications to supply an SLO value such as a deadline [63, 129, 250, 278]. I/O system calls today do not accept SLO info, arguably because setting the proper SLO is not easy [154, 184]. LinnOS simplifies all of this with fast/slow binary classification.

5.2.7 Implementation Complexity

LinnOS extends Linux v5.4.8 in 2170 LOC within the block layer, mostly for the neural network model (written in C) and the simple revocation mechanism. The memory space needed for one neural network model (in total 8706 weights and biases) is only 68 KB of kernel memory. LinnApp is written in 3820 LOC including for data collection, labeling, training (using TensorFlow), and quantization.

5.3 Evaluation

In this section, we first describe our evaluation setup (Section 5.3.1) and then present the results that answer the following important questions:

- *Stability* (Section 5.3.2): Is our inflection point algorithm stable enough for production workloads?
- *Latency predictability* (Section 5.3.3): Does LinnOS successfully deliver more predictable latencies compared to other methods?
- *Accuracy* (Section 5.3.4): How accurate is LinnOS neural network in inferring per-I/O speed?
- *Trade-off balance* (Section 5.3.5): What are the performance and accuracy trade-offs that LinnOS balances?
- *Others* (Section 5.3.6): How does LinnOS work on other public traces? Can LinnOS support full-stack storage applications? What is the CPU overhead?

5.3.1 Setup

We present the evaluation workloads, devices, experiments, and methods to which we compare.

Workloads. Our ultimate goal is to evaluate whether LinnOS can help real production scenarios. We use SSD-level traces from Microsoft Azure (**AZ**), Bing Index (**BingI/BI**), Bing Select (**BingS/BS**), and Cosmos (**CO**) servers. Each server type contains I/O traces for six devices. The average trace contains 36 hours of I/O operations.[2] For training data, from each of the four server types, we pick three busiest device traces and then pick the busiest hour (three same hours); we limit to three due to the number of (expensive) enterprise SSDs that we have (more below). For the "test data" that is dedicated for live experiments, we pick a random time slice from other busy

2. The traces are available from Microsoft to the academic community with NDA.

hours, hence training and test data do *not* overlap. Overall, the training and test data do not occupy the entire available traces.

SSD devices. For performance evaluation, we show how much LinnOS helps flash arrays deliver predictable latencies. We prepared two flash arrays with consumer (**"C"**) and enterprise (**"E"**) configurations. The former connects an array of three homogeneous SM951 consumer-level SSDs, and the latter forms three *heterogeneous* enterprise-level SSDs, Intel P4600, Samsung PM1725a, and WD Ultrastar DC SN200. We assume every block is replicated three times across the devices, a typical setup for consumer-facing storage servers. For both configurations, the machine has a 2.6GHz 18-core (36-thread) Intel i9-7980XE CPU with 128GB DRAM. We mainly do not use any accelerators (Section 5.2.5). The overhead for failing over revoked I/Os is 15µs. For accuracy evaluation, beyond these four flash models, we also use Intel SSDSC1BG40, Intel SSDSC2BX01, Intel P3700, Intel P4510, Intel S3700, and Samsung 960 EVO, for a total of 10 models. Prior to this evaluation, all devices have been used for months with many workloads that reach the devices' full capacities, hence mimicking devices in the field.

The experiments. For performance evaluation, the experiments are performed with a storage application that executes the traces on the flash arrays, where all the devices serve read/write workloads. For example, in one experiment, the application simultaneously executes three different Azure traces on three separate SM951 devices in the consumer flash array and records the latencies of completed read I/Os. The application has a failover capability to complete revoked I/Os at other devices (as shown earlier in Figure 5.1). All read I/Os are marked as latency-critical.[3] We are also aware that the traces were collected on medium-end devices at Microsoft (in 2016). Hence, for our high-end flash array configuration, we have to mimic a heavier workload by re-rating the traces to be more intensive. Our methodology is that for each re-rated trace, the resulting baseline latency distribution (after running it on the high-end device) should be similar to the latency distribution

3. We assume that no write I/Os are latency critical as they are usually absorbed by write buffers. However, if needed, our techniques can be easily integrated with kernels/applications that support write re-routing (*e.g.*, AutoRAID, RAID+).

Test Trace	Avg IOPS	Max IOPS	R:W Ratio	Avg I/O Size Read/Write	Max I/O Size Read/Write
AZ/C	745	4.9K	27:73	24K/18K	64K/64K
BI/C	361	1.8K	17:83	57K/30K	64K/1M
BS/C	114	1.1K	22:78	163K/73K	2M/9M
CO/C	113	623	32:68	479K/121K	6M/32M
AZ/E	13K	31K	25:75	25K/17K	64K/64K
BI/E	2.4K	9.2K	23:77	55K/30K	512K/1M
BS/E	1.3K	4.3K	27:73	196K/73K	2M/9M
CO/E	2.5K	7.2K	22:78	430K/107K	7M/32M

Table 5.1: **I/O characteristics of re-rated traces (Section 5.3.1).** *The upper part (first four rows) is for the consumer-level flash array and the lower is for the enterprise-level one. Every max-IOPS value is measured within a 10-second window.*

in the original trace.

Table 5.1 shows the I/O characteristics of the re-rated traces. Typically, running these re-rated traces on our drives shows a low slack (<5% of all I/Os) and noticeable burstiness (5-30%, where I/Os need to wait in the OS as the SSD queues are full), which we believe accurately emulates the slack and tail behaviors seen in real deployments. Also, we notice that the workload bursts across devices are highly correlated, which, in some cases, can cause inevitable long-tail behaviors that no failover can handle. However, in real runs we find that the internal busyness of the devices is not necessarily correlated due to device-level complexities, as LinnOS shows great improvement by evading the underlying device idiosyncrasies (Section 5.3.3). All the experiments are repeated three times, and no significant variance was observed.

Methods compared. We perform an extensive evaluation that compares eight methods: baseline, cloning, constant-percentile hedging (*e.g.*, at p95 latency), inflection-point hedging (with our algorithm), simple heuristic, advanced heuristic, LinnOS (by itself), and LinnOS with high-percentile hedging. Comparing LinnOS with white-box approaches [129, 140, 180, 216] is out of the scope of the paper because LinnOS targets black-box devices and we do not have access to an array of programmable devices.

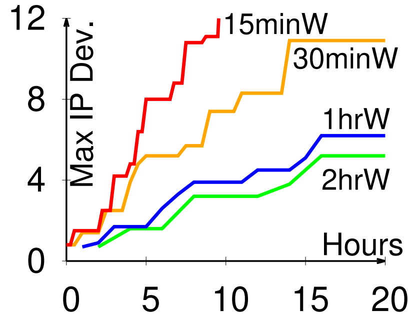

Figure 5.6: **IP stability.**

5.3.2 Inflection Point (IP) Stability

One of the contributions in this paper is finding the semi-optimal fast/slow inflection point (IP) that brings a balance between timeliness and overhead (Figure 5.4 in Section 5.2.2). Table 5.2 shows the IP values our algorithm computed for every workload-device pair. The three numbers in every cell represent three different traces (from the same server type), each running on one of the SSDs in the flash array. As shown, the IP values widely range from p72 to p98, which highlights why a constant timeout value is *not* optimal and hurts performance as we show later. These IP values will be used for fast/slow labeling and training, which then generates a unique set of weights for each device.

In our approach, we chose a busy hour (T=`1hr Window`) to collect the training data and calculate the IP values in that time slice. Figure 5.6 shows the stability of our methodology by plotting the max IP deviations (y-axis) within the next 20 hours (x-axis) for various T window values. For example, if the chosen hour exhibits p85 IP, but the next hour exhibits p75 or p95 IP, then the deviation is 10 percentiles (y=10). The graph shows that if T=`1hrWindow`, the deviation is bounded within 5 percentiles in the next 15 hours, indicating that frequent retraining is unnecessary. If T is shorter (*e.g.*, `15minWindow`), the deviation is more apparent (needs frequent retraining, which typically converges within 15-20 minutes on CPUs, due to LinnOS's light model), and if T is larger (`2hrWindow`), the gain is not significant. For generality, the figure is the result of our algorithm simulation on all the datasets (36 hours per trace, 24 traces, 4 server types).

	Consumer	Enterprise
Azure	p73.3, p77.0, p91.4	p91.0, p93.2, p97.8
BingIndex	p80.0, p94.5, p98.5	p80.1, p83.3, p97.0
BingSelect	p72.0, p76.9, p87.2	p75.3, p83.7, p86.8
Cosmos	p73.4, p82.5, p84.1	p83.2, p84.8, p95.1

Table 5.2: **Inflection point (IP) settings.** *This table, as explained in Section 5.3.2, shows the IP values that our algorithm in Section 5.2.2 computed for every workload-device pair.*

The cost of delayed retraining depends on the deviation. Let us take an example of a model trained for p95 (5ms), but then the workload deviates such that the IP is now at p90 (10ms) because the workload becomes more write-intense. In this case, LinnOS (still using 5ms) will over-revoke many IOs that could have finished before 10ms (more false revokes). If the failover overhead is negligible, this will not cause much harm. The other scenario is when the workload deviates such that the IP moves up to p99 (3ms). Here, LinnOS would over-accept (more false submits) because 3-5ms latency is inaccurately considered "fast," but actually can be made faster. This is where LinnOS without retraining hurts.

5.3.3 Latency Predictability

We now evaluate LinnOS's success in achieving extreme latency predictability. Figure 5.7 shows the average I/O latencies (user-perceived) on the two flash arrays (consumer and enterprise) across the eight methods. In more detail, Figure 5.8 shows the latencies at specific percentiles (p80 to p99.99 in the x-axis). Below we dissect the strengths and weaknesses of every method. We start from the baseline, then we jump to "`LinnOS+HL`" (the best outcome), followed by the others.

Baseline. The `Base` lines in Figure 5.8 confirm unpredictability of flash storage with latencies that spike almost exponentially in between p95 to p99.99, increasing the average latencies to 1.3–6.5 times compared to our best cases (Figure 5.7). Clearly, flash arrays with data redundancy should adopt tail-cutting methods to achieve higher predictability.

LinnOS+HL. This label represents the LinnOS method combined with high-percentile hedg-

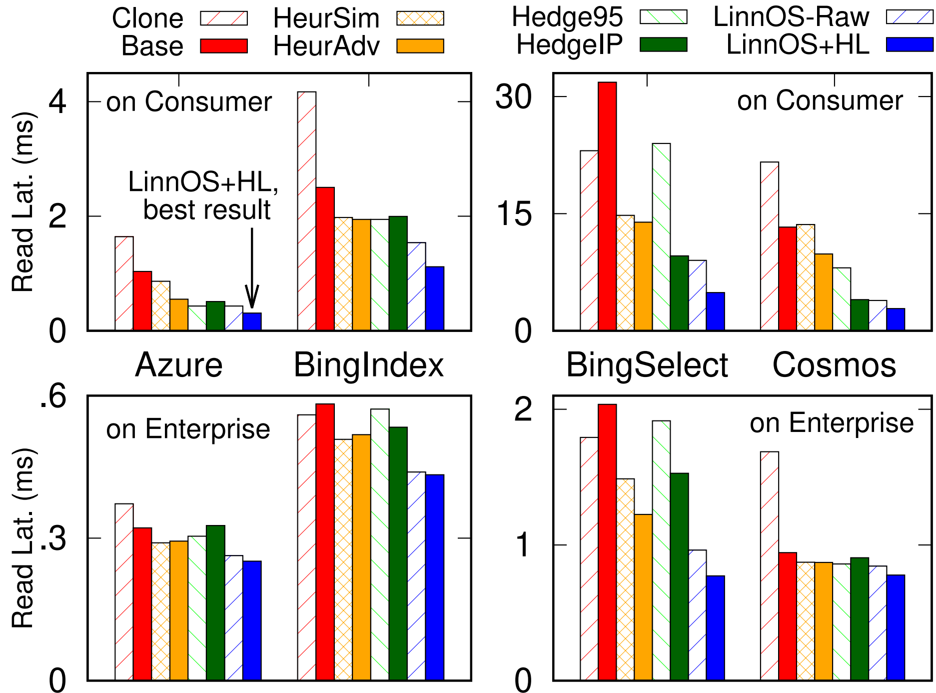

Figure 5.7: **Average latencies.** *The figures show that LinnOS consistently outperforms all other methods, as explained in Section 5.3.3. The top and bottom graphs represent experiments on the consumer and enterprise arrays, respectively.*

ing for masking the small inaccuracy that is intrinsically hard to eliminate in a neural network (Section 5.2.4). That is, to compensate for the inaccuracies that cause false submits, our application sends a duplicate I/O after pX latency time has elapsed, where X is the smaller of 95 and (1 – false submit rate)$\times$100.

[Key outcome] $\rightarrow$ The average latencies in Figure 5.7 show that `LinnOS+HL` *consistently outperforms all other methods across different workloads and platforms.* On average, `LinnOS+HL` reduces latency by 9.6-79.6% compared to p95 hedging (`Hedge95`), 14.2-49.5% to hedging with our IP algorithm (`HedgeIP`), and 10.7-71.2% to an advanced heuristic (`HeurAdv`). These speed-ups are a product of the stable latencies; *in Figure 5.8,* `LinnOS+HL` *lines exhibit stable latencies even at extremely high percentiles, p99 to 99.99.* These results bring a positive conclusion that the downsides of LinnOS (a 15µs failover overhead including a 6µs per-I/O inference cost and the inaccuracies) are outweighed by its effectiveness in delivering predictable latencies.

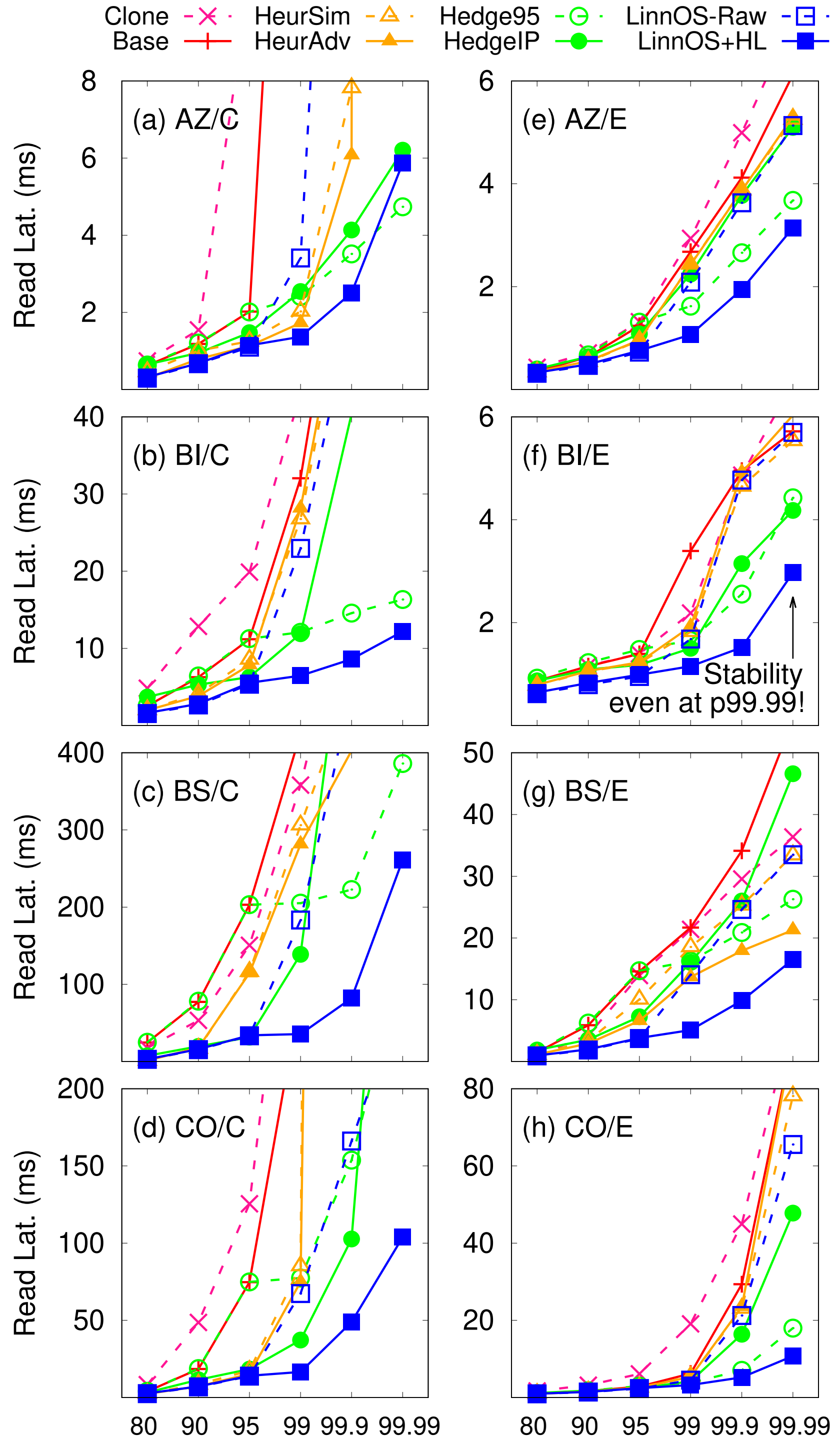

Figure 5.8: **Percentile latencies.** *Explained in Section 5.3.3, the figures show that LinnOS+HL delivers the most predictable latencies (y-axis) across all percentiles (x-axis), even at p99.99. In Figure (a), "AZ/C" means Azure running on consumer array.*

LinnOS (Raw). Here we show LinnOS efficiency even without hedging (*i.e.*, revoke+failover without I/O duplication). The `LinnOS-Raw` bars in Figure 5.7 shows that LinnOs by itself is effective enough, only 1.3-45.7% worse than `LinnOS+HL`, and compared to p95 hedging, `LinnOS-Raw` reduces latency by 0.3-62.3%, and to an advanced heuristic, by 3.0-60.7%. Figure 5.8 details why adding hedging is useful. At high percentiles, above p99, `LinnOS-Raw` starts exhibiting high latencies (due to false submits). Learning from the "small-tail" behavior of hedging (*e.g.*, the `Hedge95` lines), we combined the best of the two in `LinnOS+HL`.

Hedging at p95. Sending a duplicate I/O after a p95-latency timeout has elapsed is a popular method used in the field [43, 99]. Figure 5.8 shows that, in general, this method is effective in cutting latency tail but generally incurs *higher* latencies than `LinnOS+HL`. This is because `Hedge95` needs to *wait* for the timeout to happen before sending the duplicate I/Os, while LinnOS returns a timely revocation that allows the application to failover quickly. As the implication, `Hedge95`, *on average, is slower* than `LinnOS+HL` or even `LinnOS-Raw` (Figure 5.7).

Hedging at IP. Many of the IP values shown in Table 5.2 are below p95, which raises the question of whether hedging at IP would be better than at p95. The average values in Figure 5.7 show a mixed result. On the consumer devices, `HedgeIP` improves upon `Hedge95` by 2x for heavy workloads `BingS` and `Cosmos`, but loses by up to 15% in light workloads `Azure` and `BingI`. Similarly, on enterprise devices, `HedgeIP` wins in `BingS` while slightly losing in the others. Upon further investigation, we see that, for example, in consumer devices, `Azure` and `BingI` latencies are generally fast (<2 and 10ms respectively, as shown by the y-axis in Figure 5.8a-b), hence are *sensitive* to the extra load from duplicate I/Os; `HedgeIP` in our experiments are sending more duplicates than `Hedge95`. Nevertheless, our experiments show that for most of the workloads, `HedgeIP` is more effective than `Hedge95`, hence systems with hedging can adopt our IP algorithm.

Simple heuristic. The first heuristic we wrote, "`HeurSim`," is based on a popular heuristic for spinning disks: if the device queue length (the number of outstanding I/Os) is larger than a threshold, the incoming I/O should be retried elsewhere [58, 246, 251]. For the threshold, we use

a similar method as `HedgeIP`, but instead of using IP latency value, we use IP queue length. That is, we first profile the queue length distribution during tracing and then select the queue length at the IP percentile as the threshold for revoking. Figure 5.7 shows that `HeurSim` only gives a small improvement over the baseline and is far from the best case. In short, it is not smart enough to infer device-internal disruptions.

Advanced heuristic. We extend `HeurSim` to a more "advanced" heuristic, `HeurAdv`. For comparison fairness, we reuse the same intuition we had in building LinnOS and apply it to `HeurAdv`. An additional task that `HeurAdv` performs is scanning the last N completed I/Os (N=4, same as in LinnOS) and if this history shows a slow I/O ("slow" as defined in Section 5.2.2) but with a low queue length (less than the median), it will mark the drive as "internally busy." In this state, incoming I/Os will not be admitted unless the queue length drops to a low value (less than the lower-quartile queue length). The state will not be changed from "busy" to "normal" until it sees recent I/Os become fast ("fast" as defined in Section 5.2.2).

[2nd key outcome] $\rightarrow$ Figure 5.7 shows that `HeurAdv` improves upon `HeurSim` in most cases, but still loses from other methods. We would like to note that we spent several weeks tuning the heuristic to the "best" outcome we can achieve. Continued expansion and tuning of the heuristic is possible. However, the main difficulty that will arise is the *large design space* of parameters (normal/busy states, median and lower-quartile queue lengths, etc.) that must be optimally and *manually configured* for different workloads and devices. This is where we show that machine learning helps. The use of a lightweight neural network allows us to focus on deciding what features matter, but at the same time letting the model learn and reverse-engineer SSD behaviors. In our case, LinnOS neural network *auto-trains all the 8706 weights* for different devices and workloads.

Cloning. This method is essentially p00 hedging, sending a duplicate I/O for every I/O on the outset. Although SSDs are fast and have internal parallelism, Figure 5.7 shows that `Clone` is mostly worse than the baseline due to the 2x load.

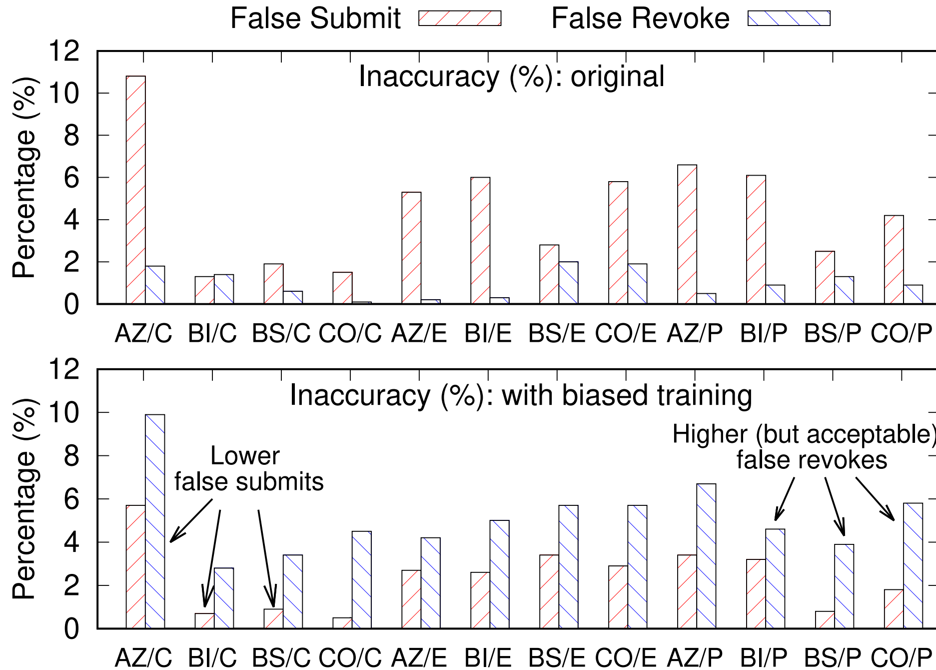

Figure 5.9: **Low inaccuracy.** *The figure shows the percentage of false submits and false revokes. Note that only false submits really matter (see Section 5.3.4). Additionally,* **"P"** *represents other device models that we can access from a public cloud. For graph readability, here for "P" we only show the results for 1 device model, while the observations stand across the rest. In total, the accuracy evaluation covers 10 device models (1C+3E+6P).*

5.3.4 (Low) Inaccuracy

We now measure LinnOS inaccuracy by counting the number of false submits and false revokes (Section 5.2.4). The live experiments can only measure the former but not the latter. This is because revoked I/Os are never submitted to the device, hence we never know whether the revoke is accurate or not. Thus, for this evaluation, we measure inaccuracy in an offline way using TensorFlow, just like the training phase. However, note that both the training and test data were collected from running the workloads on real flash arrays (*i.e.*, not simulated data). Just like before, we use 1-hour data sets for training and then pick three different 1-hour data sets for testing accuracy, and measure the average inaccuracy.

Figure 5.9 shows the inaccuracies before and after we use biased training. To recap Section 5.2.4, false submits are more dangerous than false revokes. Without bias, the top graph shows

Model:	A	B	C	D	E
Acc. (%)	−(3-12)	−(1-4)	+(1-2)	+(4-5)	+(8-12)
Perf. (µs)	−4	−1	+40	+94	+1670

Table 5.3: **Trade-offs balance.** *This table is explained Section 5.3.5. All the +/− of accuracy and performance values are compared to our final neural network model described in Section 5.2.*

that the false submit rates (red bars) are high, between 1.3% to 10.8%. With biased training, as shown in the bottom graph, we successfully lower the false submit rates to 0.7-5.7%, by *shifting* the inaccuracies to false revokes, which are more tolerable as explained in Section 5.2.4. For example, let us assume an inferior scenario of p80 inflection point (*i.e.*, 20% slowness), which means the probability that all three replicas are slow is 0.008 ($(^{20}/_{100})^3$). Thus, although we have spiked up the false revokes to 2.8-9.7% in Figure 5.9b, *only 0.008* of these false revokes probabilistically will result in slow I/Os. Finally, as mentioned before, for masking the dangerous low inaccuracy (the 0.7-5.7% false submits), combining LinnOS with high-percentile hedging (`LinnOS+HL`) led to a powerful result.

5.3.5 Trade-offs Balance

Table 5.3 shows some possible trade-offs between inference overhead and accuracy (models A-E). On one hand, if lower overhead is preferred and some accuracy loss is acceptable, then one option is to trim the input features and the model. For example, in model B with R=3 (*i.e.*, including fewer history I/Os instead of R=4) can reduce the number of input features from 31 to 24 and lower inference overhead, −1µs, but it will bring some accuracy loss, −(1-4%), due to fewer inputs. Or, if even lower overhead is favorable, then in model A we can further cut the input features (R=2, 17 features) and use a slimmer hidden layer (from 256 neurons to 128), resulting in a lower inference time,−4µs, while bringing larger accuracy loss, −(3-12%).

If higher accuracy is needed, then we can bring in more features and heavier models. For example, in model C, by adding one more hidden layer to the model, we can gain +(1-2%) higher accuracy, while the inference overhead rises by +40µs. Taking a step further, we can involve more

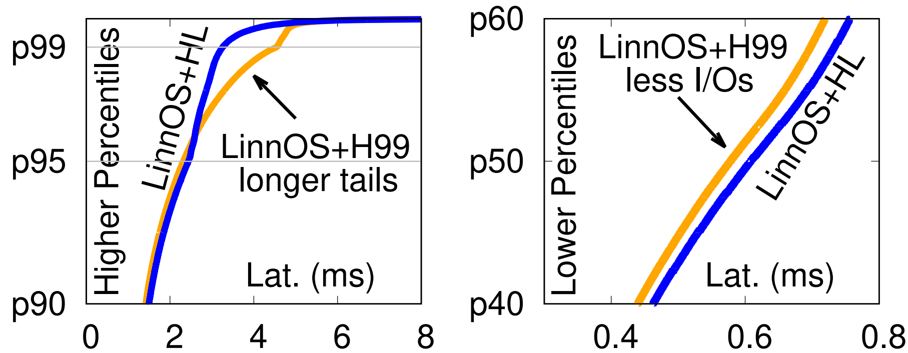

Figure 5.10: **LinnOS+H99.**

features (up to R=10 and 73 features) and more hidden layers (three layers with 256-512-256 neurons) to push the accuracy gain by +(4-5%), but an increased overhead, +94µs. The extreme model E includes block offsets in the input features (2048 features in total) and applies a model with 5 hidden layers (with 512 neurons each). For some traces, this model improves the accuracy by +(8-12)%, but its inference overhead, +1670µs, is extremely high for live inference.

5.3.6 Other Evaluations

Additional Performance Evaluations

Other possible manually-tuned heuristics. To get a sense of how much performance a heuristic can ultimately reach, we pick several 10-min slices from the traces and manually tweak the adjustable parameters of `HeurSim` and `HeurAdv` with various thresholds until an optimal outcome is achieved. In a nutshell, we start with the generic `HeurSim` and `HeurAdv`, evaluate them with the sliced traces, track the high-latency I/Os that are not revoked, update the thresholds to catch these I/Os without causing too many false revokes (*e.g.*, >15%), re-evaluate and repeat the entire process until an approximate optimum is observed. This approach is indeed capable of granting heuristics a further stretch. For example, we see a few cases where tweaked heuristics can outperform `LinnOS-Raw` by up to 20% at p95. However, this tuning procedure is onerous and impractical

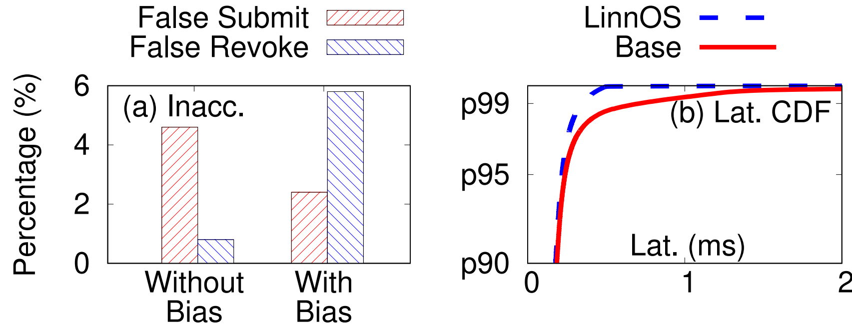

Figure 5.11: **On public traces.** *As explained in Section 5.3.6.*

in real runs as the repeated manual tweaking is too slow to catch up with the fluctuation of incoming workloads.

LinnOS+H99. We also try `LinnOS+H99`, which employs p99 hedging that only generates 1% extra I/Os. Figure 5.10a shows one of its comparisons with `LinnOS+HL`. Generally, `LinnOS+H99` encounters a larger tail area due to longer waiting, but responds faster at lower percentiles due to less extra I/Os. With that, sometimes `LinnOS+HL` can show slightly worse average latencies (up to 3%) than `LinnOS+H99` (Figure 5.10b). However, in a large majority of our benchmarks, `LinnOS+HL` achieves 1.7-39.2% better average latencies than `LinnOS+H99`.

On Public Traces

Beyond our evaluation with Microsoft traces, Figure 5.11 shows a quick evaluation with the latest SSD traces published on the SNIA website [20] run on our consumer flash array. The result confirms that LinnOS also exhibits low inaccuracy (Figure 5.11a) and substantial latency improvement (Figure 5.11b).

MongoDB on Different Filesystems

To see how data applications can benefit from LinnOS, we set up a local MongoDB replica set on top of our three enterprise drives with homogeneous filesystem settings. For each type of filesystem, MongoDB receives 120K random read requests, and all drives run Microsoft traces

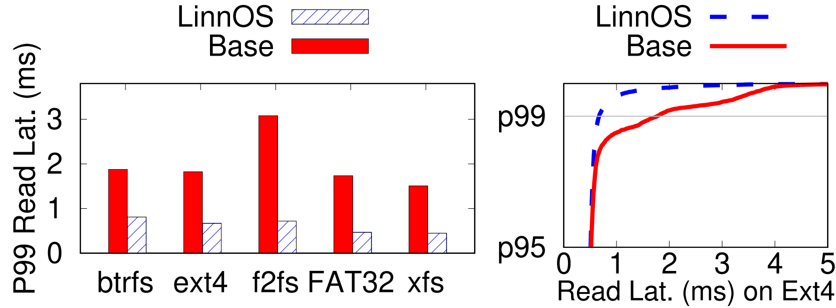

Figure 5.12: **MongoDB on different filesystems.** *This figure shows that LinnOS can easily help data applications achieve more predictable latency (Section 5.3.6).*

as background noise when serving MongoDB requests as latency-critical I/Os. Here, we focus on high-percentile latency (*e.g.*, p99 latency) since the average latency is largely impacted by filesystem buffering, while the tail latency reflects the raw performance from the devices.

Figure 5.12 shows that with LinnOS, MongoDB achieves much more predictable performance. For example, with all underlying devices formatted with f2fs, LinnOS reduces the p99 latency by 76.7%. Moreover, LinnOS only requires minor changes to MongoDB and filesystems: 50 additional LOC. For example, the filesystems should directly return LinnOS's error code to the applications instead of conducting unnecessary self-checking, and MongoDB needs to be slightly modified to reuse its built-in failover logic.

Computation Overhead/Optimization

CPU overhead. A reasonable concern is that if the entire OS has many neural networks, then it will be CPU-intensive. Across all the benchmarks and SSDs, paired with a lightweight neural network, each device only costs 0.3-0.7% of the host CPU resource, making LinnOS practical for large-scale deployments.

Co-processors for acceleration. As mentioned in Section 5.2.5, additional processors can be utilized to speed up the inference. By utilizing one more CPU core, LinnOS can reduce the inference overhead by 36% (to 4µs), with the maximal CPU usage increased up to 1.4% per device.

5.4 Conclusion and Discussions

We have presented LinnOS, to the best of our knowledge, the first operating system capable of inferring the speed of every I/O to flash storage. We have shown the feasibility of using a light neural network in the operating system for making frequent, fine-grained, black-box live inferences. LinnOS outperforms many other methods and successfully brings predictability on unpredictable flash storage. We also believe that LinnOS's success leads to exciting discussions and questions that can spur more future work:

On performance. Though LinnOS inference overhead (4-6μs) is less noticeable compared with the access latency of current SSDs (*e.g.*, 80μs), it could become problematic as SSDs march to 10μs latency range. Also, the consumption of computation resources can increase substantially as the IOPS grow. How to further lower the inference cost (*e.g.*, to 1μs) to support faster devices and higher throughput? Can advanced accelerators help accelerate OS kernel operations? Can near-storage/data processing help? Can we skip the inference when the outcome is highly assured (*e.g.*, the queue length is very low)? Can we cache the approximation results for popular predictions?

On masking the inaccuracy of machine learning. As machine learning (*e.g.*, LinnOS) can never achieve 100% accuracy, how should "ML-for-system" solutions mask the cases that machine learning fails to catch, while still benefiting from its generality? Is marrying learning and heuristic (*e.g.*, as in `LinnOS+HL`) a powerful option that exploits the advantages of both worlds?

On other integrations and extensions. One interesting question raised by LinnOS is why the latency behavior of SSDs—devices with complex idiosyncrasies—can be learned by the block layer with a few observable features. Understanding this can help other higher layers such as RAID, direct device access (SPDK), user/device-level filesystems, or distributed storage adopt our concept. Likewise, in lower layers, it is also a possibility in the future to have SSDs with latency inference capability built in. Although, arguably, one can say that the device already has a full knowledge of its internals and does not need a black-box prediction, an argument can be made that

SSD vendors can use the same machine learning method across different internal architectures that they have. Hence, they do not need to re-develop the inference logic every time they modify the internal hardware, logic, and policies. Alternatively, SSD vendors can employ "graybox" learning that incorporates some of the internal knowledge.

On precision. Can fast/slow inference be converted to a more precise latency inference, such as latency ranges (*e.g.*, 2-4μs, 4-8μs, ...), percentile buckets (*e.g.*, p0-p10, ..., p90-p100), or precise latency with high accuracy? Can model permutation or other machine learning techniques help?

CHAPTER 6

OTHER STORAGE WORK

Here we briefly describe some of our other storage-related works, including LeapIO (Section 6.1), FEMU (Section 6.2), TINYTAILFLASH (Section 6.3), and FAILSLOWATSCALE (Section 6.4).

6.1 LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs

To satisfy customer needs, today's cloud providers must implement a wide variety of storage (drive-level) functions as listed in Table 6.1. Providers must support both local and remote isolated virtual drives with IOPS guarantees. Users also demand drive-level atomicity/versioning, and not to mention other performance, reliability, and space-related features (checksums, deduplication, elastic volumes, encryption, prioritization, polling for ultra-low latencies, striping, replication, etc.) that all must be composable. As a result of these requirements, the cloud storage stack is extremely resource hungry, burning 10-20% of datacenter x86 cores, a major "storage tax" that cloud providers must pay.

Increasing prevalence of IO acceleration technologies such as SmartSSDs [47, 50], SmartNICs [45, 48] and custom IO accelerators with attached computation that can offload some functionality from the host CPU and reduce the heavy tax burden. However, IO accelerators do not provide an end-to-end solution for offloading real-deployment storage stacks. Today, the storage functions in Table 6.1 cannot be fully accelerated in hardware for three reasons: (1) the functionalities are too complex for low-cost hardware acceleration, (2) acceleration hardware is typically designed for common-case operations but not end-to-end scenarios, or (3) the underlying accelerated functions are not composable.

We present LeapIO [181], a new cloud storage stack that leverages ARM-based co-processors to offload complex storage services. LeapIO addresses many deployment challenges, including:

ⓐ **Fungibility and portability:** We need to keep servers fungible regardless of their acceler-

Local/remote *virtual* SSDs/services and caching. SR-IOV SSDs (hardware-assisted IO virtualization) do not have access to host DRAM. Thus local SSD caching for remote storage protocols (*e.g.* iSCSI [49], NVMeoF [17]) cannot be offloaded easily from x86.
Atomic write drive. Smart transactional storage devices [201, 226] do not provide atomicity across replicated drives/servers.
Versioned drive. A multi-versioned drive that allows writers to advance versions via atomic writes while the readers can stick to older versions, not supported in today's smart drives.
Priority virtual drive. Requires IO scheduling on every IO step (*e.g.*, through SSDs/NICs) with *flexible* policies, hard to achieve in hardware-based policies (*e.g.*, SSD-level prioritization).
Spillover drive. Uses few GBs of a local virtual drive and spills the remaining over to remote drives or services (elastic volumes), a feature that must combine local and remote virtual drives/services.
Replication & distribution. Accelerated cards can offload consistent and replicated writes, but they typically depend on a particular technology (*e.g.* non-volatile memory).
Other functionalities. Compression, deduplication, encryption, etc. must be composable with the above drives, not achievable in custom accelerators.

Table 6.1: **Real storage functions, not offload ready.** *The table summarizes why real cloud drive services are either not completely offload ready or not easily composable with each other.*

ation/offloading capabilities. That is, we treat accelerators as *opportunities* rather than necessities. In LeapIO, the storage software stack is portable – able to run on x86 *or* in the SoC whenever available (*i.e.*, "offload ready") as Figure 6.1a illustrates. The user/guest VMs are agnostic to what is implementing the virtual drive.

Fungibility and portability prevent "fleet fragmentation." Different server generations have different capabilities (*e.g.*, with/without ARM SoC, RDMA NICs or SR-IOV support), but newer server generations must be able to provide services to VMs running on older servers (and vice versa). Fungibility also helps provisioning; if the SoC cannot deliver enough bandwidth in peak moments, some services can borrow the host x86 cores to augment a crowded SoC.

ⓑ **Virtualizability and composability:** We need to support virtualizing and composing of, not just local/remote SSDs, but also local/remote IO *services* via NVMe-over-PCIe /RDMA/TCP/ REST. With LeapIO runtime, as depicted in Figure 6.1b, a user can obtain a local virtual drive that is mapped to a portion of a local SSD that at the same time is also shared by another remote service that glues multiple virtual drives into a single drive (*e.g.*, RAID). A storage service can be

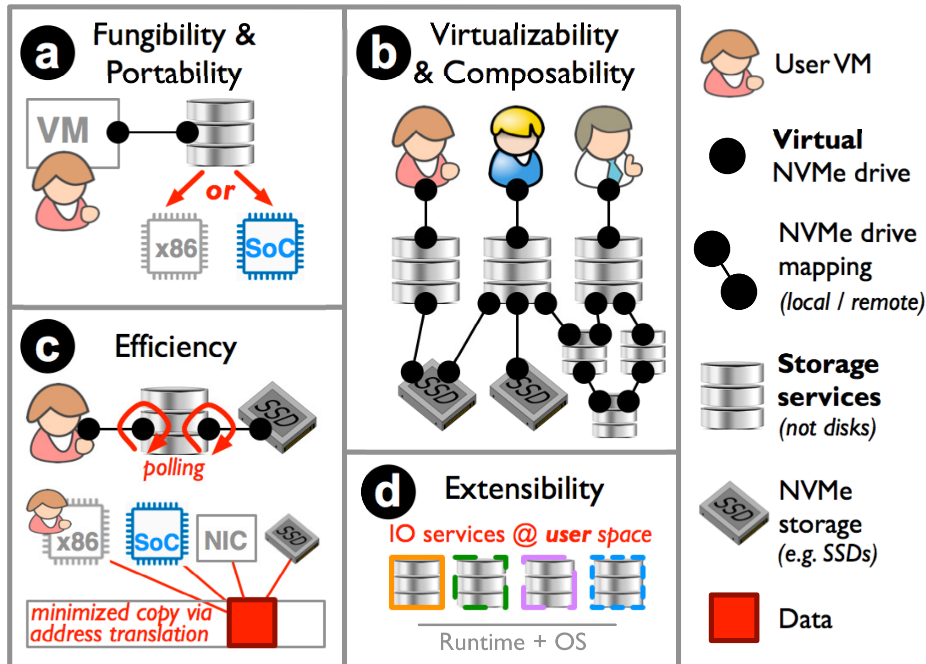

Figure 6.1: **Goals.**

composed on top of other remote services.

ⓒ **Efficiency:** It is important to deliver performance close to bare metal. LeapIO runtime must perform continuous *polling* on the virtual NVMe command queues as the proxy agent between local/remote SSD and services. Furthermore, ideally services must *minimize data movement* between different hardware/software components of a machine (on-x86 VMs, in-SoC services, NICs, and SSDs), which is achieved by LeapIO's uniform address space (Figure 6.1c).

ⓓ **Service extensibility:** Finally, unlike traditional block-level services that reside in the kernel space for performance, or FPGA-based offloading which is difficult to program, LeapIO enables storage services to be implemented at the *user space* (Figure 6.1d), hence allowing cloud providers to easily manage, rapidly build, and communicate with a variety of (trusted) complex storage services.

To address above deployment goals in a holistic way, LeapIO employs a set of OS/software techniques on top of new hardware capabilities, allowing storage services to portably leverage ARM co-processors. LeapIO helps cloud providers cut the storage tax and improve utilization

without sacrificing performance.

At the abstraction level we use NVMe, “the new language of storage” [17, 46]. All involved software layers from guest OSes, LeapIO runtime, to new storage services/functions all see the same device abstraction: *virtual NVMe drive*. They all communicate via the mature NVMe *queue-pair* mechanism accessible via basic memory instructions pervasive across x86 and ARM, QPI or PCIe.

On the software side, we build a runtime that hides the NVMe mapping complexities from storage services. Our runtime provides a *uniform address space across the x86 and ARM cores*, which brings two benefits.

First, our runtime *maps NVMe queue pairs across hardware/software boundaries* – between guest VMs running on x86 and service code offloaded to the ARM cores, between client- and server-side services, and between all the software layers and backend NVMe devices (*e.g.*, SSDs). Storage services can now be written in *user space* and be *agnostic* about whether they are offloaded or not.

Second, our runtime provides an *efficient data path* that alleviates unnecessary copying across the software components via *transparent address translation* across multiple address spaces: guest VM, host, co-processor user and kernel address spaces. The need for this is that while ARM SoC retains the computational generality of x86, it does not retain the peripheral generality that would allow different layers access the same data from their address spaces.

The runtime features above cannot be achieved without *new hardware support*. We require four new hardware properties in our SoC design: host DRAM access (for NVMe queue mapping), IOMMU access (for address translation), SoC’s DRAM mapping to host address space (for efficient data path), and NIC sharing between x86 and ARM SoC (for RDMA purposes). All these features are addressable from the SoC side; no host-side hardware changes are needed.

Storage services on LeapIO are “offload ready;” they can portably run in ARM SoC or on host x86 in a trusted VM. The software overhead only exhibits 2-5% throughput reduction compared to

bare-metal performance (still delivering 0.65 million IOPS on a datacenter SSD). Our current SoC prototype also delivers an acceptable performance, 5% further reduction on the server side (and up to 30% on the client) but with more than 20× cost savings.

6.2 The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator

Cheap and extensible research platforms are a key ingredient in fostering wide-spread SSD research. SSD simulators such as DiskSim's SSD model [57], FlashSim [127] and SSDSim [137], despite their popularity, only support internal-SSD research but not kernel-level extensions. On the other hand, hardware research platforms such as FPGA boards [216, 237, 279], OpenSSD [24], or OpenChannel SSD [74], support full-stack software/hardware research but their high costs (thousands of dollars per device) impair large-scale SSD research.

This leaves software-based emulator such as QEMU-based VSSIM [275], FlashEm [284], and LightNVM's QEMU [18], as the cheap alternative platform. Unfortunately, the state of existing emulators is bleak; they are either outdated, non-scalable, or not open-sourced.

To this end, we present FEMU [182], a QEMU-based flash emulator, with the following four "CASE" benefits.

First, FEMU is **c**heap ($0) as it will be an open-sourced software. FEMU has been successfully used in several projects, some of which appeared in top-tier OS and storage conferences [129, 270]. We hope FEMU will be useful to broader communities.

Second, FEMU is (relatively) **a**ccurate. For example, FEMU can be used as a drop-in replacement for OpenChannel SSD; thus, future research that extends LightNVM [74] can be performed on top of FEMU with relatively accurate results (*e.g.*, 0.5-38% variance in our tests). With FEMU, prototyping SSD-related kernel changes can be done without a real device.

Third, FEMU is **s**calable. As we optimized the QEMU stack with various techniques, such as exitless interrupt and skipping QEMU AIO components, FEMU can scale to 32 IO threads and still

achieve a low latency (as low as 52µs under a 2.3GHz CPU). As a result, FEMU can accurately emulate 32 parallel channels/chips, without unintended queueing delays.

Finally, FEMU is extensible. Being a QEMU-based emulator, FEMU can support internal-SSD research (only FEMU layer modification), kernel-only research such as software-defined flash (only Guest OS modification on top of unmodified FEMU), and split-level research (both Guest OS and FEMU modifications). FEMU also provides many new features not existent in other emulators, such as OpenChannel and multi-device/RAID support, extensible interfaces via NVMe commands, and page-level latency variability.

6.3 Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs

The core problem of flash performance instability is the well-known and "notorious" *garbage collection* (GC) process. A GC operation causes long delays as the SSD cannot serve (blocks) incoming I/Os. Due to an ongoing GC, read latency variance can increase by 100× [27, 99].

We address this urgent issue with "tiny-tail" flash drive (TINYTAILFLASH [270]), a GC-tolerant SSD that can deliver and guarantee stable performance. The goal of TINYTAILFLASH is to eliminate GC-induced tail latencies by *circumventing GC-blocked I/Os*. That is, ideally there should be *no* I/O that will be blocked by a GC operation, thus creating a flash storage that behaves close to a "no-GC" scenario. The key enabler is that SSD internal technology has changed in many ways, which we exploit to build novel GC-tolerant approaches.

Specifically, there are three major SSD technological advancements that we leverage for building TINYTAILFLASH. First, we leverage the increasing power and speed of today's flash controllers that *enable more complex logic* (*e.g.*, multi-threading, I/O concurrency, fine-grained I/O management) to be implemented at the controller. Second, we exploit the use of Redundant Array of Independent NAND (RAIN). Bit error rates of modern SSDs have increased to the point that ECC is no longer deemed sufficient [143, 163, 236]. Due to this increasing failure, modern

commercial SSDs employ parity-based redundancies (RAIN) as a standard data protection mechanism [30, 35]. By using RAIN, we can *circumvent GC-blocked read I/Os with parity regeneration*. Finally, modern SSDs come with a large RAM buffer (hundreds of MBs) backed by "super capacitors" [33, 38], which we leverage to *mask write tail latencies* from GC operations.

The timely combination of the technology practices above enables four new strategies in TINYTAILFLASH: **(a)** *plane-blocking GC*, which shifts GC blocking from coarse granularities (controller/channel) to a finer granularity (plane level), which depends on intra-plane copyback operations, **(b)** *GC-tolerant read*, which exploits RAIN parity-based redundancy to proactively generate contents of read I/Os that are blocked by ongoing GCs, **(c)** *rotating GC*, which schedules GC in a rotating fashion to enforce at most one active GC in every plane group, hence the guarantee to always cut "one tail" with one parity, and finally **(d)** *GC-tolerant flush*, which evicts buffered writes from capacitor-backed RAM to flash pages, free from GC blocking.

We first implemented TINYTAILFLASH in SSDSim [137] in order to simulate accurate latency analysis at the device level. Next, to run real file systems and applications, we also port TINYTAILFLASH to a newer QEMU/KVM-based platform based on VSSIM [275].

With a thorough evaluation, we show that TINYTAILFLASHsuccessfully eliminates GC blocking for a significant number of I/Os, reducing GC-blocked I/Os from 2–7% (base case) to only 0.003–0.7%. As a result, TINYTAILFLASH reduces tail latencies dramatically. Specifically, between the 99–99.99thpercentiles, compared to the perfect no-GC scenario, a base approach suffers from 5.6–138.2× GC-induced slowdowns. TINYTAILFLASH on the other hand is only 1.0 to 2.6× slower than the no-GC case, which confirms our near-complete elimination of GC blocking and the resulting tail latencies.

We also show that TINYTAILFLASH is more stable than state-of-the-art approaches that reduce GC impacts such as preemptive GC [32, 179]. Specifically, TINYTAILFLASH continuously delivers stable latencies while preemptive GC exhibits latency spikes under intensive I/Os. Furthermore, we contrast the fundamental difference of GC-impact elimination from reduction. In summary, by

leveraging modern SSD internal technologies in a unique way, we have successfully built novel features that provide a robust solution to the critical problem of GC-induced tail latencies.

6.4 Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems

FAILSLOWATSCALE [126] highlights an under-studied "new" failure type: *fail-slow hardware* – hardware that is still running and functional but in a degraded mode, slower than its expected performance. We found that all major hardware components can exhibit fail-slow faults. For example, disk throughput can drop by three orders of magnitude to 100 KB/s due to vibration, SSD operations can stall for seconds due to firmware bugs, memory cards can degrade to 25% of normal speed due to loose NVDIMM connection, CPUs can unexpectedly run in 50% speed due to lack of power, and finally network card performance can collapse to Kbps level due to buffer corruption and retransmission.

Unfortunately, fail-slow hardware is under-studied. To fill the void of strong evidence of hardware performance faults in the field, we, together with a group of researchers, engineers, and operators of large-scale datacenter systems across 12 institutions, present FAILSLOWATSCALE. More specifically, we have collected 101 detailed reports of fail-slow hardware behaviors including the hardware types, root causes, symptoms, and impacts to high-level software, with our unique and important findings in Table 6.2. To the best of our knowledge, this is the most complete account of fail-slow hardware in production systems reported publicly.

Important Findings and Observations
Varying root causes: Fail-slow hardware can be induced by internal causes such as firmware bugs or device errors/wear-outs as well as external factors such as configuration, environment, temperature, and power issues.
Faults convert from one form to another: Fail-stop, -partial, and -transient faults can convert to fail-slow faults (*e.g.*, the overhead of frequent error masking of corrupt data can lead to performance degradation).
Varying symptoms: Fail-slow behavior can exhibit a permanent slowdown, transient slowdown (up-and-down performance), partial slowdown (degradation of sub-components), and transient stop (*e.g.*, occasional reboots).
A long chain of root causes: Fail-slow hardware can be induced by a long chain of causes (*e.g.*, a fan stopped working, making other fans run at maximal speeds, causing heavy vibration that degraded the disk performance).
Cascading impacts: A fail-slow hardware can collapse the entire cluster performance; for example, a degraded NIC made many jobs lock task slots/containers in healthy machines, hence new jobs cannot find enough free slots.
Rare but deadly (long time to detect): It can take hours to months to pinpoint and isolate a fail-slow hardware due to many reasons (*e.g.*, no full-stack visibility, environment conditions, cascading root causes and impacts).
Suggestions
To vendors: When error masking becomes more frequent (*e.g.*, due to increasing internal faults), more explicit signals should be thrown, rather than running with a high overhead. Device-level performance statistics should be collected and reported (*e.g.*, via S.M.A.R.T) to facilitate further studies.
To operators: 39% root causes are external factors, thus troubleshooting fail-slow hardware must be done online. Due to the cascading root causes and impacts, full-stack monitoring is needed. Fail-slow root causes and impacts exhibit some correlation, thus statistical correlation techniques may be useful (with full-stack monitoring).
To systems designers: While software systems are effective in handling fail-stop (binary) model, more research is needed to tolerate fail-slow (non-binary) behavior. System architects, designers and developers can fault-inject their systems with all the root causes reported in this paper to evaluate the robustness of their systems.

Table 6.2: **Summary of our findings and suggestions.**

CHAPTER 7

CONCLUSION AND FUTURE WORK

Storage is important as we have big data. Furthermore, consumers are not only addicted to data, but also more of it in real-time, which is estimated to constitute nearly 30% of our data access by 2025. This surging challenge of serving data in real-time highlights the need for low and stable latencies, which is becoming a more significant challenge as we adopt faster I/O devices with milli/micro-second level access latencies. To tackle this challenge, this dissertation raises multiple approaches, including data, hardware-level, OS-level, and ML-for-system approaches. This chapter concludes this dissertation work and proposes future work in building fast and stable data and storage systems in this milli/micro-second era.

7.1 Conclusion

Though we have a straightforward goal that every data access is rapid, and our systems are stable and tail-free, the goal is difficult to achieve because modern systems have multiple layers with complex hardware and software components. Considering that, this dissertation aims to achieve this goal with multiple approaches.

Our data approach answers two key questions: Does performance instability exist? And how severe is it? Here, we conduct TAILATSTORE – the first, largest study on latency instability in storage devices. This study covers performance log in more than 450 thousand hard drives and four thousand SSDs for totally around 900 million drive hours, and mainly reveals three observations. First, disks have latency tails, and SSDs are even worse. Second, this instability is mostly caused by resource contentions and internal complexities. Third, as systems adopt faster storage devices, tail latencies are only becoming more prevalent, intrinsic, and heavy.

As we see heavy tail latencies in hardware, we take hardware-level approach and try to make our hardware "tail-free". Here, in TINYTAILFLASH, we successfully re-architect SSDs that promptly

utilize internal data redundancy to reconstruct the slow IOs, so these devices have no slow IOs and show stable and "tiny-tail" latencies, highlighting the effectiveness of hardware-level solutions.

Though hardware-level approach can dramatically improve tail performance, it requires specialized hardware and is ignorant about the global setup. To cover more hardware and application settings, we take an OS-level approach and develop MITTOS, a datacenter-aware operating system that advocates performance transparency. With MITTOS, applications can know the expected latency of every single I/O and take instant failovers to less-busy replicas with minor cost, outperforming the latest industry solutions.

MITTOS promotes transparency but it does not support all SSDs. To generically cover major commercial SSDs, we leverage ML-for-system approach. As nowadays systems receive millions and billions of I/Os per second, the I/Os themselves assemble a valuable dataset, which gives machine learning a great asset to learn. Here, we develop LINNOS. By learning the I/Os with a lightweight neural network, LINNOS can predict tail latency occurrences for every single IO, on most SSDs, achieve great accuracies on industrial workloads, and brings low overhead on CPUs.

With all these approaches at different levels, we build "tail-free" data and storage systems at milli/micro-second scale.

7.2 Future Work

The initial success in LINNOS advocates the call for more "AI for storage" research, which is raised in the Data Storage Research Vision 2025 [61, Section 3.3]. Indeed, we believe this research space is timely for the following reasons. First, the storage stack is full of policies (decision makings) from allocation, mapping, scheduling, to rerouting that potentially can be further reinforced with learning techniques. Second, storage systems serve diverse workloads and run on various devices and models where machine learning's property of generality can help. Third, with millions-of-IOPS per-node/device capability, the storage stack has its own "natural" data lake that will allow learning algorithms to be trained and insights to be gained. Finally, the fast growth of GPU/tensor

devices and custom accelerators will speedup training and inference, making learning algorithms feasible to deploy in high-throughput, low-latency storage stack.

Given this, we propose future work to explore the power of machine learning as well as its limits and overheads in improving many aspects of the storage stack (performance, quality of service, lifetime, and management complexity), as illustrated below.

Blackbox Learning of SSD Anomalies

Our initial success shows the efficacy of using a light neural network for blackbox, fine-grained and live learning of I/O latencies. However, there are more challenges to address: **(a)** *Accuracy and inference speed:* We can continue improving the model's accuracy without increasing the inference time (by reducing unnecessary features and adding more impactful ones). We can develop techniques that speed up inference (*e.g.*, by pruning small weights that have little or no impact on the inference). Increasing accuracy while decreasing inference time is a challenging task as shown in Table 5.3; other models we built, "A" and "B," reduces inference time but decreases accuracy while models "C" to "E" deliver the opposite. **(b)** *Extending to latency-range inference:* While we have an early success with binary (fast/slow) decision, we can address the challenges of linear and exponential classifications; we already tried out different output labeling, linear (0-10μs, 10-20μs, ...) and exponential (2-4μs, 4-8μs, ...), and the accuracy only reached 60-70%. A successful extension can allow applications to take more flexible actions. **(c)** *Various anomaly mitigations:* We then can build latency-range predictions and integrate them with I/O scheduling policies that not only mitigate anomalies but also optimize SLO and fairness [184, 218]; for example, policies such as Fair-EDF [218] require an underlying OS that can predict latency violation in advanced. **(d)** *Beyond the OS:* To support direct storage access outside the kernel, we can cover anomalies in modern I/O libraries such as SPDK, DPDK and RDMA [6, 22]; for example, in LeapIO [181], which utilizes a new deep storage stack (from guest VM, RDMA-over-SmartNIC, to server-side SoC-based storage), a basic read-only test already exhibits a large latency tail (5x slowdown at

$N_{1.6T}S$	40.25M	$N_{500G}S$	2M
$N_{128G}S$	1M	$N_{2T}I$	11.5M
$N_{1.6T}I$	11.5M	$N_{1T}I$	11M
$N_{1.6T}M$	15M	$A_{800G}G$	3.75M
$T_{200G}S$	1M	$T_{128G}S$	1M
$T_{100G}S$	512K	$T_{64G}S$	4M
$T_{200G}M$	64M		

Table 7.1: **Write buffer sizes.**

p95).

Graybox Learning of SSD Anomalies

In LINNOS, the blackbox learning component predicts only one property of the SSD (the resulting I/O latency), and the blackbox management component (the admission control) can work on many commodity SSDs. Here, we propose expanding the learning part from a black- to "gray"-box way. In MITTOS, we show that we can see anomalies in advance if we have knowledge of the SSD internal properties, which was a success because we used a host-managed OpenChannel ("white-box") SSD [42]. The question is then whether we can extract the internal properties of commodity SSDs. Fortunately, classical works on disk drives/arrays show that by *probing* the storage with carefully crafted workloads, device users can *reverse engineer* the observed latencies to extract the internal disk properties [102, 232, 259, 276]. For SSDs, some recent efforts have been made [83, 160, 165–167], but individually each of these efforts only extracts 1 to 3 internal properties and uses the extracted SSD properties (*e.g.*, #channels).

We can **(a)** build an exhaustive prober that can extract more than a dozen of SSD internal properties (hence, a "graybox" method) by probing injects carefully crafted read/write mixed workloads and analyzing the measured latencies using statistical methods, and then **(b)** build a flash-learning file system that uses the extracted properties to manage the performance anomalies of the SSDs.

To show feasibility, our initial prober has extracted nine properties on 21 consumer and enterprise models from 7 vendors (1 billion of latency values were analyzed). Table 7.1 shows one

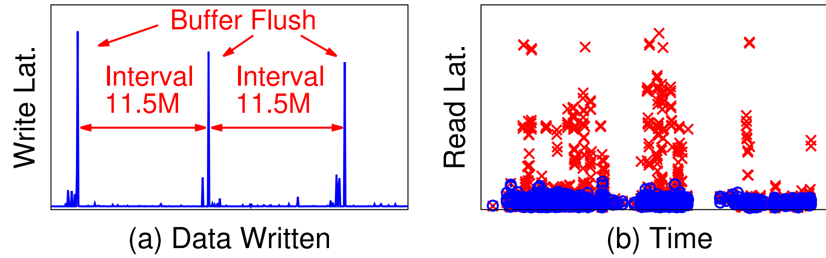

Figure 7.1: **Buffer learning and management.**

property, the internal write buffer size, that we have successfully extracted on 14 SSD models (anonymized per vendors' request). When the probed device flushes its buffer, it will cause write latency spikes, as shown in Figure 7.1a for device $N_{2T}I$. Here, the amount of data written between the latency spikes (in this case, 11.5MB) hints the size of the write buffer. To get the answer without manual efforts, we supply the data to Jenks natural breaks [10], a learning method on 1-dimensional latency data. If the first answer (*e.g.*, "11.5MB") has a significantly higher confidence value than the rest, the method successfully guesses the answer.

The red "$\times$s" in Figure 7.1b depicts the read latency spikes on device $N_{2T}I$ under a particular workload. These spikes can be avoided if the file system manages the write flow according to the device's internal buffer size. Thus, we design such logic in a user-level shim layer (for prototyping only) that rate-limits the application's write flush accordingly, whenever possible. We ran Microsoft Bing SSD-level block traces on the same device with the shim layer activated, and the improved result is depicted by the more stable blue dots in Figure 7.1b.

Potential Extensions. Potential extensions as follows. **(a)** *Complete the buffer flush learning.* On the learning side, we found more complex SSDs that perform 2-level buffering ("small+big" flushes) and background flushes (not shown in Table 7.1). These require statistical methods that can learn multi-dimensional data such as KNN [12], K-Means [11], and Affinity Propagation [1]. **(b)** *Complete the buffer flush management.* We can integrate our shim-layer algorithm into the file system layer, evaluate it on the 21 SSD models that we have, enhance the rate-limiting algorithm to include multi-dimensional buffering, and address other limitations such as handling write bursts.

(c) *Build a more intelligent file system.* We can design a more complete flash-learning file system that employs as many extracted properties as possible. For example, we also have successfully extracted the parallelism properties such as the internal chunk size (KB), stripe width (#pages), #channels, and #chips/channel (not shown), and found that many modern SSDs employ a static mapping to increase more parallelism (*e.g.*, LBNs are striped across the chips). Our file system can leverage these features to perform a device-level isolation (*e.g.*, reroute specific LBNs to specific chips) even without device isolation interfaces. This will be unique to existing works that depend on interfaces such as `SET`, `ZNS`, and OpenChannel [79, 140, 168, 176, 220].

Framework to Install Learning

It is likely possible that no single model can learn all devices/workloads. In this context, the networking community has proposed a framework that allows network administrators to deploy different congestion control learning models [210]. So far it works for medium/long-term decisions (*e.g.*, the congestion control parameters are not frequently updated). Likewise, we can explore how to build a framework where storage administrators can deploy and frequently update different learning models in the storage stack. For short-term decisions, inference speed matters. Hence, we can explore the use of knowledge distillation (model compression method) [13, 34, 78] where a small model (faster inference) deployed in the storage stack is trained to mimic a pre-trained, larger model (higher accuracy).

Another interesting topic here is how to seamlessly renew installed learning models. As workload fluctuations are common in storage services, models trained from data collected during one period typically cannot fit all scenarios. In many cases, models installed inside our storage stack will need to be renewed periodically. Typically, this will require a re-training based on newly collected data, which sounds straightforward but can be tricky at some points, as the current models can “pollute” the data footprint. For example, in LinnOS, when the models are active, the observable latency distributions will change as slow I/Os are revoked. In this case, the collected data is

inappropriate for re-training as it no longer reflects the raw latency behaviors of the devices. One alternative is to turn off the models during data re-collection. However, this will interrupt all policies built upon these models, which can bring significant downtime. Considering this, we believe that handling seamless recalibration is highly important for the learning installation framework.

Using Learning to Discover Surprising Design Space

One reason that machine learning can help build effective and generic solutions is its large space of configuration parameters, which are typically auto-tuned by training frameworks. For example, in LinnOS, a lightweight neural network with only one hidden layer already contains over eight thousand parameters – a huge space that human designers can rarely cover, even with their deep expertise. As in current system solutions the output labels – what the heuristics/learning will predict, the action schemes – what actions will systems take based on the output labels, and the input feature settings – what the heuristics/learning will take as available information, are still heavily shaped by designers' cognition with "foreseeable" number of dimensions, one potential topic is to explore whether learning can its large parameter/dimension coverage to help us discover surprising design spaces. For example, can learning help us locate useful while underrated features that are out of our general scope of expertise, or figure out the action schemes that we have neglected? Successful efforts on this aspect may open more design space to leverage and provide more opportunities for "auto-designed" systems.

Accelerators for Learning-Powered Storage

In the integration where learning in every layer is needed to make live and short-term answers and decisions, it will cause a heavy tax on the CPUs and potentially a long chain of inference overhead. Technology trend suggests that 100-200x improvement on inference latency can be foreseen in the near future with more advanced hardware innovations [16], which can make ML-for-storage solutions more deployable in the future. Until this technology arrives, we can explore

cheap, custom accelerators that can be used and shared by multiple learning layers in the storage stack. We can browse the literature of accelerators for machine learning purposes [66, 87, 152, 213, 214, 256, 258]; for example, an ASIC-based binary/quantized neural network accelerator [213] can power our light neural network.

I/O Latency Dataset (Benchmark)

"Benchmarks shape a field" [217]. In the ML/visualization community, the large ImageNet benchmarks have spurred research in image recognition. Similarly, we can provide benchmarks for fostering storage research in ML-based per-IO latency prediction. In the storage community, workload benchmarks are available in the form of traces [9, 21, 31, 94], but to evaluate whether a predictor is accurate, one must run the traces through the predictor on *many* varieties of storage devices, which not all researchers have access to. Thus, we can assemble a large benchmark for evaluating prediction models (and make it publicly accessible) with help from our industry partners.

REFERENCES

[1] Affinity Propagation. https://en.wikipedia.org/wiki/Affinity_propagation.

[2] Cassandra - Speculative Execution for Reads / Eager Retries. https://issues.apache.org/jira/browse/CASSANDRA-4705.

[3] Cassandra Snitches. https://docs.datastax.com/en/archived/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html.

[4] CFQ (Complete Fairness Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[5] Data Age 2025: The Digitization of the World from Edge to Core. https://www.seagate.com/files/www-content/our-story/trends/files/dataage-idc-report-final.pdf.

[6] DPDK: Data Plane Development Kit. https://www.dpdk.org.

[7] Ethernet: The High Bandwidth Low-Latency Data Center Switching Fabric. http://www.force10networks.com/whitepapers/pdf/F10_wp_Ethernet.pdf.

[8] GreyBeards on Storage. https://silvertonconsulting.com/gbos2/tag/tail-latency/.

[9] IOzone Filesystem Benchmark. http://www.iozone.org.

[10] Jenks Natural Breaks Optimization. https://en.wikipedia.org/wiki/Jenks_natural_breaks_optimization.

[11] K-Means Clustering. https://en.wikipedia.org/wiki/K-means_clustering.

[12] K-Nearest Neighbors Algorithm. https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm.

[13] Knowledge Distillation. https://nervanasystems.github.io/distiller/knowledge_distillation.html.

[14] ML-OS literature-review. https://bit.ly/2stI7hR.

[15] MongoDB - Basic Support for Operation Hedging in NetworkInterfaceTL. https://jira.mongodb.org/browse/SERVER-45432.

[16] New GraphCore IPU BenchMarks. https://www.graphcore.ai/posts/new-graphcore-ipu-benchmarks.

[17] NVM Express Base Specification 1.4. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[18] Open-Channel Solid State Drives. http://lightnvm.io/.

[19] Pareto Distribution. https://en.wikipedia.org/wiki/Pareto_distribution.

[20] SNIA I/O Trace Data Files. http://iotta.snia.org/traces.

[21] SNIA IOTTA: Storage Networking Industry Association's Input/Output Traces, Tools, and Analysis Trace Repository. http://iotta.snia.org.

[22] SPDK: Storage Performance Development Kit. https://spdk.io.

[23] The Evolution of Image Classification Explained. https://stanford.edu/~shervine/blog/evolution-image-classification-explained.

[24] The OpenSSD Project. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[25] Tuning Linux I/O Scheduler for SSDs. https://www.nuodb.com/techblog/tuning-linux-io-scheduler-ssds.

[26] Weak Head. http://forum.hddguru.com/search.php?keywords=weak-head.

[27] Google: Taming The Long Latency Tail - When More Machines Equals Worse Results. http://highscalability.com/blog/2012/3/12/google-taming-the-long-latency-tail-when-more-machines-equal.html, 2012.

[28] Personal Communication from Andrew Baptist of Cleversafe, 2013.

[29] Personal Communication from Garth Gibson of CMU, 2013.

[30] The Crucial M550 SSD. http://www.crucial.com/usa/en/storage-ssd-m550, 2013.

[31] Filebench. https://github.com/filebench/filebench/wiki, 2014.

[32] Sandisk: Pre-emptive garbage collection of memory blocks. https://www.google.com/patents/US8626986, 2014.

[33] Supercapacitors have the power to save you from data loss. http://www.theregister.co.uk/2014/09/24/storage_supercapacitors/, 2014.

[34] Geoffrey E. Hinton, Oriol Vinyals, Jeffrey Dean. https://arxiv.org/abs/1503.02531, 2015.

[35] Micron P420m Enterprise PCIe SSD Review. http://www.storagereview.com/micron_p420m_enterprise_pcie_ssd_review, 2015.

[36] Personal Communication from NetApp Hardware and Product Teams, 2015.

[37] The case against SSDs. http://www.zdnet.com/article/the-case-against-ssds/, 2015.

[38] What Happens Inside SSDs When the Power Goes Down? http://www.army-technology.com/contractors/data_recording/solidata-technology/presswhat-happens-ssds-power-down.html, 2015.

[39] Why SSDs don't perform. http://www.zdnet.com/article/why-ssds-dont-perform, 2015.

[40] Non-Volatile Memory (NVM) Market by Type (Electrically Addressed, Mechanically Addressed, and others), Applications (Consumer Electronics, Healthcare Monitoring Application, Automotive & Transportation Application, Enterprise Storage, Industrial) and Vertical (Telecom & IT, Healthcare, Automotive, Energy & Power, Manufacturing Industries) - Global Opportunity Analysis and Industry Forecasts, 2014 - 2022. https://www.alliedmarketresearch.com/non-volatile-memory-market, 2016.

[41] Non-Volatile Memory (NVM) Market to Reach $82 Billion by 2022. https://tinyurl.com/nvmeMarket82B, 2016.

[42] Open-Channel Solid State Drives. http://lightnvm.io/, 2016.

[43] Tuning Speculative Retries to Fight Latency. https://www.youtube.com/watch?v=uRJSuQofJWQ, 2016.

[44] Solid State Drives Market (SSD Market) by Form Factor (2.5", 3.5", M.2, U.2/SFF 8639, FHHL/HHHL), Interface (SATA, SAS, PCIe), Technology (SLC, MLC, TLC), End-user (Enterprise, Client, Industrial, Automotive), and Geography - Global Forecast to 2023. https://www.marketsandmarkets.com/Market-Reports/solid-state-drives-market-75076578.html, 2017.

[45] BlueField SmartNIC. http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic, 2018.

[46] NVMe Is The New Language Of Storage. https://www.forbes.com/sites/tomcoughlin/2018/05/03/nvme-is-the-new-language-of-storage, 2018.

[47] *Smart SSD: Faster Time To Insight*, 2018. https://samsungatfirst.com/smartssd/.

[48] Stingray 100GbE Adapter for Storage Disaggregation over RDMA and TCP. https://www.broadcom.com/products/storage/ethernet-storage-adapters-ics/ps1100r, 2018.

[49] iSCSI: Internet Small Computer Systems Interface. https://en.wikipedia.org/wiki/ISCSI, 2019.

[50] NGD Newport Computational Storage Platform. https://www.ngdsystems.com, 2019.

[51] $47+ Billion Solid State Drive (SSD) Markets, 2025: Declining Prices of SSDs, Performance Improvements due to Formation of NVMe, Surge in Cloud Customers. https://tinyurl.com/ssdMarket47B, 2020.

[52] Solid State Drive (SSD) Market - Growth, Trends, and Forecasts (2020 - 2025). https://www.mordorintelligence.com/industry-reports/solid-state-drive-market, 2020.

[53] Solid State Drive (SSD) Market revenue to cross USD 125 Bn by 2026. https://tinyurl.com/ssdMarket125B, 2020.

[54] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[55] Abutalib Aghayev and Peter Desnoyers. Skylight-A Window on Shingled Disk Operation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[56] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving Ext4 for Shingled Disks. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[57] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[58] Irfan Ahmad. Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2007.

[59] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[60] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic storage maintenance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[61] George Amvrosiadis, Ali R. Butt, Vasily Tarasov, Erez Zadok, Ming Zhao, Irfan Ahmad, Remzi H. Arpaci-Dusseau, Feng Chen, Yiran Chen, Yong Chen, Yue Cheng, Vijay Chidambaram, Dilma Da Silva, Angela Demke-Brown, Peter Desnoyers, Jason Flinn, Xubin He, Song Jiang, Geoff Kuenning, Min Li, Carlos Maltzahn, Ethan L. Miller, Kathryn Mohror, Raju Rangaswami, Narasimha Reddy, David Rosenthal, Ali Saman Tosun, Nisha Talagala, Peter Varman, Sudharshan Vazhkudai, Avani Waldani, Xiaodong Zhang, Yiying Zhang, and Mai Zheng. Data Storage Research Vision 2025: Report on NSF Visioning Workshop.

[62] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[63] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[64] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[65] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[66] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen mei W. Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[67] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[68] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, 2001.

[69] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

[70] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007.

[71] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[72] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communication of the ACM*, 60(4):48–54, 2017.

[73] Alex Beutel, Tim Kraska, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. A Machine Learning Approach to Databases Indexes. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS)*, 2017.

[74] Matias Bjorling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[75] Florian Brandherm, Lin Wang, and Max Muhlhauser. A Learning-based Framework for Optimizing Service Migration in Mobile Edge Clouds. In *Proceedings of the 2019 EuroSys Conference (EuroSys)*, 2019.

[76] Eric Brewer. Spinning Disks and Their Cloudy Future (Keynote). In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[77] Jake Brutlag. Speed Matters for Google Web Search. http://services.google.com/fh/files/blogs/google_delayexp.pdf, 2009.

[78] Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. Model Compression. In *Proceedings of the 12th SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, 2006.

[79] Tim Canepa. SSD Architecture for IO Determinism. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180807_ARCH-101-1_Canepa.pdf.

[80] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. Curator: Self-Managing Storage for Enterprise Clusters. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[81] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the Performance Variation in Modern Storage Stacks. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[82] Danilo Carastan-Santos and Raphael Y. de Camargo. Obtaining dynamic scheduling policies with simulation and machine learning. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[83] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA-17)*, 2011.

[84] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[85] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[86] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[87] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[88] Yanpei Chen, Sara Alspaugh, and Randy H. Katz. Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB)*, 2012.

[89] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[90] Chanwoo Chung, Jinhyung Koo, Junsu Im, Arvind, and Sungjin Lee. LightStore: Software-defined Network-attached Key-value Drives. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[91] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[92] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015.

[93] Maxime Colmant, Masha Kurpicz, Pascal Felber, Loic Huertas, Romain Rouvoy, and Anita Sobe. Process-level Power Estimation in VM-based Systems. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[94] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[95] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[96] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[97] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

[98] Jeff Dean. Machine Learning for Systems and Systems for Machine Learning (Invited Talk). In *ML Systems Workshop at NIPS 2017*, 2017.

[99] Jeffrey Dean and Luiz Andre Barroso. The Tail at Scale. *Communications of the ACM (CACM)*, 56(2), 2013.

[100] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[101] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[102] Timothy E. Denehy, John Bent, Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, 2004.

[103] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[104] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HardFS: Hardening HDFS with Selective and Lightweight Versioning. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[105] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[106] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, P. Brighten Godfrey, and Michael Schapira. PCC Vivace: Online-Learning Congestion Control. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[107] Mark Endrei, Chao Jin, Minh Ngoc Dinh, David Abramson, Heidi Poxon, Luiz DeRose, and Bronis R. de Supinski. Energy efficiency modeling of parallel applications. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.

[108] Benjamin Farley, Venkatanathan Varadarajan, Kevin Bowers, Ari Juels, Thomas Ristenpart, and Michael Swift. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, 2012.

[109] Michael Feldman. Startup Takes Aim at Performance-Killing Vibration in Datacenter. http://www.hpcwire.com/2010/01/19/startup_takes_aim_at_performance-killing_vibration_in_datacenter/, 2010.

[110] Tim Feldman and Garth Gibson. Shingled Magnetic Recording Areal Density Increase Requires New Data Management. *LOGIN*, 38(3):22–30, 2013.

[111] Robert Frickey. Data Integrity on 20nm SSDs. In *Flash Memory Summit*, 2012.

[112] Justin Funston, Maxime Lorrillere, Alexandra Fedorova, Baptiste Lepers, David Vengerov, Jean-Pierre Lozi, and Vivien Quema. Placement of Virtual Containers on NUMA systems: A Practical and Comprehensive Model. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[113] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[114] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[115] 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)*, 2017.

[116] Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. Improving backfilling by using machine learning to predict running times. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[117] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[118] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, 2009.

[119] Laura M. Grupp, John D. Davis, and Steven Swanson. The Bleak Future of NAND Flash Memory. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[120] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[121] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[122] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[123] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[124] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[125] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[126] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Symposium on File and Storage Technologies (FAST)*, 2018.

[127] 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)*, 2009.

[128] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[129] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[130] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[131] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[132] Robin Harris. Bad, bad, bad vibrations. http://www.zdnet.com/article/bad-bad-bad-vibrations/, 2010.

[133] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the 2017 EuroSys Conference (EuroSys)*, 2017.

[134] Jun He, Duy Nguyen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Reducing File System Tail Latencies with Chopper. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[135] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[136] Chin-Jung Hsu, Rajesh K Panta, Moo-Ryong Ra, and Vincent W. Freeh. Inside-Out: Reliable Performance Prediction for Distributed Storage Systems in the Cloud. In *The 35th Symposium on Reliable Distributed Systems (SRDS)*, 2016.

[137] 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 25th International Conference on Supercomputing (ICS)*, 2011.

[138] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, 2012.

[139] H. Howie Huang, Shan Li, Alexander S. Szalay, and Andreas Terzis. Performance Modeling and Analysis of Flash-based Storage Devices. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[140] 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 Symposium on File and Storage Technologies (FAST)*, 2017.

[141] Peng Huang, Chuanxiong Guo, Lindong Znhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randonph Yao. Gray Failure: The Achilles' Heel of Cloud Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)*, 2017.

[142] Amber Huffman. Addressing IO Determinism Challenges at Scale with NVM Express – Part 2: Renegotiating the Host/Device Contract. In *Proceedings of the 2017 Non-Volatile Memory Workshop (NVMW)*, 2017.

[143] Soojun Im and Dongkun Shin. Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD. In *IEEE Transactions on Computers (TC)*, 2011.

[144] Tanzima Zerin Islam, Jayaraman J. Thiagarajan, Abhinav Bhatele, Martin Schulz, and Todd Gamblin. A Machine Learning Framework for Performance Coverage Analysis of Proxy Applications. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[145] Sitaram Iyer and Peter Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[146] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[147] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy (SP)*, 2012.

[148] Nathan Jay, Noga H. Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.

[149] Yu Ji, Youhui Zhang, Wenguang Chen, and Yuan Xie. Bridging the Gap Between Neural Networks and Neuromorphic Hardware with A Neural Network Compiler. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[150] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A Flexible, High-Performance Key-Value SSD. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA-23)*, 2017.

[151] Michael I. Jordan. SysML: Perspectives and Challenges (Invited Talk). In *Proceedings of the 1st Conference on Machine Learning and Systems (MLSys)*, 2018.

[152] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. 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)*, 2017.

[153] Myoungsoo Jung, Wonil Choi, Miryeong Kwon, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Kandemir. Design of a Host Interface Logic for GC-Free SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 8(1), May 2019.

[154] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise

Clusters. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[155] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating Hardware Device Failures in Software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[156] Michael P. Kasick, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Black-Box Problem Diagnosis in Parallel File Systems. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[157] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[158] Bryan S. Kim, Hyun Suk Yang, and Sang Lyul Min. AutoSSD: an Autonomic SSD Architecture. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[159] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)*, 2008.

[160] Jae-Hong Kim, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs). In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009.

[161] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards SLO Complying SSDs Through OPS Isolation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[162] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Improving SSD Reliability with RAID via Elastic Striping and Anywhere Parity. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2013.

[163] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Improving SSD Reliability with RAID via Elastic Striping and Anywhere Parity. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2013.

[164] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[165] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-Aware I/O Management for Solid State Disks (SSDs). In *IEEE Transactions on Computers (TC)*, 2011.

[166] 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, 2018.

[167] Joonsung Kim, Pyeongsu Park, Jaehyung Ahn, Jihun Kim, Jong Kim, and Jangwoo Kim. SSDcheck: Timely and Accurate Prediction of Irregular Behaviors in Black-Box SSDs. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*, 2018.

[168] Seonbong Kim and Joon-Sung Yang. Optimized I/O Determinism for Emerging NVM-based NVMe SSD in an Enterprise System. In *Design Automation Conference (DAC)*, 2018.

[169] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *Proceedings of the 17th USENIX Symposium on File and Storage Technologies (FAST)*, 2019.

[170] Youngjae Kim, Junghee Lee, Sarp Oral, David A. Dillow, Feiyi Wang, and Galen M. Shipman. Coordinating Garbage Collection for Arrays of Solid-State Drives. *IEEE Transactions on Computers (TC)*, 63(4), April 2014.

[171] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-state Drives. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[172] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[173] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous Cloud Storage Configuration for Data Analytics. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[174] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring System Challenges of Ultra-Low Latency Solid State Drives. In *the 10th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[175] Elie Krevat, Joseph Tucek, and Gregory R. Ganger. Disks Are Like Snowflakes: No Two Are Alike. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, 2011.

[176] 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 *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*, 2020.

[177] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[178] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[179] 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 International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[180] Sungjin Lee, Ming Liu, SangWoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed Flash. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[181] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[182] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *Proceedings of the 16th USENIX Symposium on File and Storage Technologies (FAST)*, 2018.

[183] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[184] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. PSLO: Enforcing the Xth Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the 2016 EuroSys Conference (EuroSys)*, 2016.

[185] Shan Li and H. Howie Huang. Black-Box Performance Modeling for Solid-State Drives. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010.

[186] Zhao Lucis Li, Chieh-Jan Mike Liang, Wei Bai, Qiming Zheng, Yongqiang Xiong, and Guangzhong Sun. Accelerating Rule-matching Systems with Learned Rankers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[187] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural Packet Classification. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019.

[188] Greg Linden. Make Data Useful. https://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf, 2006.

[189] Greg Linden. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2006.

[190] Chun-Yi Liu, Jagadish B. Kotra, Myoungsoo Jung, Mahmut T. Kandemir, and Chita R. Das. SOML Read: Rethinking the Read Operation Granularity of 3D NAND SSDs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[191] Libin Liu and Hong Xu. Elasecutor: Elastic Executor Scheduling in Data Analytics Systems. In *Proceedings of the 9th ACM Symposium on Cloud Computing (SoCC)*, 2018.

[192] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical Disentanglement in a Container-Based File System. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[193] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David Nagle, and Erik Riedel. Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, 2000.

[194] Ao Ma, Fred Douglis, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[195] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based Memory Allocation for C++ Server Workloads. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[196] Martin Maas, Krste Asanovic, Tim Harris, and John Kubiatowicz. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[197] Adam Manzanares, Noah Watkins, Cyril Guyot, Damien LeMoal, Carlos Maltzahn, and Zvonimr Bandic. ZEA, A Data Management Approach for SMR. In *the 8th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[198] Aniruddha Marathe, Rushil Anirudh, Nikhil Jain, Abhinav Bhatele, Jayaraman J. Thiagarajan, Bhavya Kailkhura, Jae-Seung Yeom, Barry Rountree, and Todd Gamblin. Performance modeling under resource constraints using deep transfer learning. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[199] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming Performance Variability. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[200] Michael Mesnier, Jason B. Akers, Feng Chen, and Tian Luo. Differentiated Storage Services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[201] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, 2015.

[202] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[203] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. Hiding the Microsecond-Scale Latency of Storage-Class Memories with Duplexity. In *Proceedings of the 2019 Non-Volatile Memory Workshop (NVMW)*, 2019.

[204] Nikita Mishra, Connor Imes, John D. Laferty, and Henry Hoffmann. CALOREE: Learning Control for Predictable Latency and Low Energy. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[205] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[206] Pulkit A. Misra, María F. Borge, Iñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proceedings of the 2019 EuroSys Conference (EuroSys)*, 2019.

[207] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[208] Saurav Muralidharan, Amit Roy, Mary W. Hall, Michael Garland, and Piyush Rai. Architecture-Adaptive Code Variant Tuning. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[209] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[210] Akshay Narayan, Frank Cangialosi, Deepti Raghavan, Prateesh Goyal, Srinivas Narayana, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Restructuring Endpoint Congestion Control. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[211] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[212] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[213] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit K. Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC. In *2016 International Conference on Field-Programmable Technology*, 2016.

[214] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan J. M. Moss, Suchit Subhaschandra, and Guy Boudoukh. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[215] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[216] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage System. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[217] David Patterson. Technical Perspective: For Better or Worse, Benchmarks Shape a Field. *Communications of the ACM*, 55(7), 2012.

[218] Yuhan Peng and Peter Varman. Fair-EDF: A Latency Fairness Framework for Shared Storage Systems. In *the 11th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2019.

[219] Chris Petersen. Addressing IO Determinism Challenges at Scale with NVM Express. In *Proceedings of the 2017 Non-Volatile Memory Workshop (NVMW)*, 2017.

[220] Chris Petersen, Wei Zhang, and Alexei Naberezhnov. Enabling NVMe I/O Determinism at Scale. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180807_INVT-102A-1_Petersen.pdf.

[221] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[222] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[223] Rekha Pitchumani, James Hughes, and Ethan L. Miller. SMRDB: Key-Value Data Store for Shingled Magnetic Recording Disks. In *Proceedings of the 8th ACM International Conference on Systems and Storage (SYSTOR)*, 2015.

[224] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[225] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[226] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[227] Bryan Reinero. Selecting AWS Storage for MongoDB Deployments: Ephemeral vs. EBS. https://www.mongodb.com/blog/post/selecting-aws-storage-for-mongodb-deployments-ephemeral-vs-ebs.

[228] Viraf (Willy) Reporter. The Value of a Millisecond: Finding the Optimal Speed of a Trading Infrastructure. https://research.tabbgroup.com/report/v06-007-value-millisecond-finding-optimal-speed-trading-infrastructure, 2008.

[229] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[230] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. It's Time for Low Latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, 2011.

[231] Jiri Schindler and Gregory R. Ganger. Automated Disk Drive Characterization. In *Proceedings of the 2000 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2000.

[232] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST)*, 2002.

[233] Florian Schmidt, Mathias Niepert, and Felipe Huici. Representation Learning for Resource Usage Prediction. In *Proceedings of the 1st Conference on Machine Learning and Systems (MLSys)*, 2018.

[234] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[235] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST)*, 2007.

[236] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[237] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[238] Anand Lal Shimpi. Intel Discovers Bug in 6-Series Chipset: Our Analysis. http://www.anandtech.com/show/4142/intel-discovers-bug-in-6series-chipset-begins-recall, 2011.

[239] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[240] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[241] Changheng Song, Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Weihua Zhang. Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[242] Steve Souders. Velocity and the Bottom Line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html, 2009.

[243] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Uma Maheswara Rao G., and Haryadi S. Gunawi. PBSE: A Robust Path-Based Speculative Execution for Degraded-Network Tail Tolerance in Data-Parallel Frameworks. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[244] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[245] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[246] Toby J. Teorey and Tad B. Pinkerton. A Comparative Analysis of Disk Scheduling Policies. *Communications of the ACM (CACM)*, 15(3), 1972.

[247] Utah Emulab Testbed. RAID controller timeout problems on boss node, on 6/21/2013. http://www.emulab.net/news.php3, 2013.

[248] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[249] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[250] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2012.

[251] Elizabeth Varki, Arif Merchant, Jianzhang Xu, and Xiaozhou Qiu. Issues and Challenges in the Performance Analysis of Real Disk Arrays. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(6), 2004.

[252] Kristian Vatto. Samsung Acknowledges the SSD 840 EVO Read Performance Bug - Fix Is on the Way. http://www.anandtech.com/show/8550/samsung-acknowledges-the-ssd-840-evo-read-performance-bug-fix-is-on-the-way, 2014.

[253] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The Power of Choice in Data-Aware Cluster Scheduling. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[254] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[255] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[256] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. DLAU: A Scalable Deep Learning Accelerator Unit on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(3):513–517, 2017.

[257] Mengzhi Wang, Kinman Au, Anastassia Ailamaki, Anthony Brockwell, Christos Faloutsos, and Gregory R. Ganger. Storage Device Performance Prediction with CART Models. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2004.

[258] Olivia Weng and Andrew A. Chien. Evaluating Achievable Latency and Cost: SSD Latency Predictors. In *2nd Workshop on Accelerated Machine Learning (AccML)*, 2020.

[259] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1995.

[260] Guanying Wu and Xubin He. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[261] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and Inference with Integers in Deep Neural Networks. In *6th International Conference on Learning Representations (ICLR)*, 2018.

[262] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. Overcome the GC-induced Performance Variability in SSD-based RAIDs with Request Redirection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 38(5), May 2019.

[263] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and Bo Mao. GC-aware Request Steering with Improved Performance and Reliability for SSD-based RAIDs. In *Proceedings of the 32th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[264] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[265] Huaxia Xia and Andrew A. Chien. RobuSTore: Robust Performance for Distributed Storage Systems. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC)*, 2007.

[266] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Andy Rudoff, and Steven Swanson. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[267] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[268] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[269] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. Wrangler: Predictable and Faster Jobs using Fewer Resources. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[270] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[271] Haojin Yang, Martin Fritzsche, Christian Bartz, and Christoph Meinel. BMXNet: An Open-Source Binary Neural Network Implementation Based on MXNet. In *Proceedings of the 2017 ACM on Multimedia Conference*, 2017.

[272] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level I/O scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[273] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[274] Li Yin, Sandeep Uttamchandani, and Randy Katz. An Empirical Exploration of Black-Box Performance Models for Storage Systems. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2006.

[275] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, Sooyong Kang, Jongmoo Choi, Sungroh Yoon, and Jaehyuk Cha. VSSIM: Virtual machine based SSD simulator. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2013.

[276] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. In *Proceedings of the 2005 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2005.

[277] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[278] Hong Zhang, Kai Chen, Wei Bai, Dongsu Han, Chen Tian, Hao Wang, Haibing Guan, and Ming Zhang. Guaranteeing Deadlines for Inter-Datacenter Transfers. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[279] Jianquan Zhang, Dan Feng, Jianlin Gao, Wei Tong, Jingning Liu, Yu Hua, Yang Gao, Caihua Fang, Wen Xia, Feiling Fu, and Yaqing Li. Application-Aware and Software-Defined SSD Scheme for Tencent Large-Scale Storage System. In *Proceedings of 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[280] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[281] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, vrigo Gokhale, and John Wilkes. CPI2: CPU performance isolation for shared compute clusters. In *Proceedings of the 2013 EuroSys Conference (EuroSys)*, 2013.

[282] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[283] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS)*, 2014.

[284] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[285] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.