THE UNIVERSITY OF CHICAGO

EVOLVING STORAGE STACK FOR PREDICTABILITY AND EFFICIENCY

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

HUAICHENG LI

CHICAGO, ILLINOIS
AUGUST 2020

Dedicated to my parents and significant other.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

It is with tremendous help and support from many people that this Ph.D. is made possible and unique. I am very grateful to these individuals and would like to thank them in this special section.

First and foremost, I am deeply indebted to my advisor, Haryadi Gunawi, for his tireless mentoring, guidance, and assistance during my Ph.D. study. Haryadi was not my assigned advisor when I first came to University of Chicago. But my passion in Operating/Storage Systems research urged me to talk to him soon after I arrived in Chicago. Haryadi saw my passion and kindly took me into his group. Until today, I still feel lucky our first conversation worked out and it earned me the opportunity for me to pursue my research dreams in Systems!

Haryadi provided me all the necessary help to grow into a successful researcher. He always puts his students in the first place. Thus, I have never found it difficult to reach him, either through Skype/Email or office visits. Besides our weekly meetings, my sporadic visits to his office significantly boosted my research progress during the first two years in timely clearing potential miscommunications and getting quick feedback on intermediate results. Haryadi knows us very well and he customizes the advising styles to best suit each student. I benefited a lot from this style in reflecting and overcoming my shortcomings as well as building my advantage. Given my interest in developing a research career in academia, Haryadi gave me first-hand experience of the professional lifestyle and helped me develop my own style in effectively advising students and broadening my network by attending conferences, reviewing papers, and mentoring students. As for research, Haryadi always holds high standards and pushes me to think one level deeper. From him, I learned to make short-term "visible" progress by thinking ahead to divide the tasks into small pieces. I believe this will become a fundamental rule I will stick to for my future research. Haryadi is a role model to me with many good qualities: He is diligent, and I run into him many times in the hallway during weekends at both Ryerson and JCL; He is organized, and my Ph.D. life becomes much easier thanks to many automation scripts he shared; He is caring, and I'm grateful to his help in paper revising, internship recommendations, and allowing me the freedom to pursue

what interests me most.

I would like to thank Andrew Chien and Anirudh Badam for helping serve on my committee and their feedback on my dissertation makes it much stronger. Andrew has been a long-time collaborator and I always learn a lot from his wisdom through conversations and lectures. Anirudh is a mentor as well as a friend to me. I was lucky enough to be his intern in 2018 at Microsoft Research and he greatly influenced my research mindset. Moreover, I'm also grateful for their support during my job search.

Additionally, many thanks to the UCARE members (alphabetical order) for the friendship and for making the UCARE group a warm hub for mutual encouragement and fun: Cesar Stuardo, Daniar Kurniawan, Huan Ke, Jeffery Lukman, Michael Hao Tong, Mingzhe Hao, Meng Wang, Riza Suminto, Shiqin Yan, Tanakorn Leesatapornwongsa.

I'm fortunate to have mentored several outstanding undergraduate and master students: Martin L. Putra, Fadhil I. Kurnia, Ronald Shi, Yesa Rahmad, etc.. Thank you for your contributions to the research projects!

The life at University of Chicago wouldn't be so joyful without many friends. Thank you, Dixin Tang, Guangpu Li, Shu Wang, Yi Ding, Yun Li, Yuxi Chen, Zechao Shang and Zhixuan Zhou for the many dinners/conversations we shared and the fun we had together.

I also want to thank the CERES research center, the CS department, and our awesome staff. The CERES summits have been an excellent opportunity for me to socialize with others and learn about many interesting ongoing works. I feel lucky to grow together with the department in the last five years. And the tech and admin staff are very nice. I want to thank them for their services: Bob Bartlett, Colin Hudler, Donna Brooms, Margaret P. Jaffey, Nita K. Yack, Phil Kauffman, and Tom Dobes. In particular, I frequently bug Bob, Tom, and Phil for hardware purchases, fixing networking issues, and setting up special hardware, and I'm really grateful that they kindly help guide me through these.

There are many other collaborators, mentors, and friends I enjoyed working with and I would like to express my gratitude to. I have been very fortunate to be a 3× research intern at Microsoft Research throughout my Ph.D., working with many smart people: Dan R. Ports, Daniel S. Berger, Irene Zhang, Jaeyoung Do, Ricardo Bianchini, and Stanko Novakovic. I enjoyed many insightful discussions with them. I also spent a few months at NetApp ATG working with Xing Lin, Fan Ni, and Art Harkin. Many thanks to them for kindly hosting me during an irregular period of time. Matias Bjørling and Javier Gonzalez helped me a lot when I started working on OpenChannel-SSDs, and I appreciate that!

Lastly, I want to express my gratitude to my family for their unconditional love. My parents work very hard to support the entire family and provide me the best they can. Their perseverance and diligence always motivate me to aim higher and better and never stop moving forward. My significant other, Jingyu Zhang, and I have been together for 8 years and I couldn't thank her more for her company and the happy time we spent together along the way. She lightened the spark in me to pursue a Ph.D. and encouraged me to come to the U.S. for graduate school even though we would be separated by the Pacific Ocean. I dedicate this dissertation to her and my parents.

Abstract

With the exponential growth of data that are expected to reach 175 zettabytes by 2025, cloud storage is increasingly becoming the central hub for data management and processing. Among the many benefits cloud platforms promise, predictable performance and cost-efficiency are two fundamental factors driving the success of modern cloud storage. However, under rapid changes in modern cloud storage infrastructure in terms of both software and hardware, new challenges emerge for achieving predictable performance with efficiency.

In more detail, modern data intensive applications and the new wave of computing paradigms (*e.g.*, data analytics, ML, serverless) drive the storage stack to undergo a radical shift towards more feature-rich software designs on top of increasingly heterogeneous architectures. As a result, today's cloud storage stack is extremely heavy-weight and complex, burning 10-20% of data center CPU cycles and introducing severe performance non-determinism (*i.e.*, long tail latencies). Unfortunately, the deployment of new acceleration hardware (*e.g.*, NVMe SSDs and IO co-processors) only partially addresses the problem. Due to the intrinsic complexities and idiosyncrasies in hardware (*e.g.*, NAND Flash management) and lack of system-level support, it remains a challenge to design performant and cost-efficient cloud storage systems. In particular, achieving sub-millisecond level latency predictability in a cost-efficient manner is the new battlefield.

Rooted in deep understanding and analysis of existing software/hardware stack, *this dissertation focuses on building new abstractions, interfaces, and end-to-end storage systems to achieve predictable performance and cost-efficiency using a software/hardware co-design approach.* By revisiting the challenges across different layers in a holistic manner, the co-design approach opens up simple yet powerful system-level policy designs to opportunistically exploit hardware idiosyncrasies and heterogeneity. The systems we build can effectively decrease latency spikes by up to orders of magnitude and increase cost savings by 20×.

To address the challenge of predictable performance in modern Flash storage systems, we present TEAFA, a tail-evading flash array design delivering deterministic performance. TEAFA uniquely combines a simple yet powerful host-SSD interface, time window mechanism, and data redundancy to proactively and deterministically reconstruct late requests, with only minor changes to the host software and device firmware. The evaluation results across 9 data center storage traces and several real storage workloads (*e.g.*, FileBench, YCSB/RocksDB) show that TEAFA improves the baseline by orders of magnitude and is only 1.1× to 2.1× slower than an ideal case where there are no background operations induced tail latencies. When compared to other state-of-the-art works (*e.g.*, Proactive approach, Preemptive GC, P/E Suspension, Flash-on-Rails, and Harmonia) focusing on improving IO performance, TEAFA is more deterministic and effective in cutting tail latencies while being less intrusive and easy to deploy. Along with TEAFA, we also introduce OS-level and device-level approaches, MITTOS and TTFLASH, respectively, to eliminate tail latencies across the entire storage stack.

Although TEAFA can effectively improves tail latencies, a significant portion of CPU cycles is needed to fulfill the reconstruction computations. Worse, at a large scale, the "storage tax" that cloud providers have to pay takes up to 10-20% of datacenter CPU cycles. Thus, it's challenging to achieve cost/resource-efficiency in modern cloud storage stack designs. One opportunity is to utilize modern IO accelerators for cost-efficient storage offloading. Yet, the complex cloud storage stack is not completely offload-ready to today's IO accelerators. To tackle the cost-efficiency challenge, we present LeapIO, a next-generation cloud storage stack that leverages ARM-based co-processors to offload complex storage services. LeapIO addresses many deployment challenges, such as hardware fungibility, software portability, virtualizability, composability, and efficiency. It employs a set of OS/software techniques and new hardware properties to provide a uniform address space across the x86 and ARM cores to minimize data copies and directly expose virtual NVMe storage to unmodified guest VMs. At the core, LeapIO runtime enables agile storage service development at the user-space. 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 the peak bandwidth

of 0.65 million IOPS on a datacenter SSD). Our current SoC prototype also delivers acceptable performance, 5% further reduction on the server side (and up to 30% on the client) but with more than 20$\times$ cost savings. Overall, LeapIO helps cloud providers cut the storage tax and improve utilization without sacrificing performance.

Finally, we discuss the importance of scalable and extensible research platforms for fostering future full-stack software/hardware storage research. Existing software platforms (*e.g.*, SSD/SoC simulators or emulators) are limited by the types of research they support, outdated, and not scalable. Hardware platforms suffer from wear-out issues and are difficult to use. Thus, it's not an excellent choice for new idea exploration in the early phase neither. We argue that it is a critical time for the storage research community to have a new software-based full-system SSD emulator. To this end, we build FEMU, a software (QEMU-based) NVMe flash emulator. FEMU is cheap (open-sourced), relatively accurate (0.5-38% variance as a drop-in replacement of OpenChannel SSD), scalable (can support 32 parallel channels/chips), and extensible (support internal-only and split-level SSD research). FEMU has been used by researchers from tens of institutions and in classes, demonstrating the urgent need for such a research platform and its success.

CHAPTER 1

INTRODUCTION

In this data era, data is being generated everywhere and we face a more urgent need than ever before to store and process data fast and efficiently. With the exponential growth of data that is expected to reach 175 zettabytes by 2025, cloud storage is increasingly becoming the central hub for data management and processing. Thus, storage research is of crucial importance to the continuity of the data-driven development of the modern world.

Among the many benefits cloud platforms promise, predictable performance and cost-efficiency are two fundamental factors driving the success of modern cloud storage. Specifically, users demand fast access to the data in an economical way. To that end, cloud providers keep improving the storage stack to match the ever-increasing data needs with new techniques. However, under rapid changes in modern cloud storage infrastructure in terms of both software and hardware, new challenges emerge for achieving predictable performance with efficiency.

In more detail, modern data intensive applications and the new wave of computing paradigms (*e.g.*, data analytics, ML, serverless) drive the storage stack to undergo a radical shift towards more feature-rich software designs on top of increasingly heterogeneous architectures. As a result, today's cloud storage stack is extremely heavy-weight and complex, burning 10-20% of data center CPU cycles and introducing severe performance non-determinism (*i.e.*, long tail latencies). Unfortunately, the deployment of new acceleration hardware (*e.g.*, NVMe SSDs and IO co-processors) only partially addresses the problem. Due to the intrinsic complexities and idiosyncrasies in hardware (*e.g.*, NAND Flash management) and lack of system-level support, it remains a challenge to design performant and cost-efficient cloud storage systems. In particular, achieving sub-millisecond level latency predictability in a cost-efficient manner is the new battlefield.

1.1 Thesis Statement

In this dissertation, we seek to answer the question: *How should the storage stack evolve to satisfy the increasing needs for low and predictable latencies and cost-efficiency in the context of heavy/unpredictable software stack and heterogeneous architectures?* In particular, we make the following thesis statement:

The performance predictability and CPU efficiency challenges associated with modern storage stack can be effectively tackled by cross-layer co-designs across applications/OS and software/hardware boundaries to exploit data redundancy for low and consistent latencies as well as to seamlessly utilize low-cost co-processors for agile storage service offloading, both with small efforts for easy deployment.

To support this statement, in the first part of the dissertation, we explore various novel techniques and designs to achieve predictable latencies across different layers of the storage stack (Chapter 3). In the second part, we think further to design the next-generation offloaded storage stack to target both performance and cost-efficiency (Chapter 4). In the third part, we revisit the lack of proper research platform support for conducting modern storage research (like the ones discussed in the aforementioned parts) and develop a new storage research platform to fill the gap (Chapter 5). In summary, this dissertation will cover the following systems we built:

• **MITTOS:** An OS that is transparent, exposing its resource busyness to enable applications to achieve millisecond tail tolerance.

• **TEAFA:** A flash array delivering predictable performance, which uniquely combines a simple yet powerful host-SSD interface to proactively and deterministically reconstruct late requests.

• **TTFLASH:** A "tiny-tail" flash drive (SSD) that eliminates GC-induced tail latencies by circumventing GC-blocked IOs with several novel strategies.

• **FEMU:** A software (QEMU-based) flash emulator for fostering future full-stack software/

hardware SSD research, with accuracy, extensibility and scalability.

• **LeapIO:** A new cloud storage stack that leverages ARM-based co-processors to offload complex storage services while satisfying many deployment requirements.

For the rest of the chapter, we will briefly introduce the problems of performance predictablity (§1.2) and cost-efficiency (§1.3) in modern storage stack, categorize state-of-the-art approaches and give a high-level overview of our solutions. In §1.4, we summarize the contributions of this dissertation. In §1.5, we introduce the outline for the rest of the dissertation.

1.2 Performance Predictability

Low and stable latency is a critical key to the success of many services, but variable load and resource sharing common in cloud environments induce resource contention that in turn produces "the tail latency problem." With flash storage becoming the mainstream destination for storage users, the SSD consumer market continues to grow at a significant rate [17], SSD-backed cloud-VM instances are becoming the norm [14, 15]. From the users' side, they demand fast and stable latencies [143, 160]. However, SSDs do not always deliver the performance that users expect [36]. Some even suggest that flash storage "may not save the world" (due to the tail latency problem) [28]. Some recent works dissect why it is hard to meet Service Level Agreement (SLA) with SSDs [199] and reveal high performance variability in 7 million hours of SSDs deployments [165]. Built on top of flash/SSD, All Flash/SSD Array (AFA) is a popular solution for high-end storage servers [67, 73, 80, 138]. The increasing deployment of real-time analytics and machine learning applications further fuels the growth of AFA market to a predicted $18 billion market by 2023 [7, 11]. Similarly, large scale flash storage common in large datacenter/cloud storage providers [6, 55, 257] must address users' craving for low and predictable latencies [118, 141].

In this context, tail latency is very important. Many recent SSD products are released and evaluated not just on the average speed but the percentile latencies as well [37, 77, 180]. These all paint the reality that customers would like SSD products with deterministically stable and low latency. Unfortunately, it's hard to have SSDs deliver stable latencies. The non-deterministic performance comes from the fact that SSDs must perform many internal (background) management activities such as the garbage collection (GC) process, wear leveling, and internal buffer flush [34, 149, 331]. Even a single background write/erase operation will cause the user IOs to be queued (delayed) behind for tens of milliseconds or worse [205, 232, 309]. At the core 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 IOs. Due to an ongoing GC, read latency variance can increase by 100$\times$ [28, 141]. In the last decade, there is a large body of work that *reduces* the number of GC operations with a variety of novel techniques. 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.

1.2.1 Existing Approaches

Due to this problem, many works have been proposed in the last decade to achieve stable latencies, to reduce latency tail. One view of the design space is depicted in Figure 1.1, which we discuss at a high level below.

• **White-box (*i.e.*, Use full-knowledge of the hardware for improved performance):** This category arguably has the highest number of publications. The lower left of the figure represents pure white-box approaches that assume the SSD firmware can be entirely modified [137, 149, 185, 186, 190, 196, 220, 233, 249, 250, 296, 310, 311, 318, 328]. Many of the techniques here only delay/optimize the GC process but not evade the GC process, thus still far from reducing the tail significantly. Rearchitecting the firmware is also not a desirable option for many commodity SSDs. The upper side of the left region is an entirely host-managed/software-defined flash [56, 126, 164, 170, 222, 257, 279]. Theoretically here the host can be specifically tuned such that the SSD management operations do not interfere with user operations, but host-managed

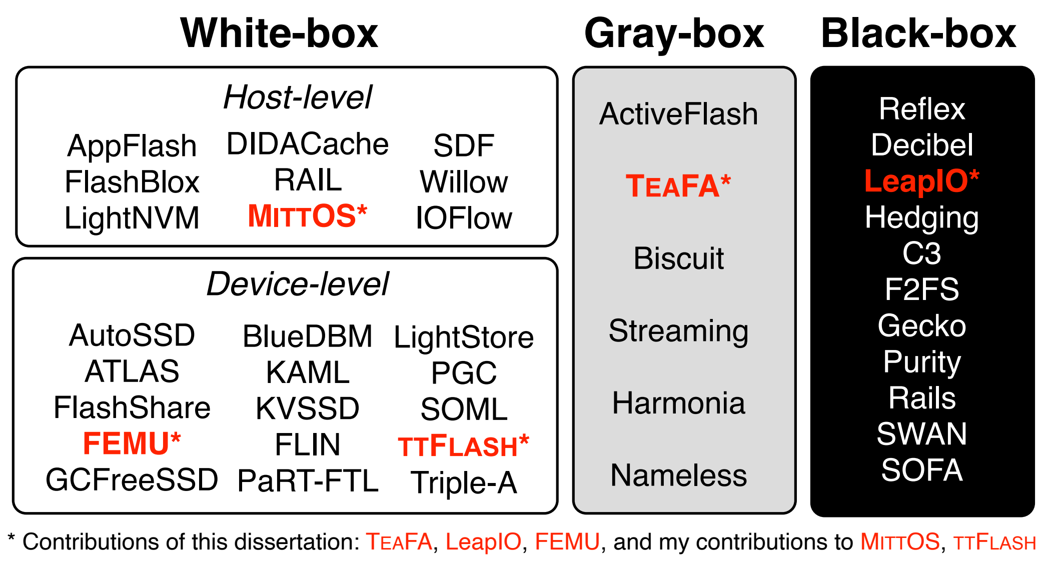


Figure 1.1: **Design space for achieving stable performance on top of modern Flash storage stack.** *The figure depicts the design space of research in reducing tail latencies. The x-axis represents the three spectrums, white-, gray-, and black-box approaches.* **Top left:** *AppFlash [222], DIDACache [283], SDF [257], FlashBlox [170], RAIL [56], Willow [279], LightNVM [126], MittOS [164], IOFlow [297];* **Bottom left:** *AutoSSD [196], BlueDBM [183], LightStore [137], ATLAS [149], KAML [178], PGC [220], FlashShare [328], KVSSD [190], SOML [233], FEMU [228], FLIN [296],* TTFLASH *[318], GCFreeSSD [185], PaRT-FTL [249], Triple-A [186];* **Center:** *Nameless [330], Harmonia [209], Stream [207], Biscuit [155],* TEAFA *[229], ActiveFlash [298];* **Right:** *Reflex [211], Decibel [251], LeapIO [227], Hedging [105, 141], C3 [295], F2FS [218], Gecko [286], Purity [138], Rails [292], SWAN [201], SOFA [134].*

strategies require "open" devices (*e.g.*, OpenChannel SSDs) that are not as pervasive as commodity SSDs in deployment.

• **Black-box (*i.e.*, Application-level only strategies to optimize performance):** On the other extreme, the right side of the figure are black-box approaches that attempt to circumvent tail latencies at the application/OS layer without modifying the firmware or host-SSD interface [134, 138, 201, 218, 251, 286, 292, 312]. The advantage is that they can directly work on off-the-shelf SSDs, but the downside is the difficulty to guarantee performance without full control of the SSD. For example, early efforts to cut latency tails focused on coarse-grained jobs (tens to hundreds of seconds) [142], 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 [105, 106, 326]. More challenging are applications that generate large numbers of small requests, each expected to finish in milliseconds. For these, techniques that "wait-then-speculate" are 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 destinations (replicas) and the first to respond is used [105, 304, 313]. This is proactive speculation, but *doubles* the workload intensity. To reduce extra load, applications can delay the duplicate request slightly and cancel the clone when a response is received (a *"tied requests"*) [141]; to achieve this, however, IO queue management and revocation capability must be *built* in the application layer [127]. A more conservative alternative is *"hedged requests"* [141], 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, if extra requests are not desirable, *"snitching"* [8, 295] – the application 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/Hypervisor 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 or device for some time. Currently, there is no way for the OS or SSD 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/SSD is busy."

• **Gray-box (*i.e.*, Cross-layer collaboration for performance):** After more than a decade of research, the storage community decides that perhaps a better way is a middle (gray-box) approach where the host (or applications) is allowed to control SSD internal operations but only in a limited way [27, 207, 208, 209, 330]. The best recent example is the birth of the *IO Determinism* (IOD) interface, which has been accepted to the NVMe 1.4 specification in mid-2019. One feature of

IOD is to allow the host to submit a time window to the SSD, telling the device to try its best not to perform any management activities within the time window, ultimately to guarantee a "deterministic" performance [83]. But challenges remain. IOD is still a best-effort interface that does not 100% guarantee free disturbance within the window. How to program the window value and integrate IOD to the host stack are also open problems.

1.2.2 Overview of Our Solutions

We tackle the problem at different layers of the storage stack to achieve best performance. Below we introduce our approaches in a top-down manner (MITTOS at the Application/OS level, TEAFA at the Host/SSD boundary, and TTFLASH at the hardware level).

• **OS Support for Predictable Performance:** First, to solve the problem at the application and the OS level, we advocate a new philosophy: *OS calls should transparently expose OS busyness to applications.* 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 manage tail impacts. If advantageous, they can choose *not* to wait, for example performing an *instant* failover to another replica or taking other corrective actions. This approach pushes prediction mechanisms into the OS layer which has more information, henceforth the tail-tolerance logic in applications can be simplified. To this end, we introduce MITTOS, an OS that is transparent, exposing its resource busyness to enable applications to achieve millisecond tail tolerance. We materialize busyness transparency within the storage software stack, primarily because storage devices are a major resource of contention [165, 199, 289]. In a nutshell, with MITTOS, applications attach a deadline to IO operations (*e.g.*, "`read()` should not take more than 100ms"). And, if the deadline cannot be satisfied (*e.g.*, long disk queue), MITTOS immediately returns `EBUSY`, freeing the application to quickly retry to another node *without* a long wait (perhaps tens to hundreds of milliseconds). To examine how busyness transparency in 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, and thus can benefit from MITTOS support. Compared to hedged requests (the most effective black-box approach), MITTOS reduces the completion time of individual IO requests by 23-26% at 95th percentile, and 6-10% on average. Better, as tail latencies can be amplified by scale (*i.e.*, a user request can be composed of 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 95th percentile and 16-23% on average. The higher the scale factor, the more reduction MITTOS delivers.

• **Host/SSD Co-design for Tail Tolerance:** In the spirit of the gray-box movement, we introduce TEAFA, a tail-reducing flash array. TEAFA leverages the existing IOD interface, uniquely programs the window time in the context of flash array, extends slightly the NVMe read submission/completion commands, modifies the SSD firmware level only minimally, and implements most of the logic in the OS/host layer. All of these are done without adding a new interface nor modifying the heavy SSD internal management (*e.g.*, modify the GC/flush algorithm), but at the same time being able to provide stable and deterministic latency (*e.g.*, up to the 99.99th percentile). Our contributions specifically lie in the following techniques. **(a)** *Fast-fail and busy bits:* In TEAFA, we extend the NVMe IO submission command with a "fast-fail" bit and the completion command with a "busy" bit. With this simple extension, latency-sensitive read IOs can be tagged with the fast-fail bit enabled, telling the SSD firmware to quickly return the read to the host with the busy bit set if the read must wait behind some background operations. In the context of flash array with redundancy (*e.g.*, RAID-5), this busy notification allows the host to reconstruct the content of the busy read by performing extra reads from other drives and reconstruct the late data with parity computation. **(b)** *Shortest (background) remaining time:* In the case of a large IO that reads from multiple SSDs, multiple of the sub-IOs may receive busy errors from multiple SSDs and the host cannot reconstruct all the busy reads (*e.g.*, RAID-5 host can only reconstruct one busy read). For this, we extend the completion command to also include "background remaining time" to be piggybacked with the busy signal. For example, in RAID-5, when two sub-IOs are delayed by two SSDs, the host can resubmit one of them to the SSD with the shortest background remaining time (SBRT policy). This is useful as background remaining time can range up to tens of milliseconds. **(c)** *Static/dynamic IOD time window:* SBRT policy only reduces the delay, but ideally we should mitigate multiple busy reads completely. For this, TEAFA leverages the IOD time-window interface to inform the SSD when not to perform any background operation. Essentially, TEAFA enforces the SSDs in an array to perform background operations in non-overlapping windows – if one SSD is performing background operations in one time window, others in the array are not allowed. As mentioned above, IOD time window is a new concept; to the best of our knowledge, no publication shows how to configure the window value. We contribute by introducing a static and a dynamic algorithm to set up the window value in TEAFA. The static algorithm is based on the SSD internal parameters and the dynamic one is about the host/OS modifying the value on the run. We comprehensively evaluate all the TEAFA techniques above using a variety of micro- and macro-benchmarks and compare it with other works such as Flash on Rails [292], Harmonia [209], a naive black-box method (proactively issuing multiple requests), and preemptive GCs [220]. Compared to pure no-background ("NoBG") scenario, TEAFA only produces 1.1× to 2.1× slowdowns between p99 and p99.99, improving the baseline by *orders* of magnitude. Compared to other works, TEAFA is more deterministic in trimming the tail latencies and easy to deploy with minimal changes.

• **High Performance SSD Architecture Design:** We address this urgent issue with "tiny-tail" flash drive (TTFLASH), a GC-tolerant SSD that can deliver and guarantee stable performance. The goal of TTFLASH is to eliminate GC-induced tail latencies by *circumventing GC-blocked IOs*. That is, ideally there should be *no* IO 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 TTFLASH. First, we leverage the increasing power and speed of today's flash controllers that *enable more complex* *logic* (*e.g.*, multi-threading, IO concurrency, fine-grained IO 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 [171, 200, 277]. Due to this increasing failure, modern commercial SSDs employ parity-based redundancies (RAIN) as a standard data protection mechanism [13, 21]. By using RAIN, we can *circumvent GC-blocked read IOs with parity regeneration*. Finally, modern SSDs come with a large RAM buffer (hundreds of MBs) backed by "super capacitors" [20, 24], which we leverage to *mask write tail latencies* from GC operations. The timely combination of the technology practices above enables four new strategies in TTFLASH: **(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 IOs 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. With a thorough evaluation, we show that TTFLASH successfully eliminates GC blocking for a significant number of IOs, reducing GC-blocked IOs from 2–7% (base case) to only 0.003–0.7%. As a result, TTFLASH reduces tail latencies dramatically. Specifically, between the 99–99.99th percentiles, compared to the perfect no-GC scenario, a base approach suffers from 5.6–138.2$\times$ GC-induced slowdowns. TTFLASH on the other hand is only 1.0 to 2.6$\times$ slower than the no-GC case, which confirms our near-complete elimination of GC blocking and the resulting tail latencies. 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.

1.3 “Storage Tax” — Efficiency Challenge

Cloud storage has improved drastically in size and speed in the last decade, with a market size expected to grow to $88 billion by 2022 [66]. With this growth, making cloud storage efficient is paramount. On the technical side, cloud storage is facing two trends, the growing complexity of cloud drives and the rise of IO accelerators, both unfortunately have not blended to the fullest extent.

First, to satisfy customer needs, today’s cloud providers must implement a wide variety of storage (drive-level) functions as listed in Table 1.1. 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. Last but not least, future cloud drives must support fancier interfaces [89, 97, 112, 257, 279].

As a result of these requirements, the cloud storage stack is extremely resource hungry. Our experiments suggest that the cloud provider may pay a *heavy tax* for storage: 10–20% of x86 cores may have to be reserved for running storage functions. Ideally, host CPU cores are better spent for providing more compute power to customer VMs.

The second trend is the increasing prevalence of IO acceleration technologies such as SmartSSD [57, 82], SmartNIC [49, 58] 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 1.1 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.

Table 1.1 summarizes why the functionalities are not fully offload ready. We use one simple case as an example. For "local virtual SSDs," a cloud storage provider can employ Single Root IO Virtualization (SR-IOV) SSDs [90] where IOPS/bandwidth virtualization management is offloaded to the SSD hardware, freeing the host from such a burden. However, the cloud provider might want to combine virtualization with aggressive caching in spare host DRAM, but in-SSD accelerators cannot leverage the large host DRAM (*i.e.*, not composable with other host resources) and do not provide the same flexibility as software.

Custom IO accelerators have another downside. As acceleration hardware evolves, the entire fleet of servers may never be uniform; each generation of servers will have better, but slightly different hardware from previous ones. Without a *unifying software platform*, we run the risk of fragmenting the fleet into silos defined by their hardware capabilities and specific software optimizations.

We observe another trend in cloud platforms: ARM co-processors are being deployed for server workloads. This is a more suitable alternative compared to custom accelerators; ARM cores are more general (retain x86 generality) and powerful enough to run complex storage functions without major performance loss.

Offloading the storage stack to ARM co-processors can bring substantial cost savings. The bill-of-material cost of an ARM System-on-Chip (SoC) is low and the power consumed is 10W, making an annual total Cost of Ownership (TCO) of less than ~\$100 (<~3% of a typical server's annual TCO). In turn, this SoC frees up several x86 cores thereby directly increasing the revenue from the services running on the server proportional to the additional cores – annually ~\$2,000 or more (20$\times$) when the cores are used for running customer VMs and significantly higher for more lucrative services. Even when accounting for typical replacement rates, ARM SoC TCO would still be less than one year rent of the smallest recommended VM in the cloud.

But there is a major challenge, just dropping an ARM SoC on a PCIe slot would not be enough. We had to rethink the entire storage stack design to meet real deployment challenges: hardware

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 [75], NVMeoF [38]) cannot be offloaded easily from x86.
Atomic write drive. Smart transactional storage devices [246, 267] 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 1.1: **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.*

fungibility, portability, virtualizability, composability, efficiency, and extensibility, which led us to designing LeapIO.

1.3.1 LeapIO Introduction

We present LeapIO, our next-generation cloud storage stack that leverages ARM SoC as co-processors. To address deployment goals (§2.6) 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" [52]. 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.

We build LeapIO in 14,388 LOC across the runtime, host OS/hypervisor and QEMU changes, and design the SoC using Broadcom StingRay V1 SoC.

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$\times$ cost savings.

Finally, we implement and compose different storage functions such as a simple RAID-like aggregation and replication of local/remote virtual drives via NVMe-over-RDMA/TCP/REST, an

IO priority mechanism, a multi-block atomic-write drive, a snapshot-consistent readable drive, the first virtualized OpenChannel SSDs exposed to guest VMs, block cache, and many more, all written in 70 to 4400 LOC in user space, demonstrating the ease of composability and extensibility that LeapIO delivers.

1.4 Contributions

Overall, this dissertation makes the following contributions:

- We introduce a holistic predictable storage stack design spanning different layers of the software/hardware hierarchy, ranging from novel upper-level application/OS abstractions to low-level OS/SSD interface and SSD controller architecture designs. And we demonstrate their superior performance (*i.e.*, tail latencies) through extensive evaluations over a wide range of storage/data workloads and comparisons with many state-of-the-art works.
- We advocate a new principle that the OS or the hardware (*e.g.*, SSD controllers) should quickly reject IOs ("fast fail") that cannot be promptly served to enable flexible systems-level policy designs for consistent IO performance. We materialize this principle in two scenarios by exploiting data redundancy in modern storage systems: First, we introduce a fast rejecting SLO-aware OS interface to add OS support for millisecond level tail tolerance by quickly failing-over to less busy replicas; Second, we design a tail-evading all flash array on top of a slightly modified NVMe interface by proactively reconstructing late IO requests through RAID-level parity.
- We introduce a new SSD architecture that achieves guaranteed performance close to an ideal scenario with a fine-grained plane-blocking Garbage Collection (GC) architecture to reduce GC interferences to the foreground IOs, and exploiting Redundant Array of Independent NAND (RAIN) and rotating GC for guaranteed low-latency IO reconstruction.
- We define and design a complete set of new hardware properties to make ARMSoC-to-

peripheral communications as efficient as x86-to-peripherals. Furthermore, by taking advantage of these new hardware properties, we introduce a uniform address space across x86, ARM SoC, and other PCIe devices (SSDs, NICs) to enable line-rate address translations and data movement.

- We develop a portable storage offloading framework/runtime which abstracts away hardware capabilities and exploits the uniform address space to make offloading seamless and flexible. This enables an "offload-ready" design which allows storage services to portably run on x86 or ARM whenever they are available. In essence, we treat hardware acceleration as an opportunity rather than a necessity. This helps avoid fragmenting server fleets in data centers into silos defined by their hardware capabilities and software optimizations. On top of the runtime, we build several novel services composed of local/remote SSDs/services and perform detailed performance benchmarks as well as analysis.
- We develop FEMU, a software-based NVMe SSD emulator to meet the increasing demands for a cheap, accurate, scalable, and extensible storage platform that supports the rising software-defined, split-level, and full-stack research.
- We contribute open-source[1] implementations of MITTOS [1], TTFLASH [2], LeapIO [3], TEAFA [4], and FEMU [5], demonstrating the efficacy of our principle, system and policy designs. Additionally, these systems are seeing impacts: LeapIO is being deployed in Microsoft data centers to serve production workloads, MITTOS findings have been partially merged to official Linux kernel and FEMU has been used by tens of institutions and classes.

1.5 Thesis Organization

The rest of the dissertation is organized as follows: In Chapter 2, we go through some background knowledge to understand the state of the existing storage stack, quantify the unpredictability, and

1. TEAFA and LeapIO are to be released as of the writing.

motivate the offloading design for efficiency. In Chapter 3, we dive deep into the designs and evaluation of our solutions on achieving predictable performance across the entire storage stack: new abstractions at application/OS level (MITTOS), host/SSD interface for deterministic communication (TEAFA), and new architectures of SSD controller designs to eliminate tail latencies (TTFLASH). In Chapter 4, we present our principled design of offload-ready cloud storage stack design to satisfy data center deployment requirements (LeapIO). In Chapter 5, we discuss the urgent need for a software-based SSD research platform to foster future full-stack research, and talk about our solution (FEMU). In Chapter 6, we discuss the limitations of our solutions and the broad design space/concerns regarding tail-tolerance and efficient offloading. In Chapter 7, we review an extensive list of related works and distinguish our solutions from them. Lastly, we talk about future work in Chapter 8 and conclude in Chapter 9.

Chapter 2

Background and Motivation

This chapter provides the necessary background and motivation for various aspects of this dissertation. We start with a high-level overview of modern storage stack (§2.1), introducing the key components this dissertation revisits. Then, we talk about the NVMe interface (§2.2) and SSD controller architectures (§2.3) to conceptually understand the state of modern flash storage devices and reason about the potential "sources" of unpredictable IO latencies. Next, we analyze and quantify the effects of GC-induced performance unpredictability at both single SSD device (§2.4) and flash array scale (§2.5) to motivate our holistic tail-tolerant designs. Lastly, we discuss the motivation for storage offloading to achieve cost-efficiency (§2.6).

2.1 Modern Storage Stack Overview

As shown in Figure 2.1, modern storage stack consists of 5 main pieces, which we elaborate below.

❶ **User Storage Applications:** Traditional storage applications such as file server, object store, KV store, and databases run in the user space and utilize the high-level APIs to access storage services. As modern storage workloads become much more data-intensive and performance-hungry, it is extremely important to deliver fast and efficient access to data with minimal application changes. To this end, we revisit the software and hardware trends to redesign the entire storage stack while only requiring very small application-side modifications (§3.2, §3.1).

❷ **Application/OS Interface:** Applications can simply call `read()` or `write()` to read/write data. These block/file-based (POSIX) APIs have been in use for decades and are the standard storage interface between applications and the low-level storage stack. While being easy to use, the interface is passive, not allowing applications enough flexibility to achieve performance predictability. For example, lower-level latency hiccups caused by resource contentions will always propagate back to applications and there is currently no way for applications to request deterministic latency.

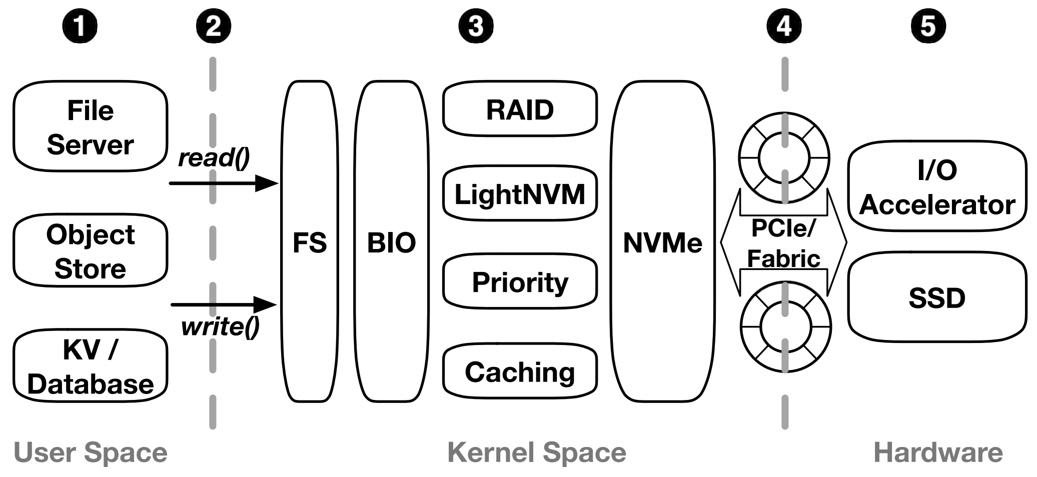


Figure 2.1: **Modern Storage Stack.** *The figure shows the simplified storage stack across storage applications running at the user space, IO management functionalities (i.e., File/Block IO stack) at the kernel space, storage/acceleration devices (e.g., SSDs and IO Accelerators) at the hardware level, as well as the interfaces connecting them (i.e., POSIX sitting between user applications and the OS, and NVMe in between the software and hardware).*

What new interfaces are needed to overcome the aforementioned drawbacks? We answer this later with MittOS (§3.1).

❸ **Kernel/OS level Block IO Layers:** The kernel-level block IO (BIO) stack is the core of the entire storage stack. It implements a wide range of storage services using a layered approach. For example, we have a file system (FS) to satisfy file-level requests and BIO for block-level accesses. There are also many other storage functions such as RAID, IO priority scheduling, and caching to satisfy reliability and performance requirements. Additionally, with emerging software-defined flash designs, the kernel/OS also fulfills flash managements responsibility with a host-side "firmware" (*e.g.*, LightNVM). All these services are composable with each other. However, as hardware gets faster, the kernel level storage layers impose significant performance overhead and consume way too many CPU cycles. Worse, current OS's FS/BIO designs lack support for stable IO latencies, which we will aim to tackle by advocating new OS design principles (MittOS in §3.1) and proposing new frameworks to build cost-efficient storage services with performance guarantees (LeapIO in §4.1).

❹ **Software/Hardware Interface:** OS communicates with storage devices over PCIe bus for direct device management or through ethernet/fiber-channel fabrics for remote storage access. On top of the physical links, NVMe is the new de-facto storage protocol widely used in different scenarios (local/remote storage, VMs, etc.). It provides a lightweight queue pair model for IO submission and processing and has the potential to achieve more advanced functionalities. More details will be presented in §2.2 as it is a key element and repurposed/extended for different roles in several of our works (§3.2, §4.1, §5.3).

❺ **Hardware Devices:** Flash storage (in the form of SSDs) are largely replacing hard disk drives in real deployment due to their superior performance. Moreover, with the rise of hardware specialization, IO accelerators (*e.g.*, ARM- or FPGA based SmartSSDs/SmartNICs) are also being increasingly deployed to achieve efficiency. We will dive into SSD internals in §2.3 as an example to understand how hardware level intricacies will make performance predictability challenging to achieve. This dissertation build hardware-native solutions to exploit hardware capabilities for performance and efficiency (§3.3, §3.2, §4.1, §5.3).

2.2 NVMe Primer

NVM Express (NVMe) is the de-facto storage protocol widely used for today's fast storage devices, such as NAND Flash and 3D Xpoint SSDs. It utilizes a queue pair model to facilitate IO processing while exposing minimal software overhead. As shown in Figure 2.2, each queue pair consists of a Submission Queue (`SQ`) and a Completion Queue (`CQ`). The `SQ` and `CQ` are shared memory between the host and the device. Thus, they are directly accessible from both sides with no overhead. The host creates IO queues during the driver's initialization phase, and the number of queue pairs usually match the number of cores for lock-less access, better cache locality, and thus better IO scalability.

With NVMe, the IO processing logic is quite simple, with the following steps: ❶ *IO submission to the* `SQ`: IOs are encapsulated into NVMe commands by the driver. Each NVMe command takes 64 bytes and stores all the information which represent the IO, including `SLBA` (starting log-

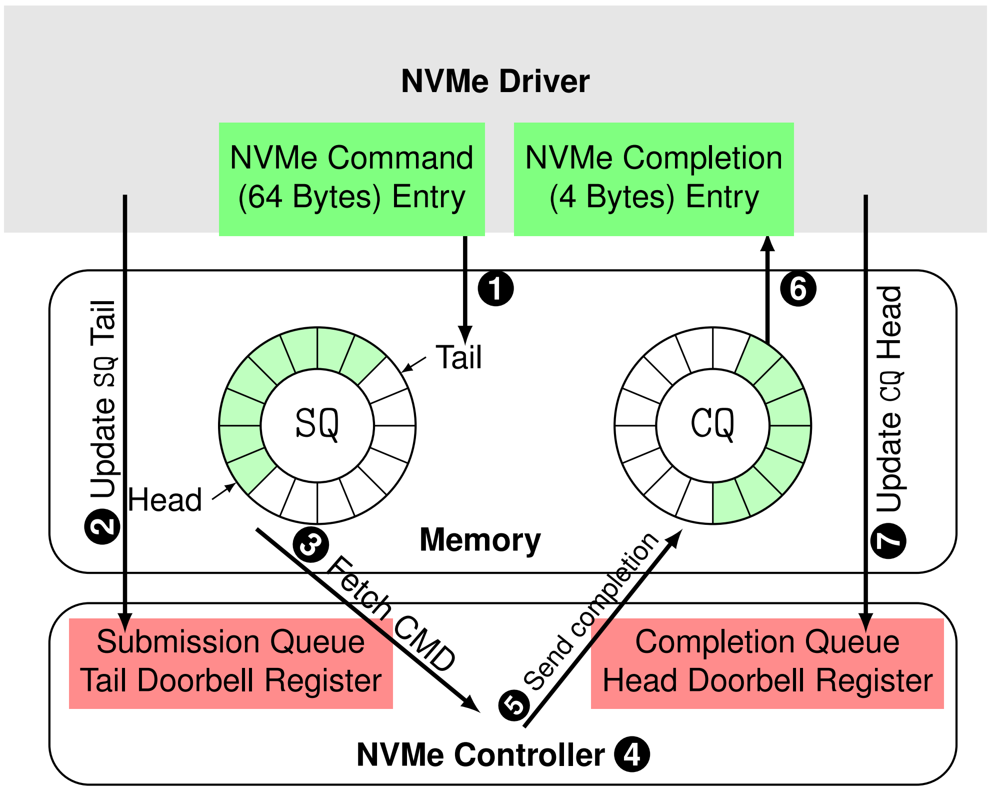

NVMe Driver
NVMe Command (64 Bytes) Entry
NVMe Completion (4 Bytes) Entry
❶
❻
Tail
SQ
CQ
Head
❷ Update SQ Tail
❼ Update CQ Head
Memory
❸ Fetch CMD
❺ Send completion
Submission Queue Tail Doorbell Register
Completion Queue Head Doorbell Register
NVMe Controller ❹

Figure 2.2: **NVMe Architecture.** *This figure presents the NVMe architecture, including the interaction between the host-side NVMe driver and the device-side NVMe device controller (e.g.,* FEMU, *LeapIO,* TEAFA*).* `SQ` *and* `CQ` *represent the submission and completion queues. The I/O processing logic is denoted as* ① *to* ⑦*, with a detailed explanation in §2.2.*

ical block address), `NLB` (number of logical blocks, i.e., IO length), `PRPs` (Physical Region Pages, *i.e.*, physical addresses of the IO buffers), and other meta- and management fields. The NVMe command entry is put into the tail of the `SQ`, indicating a new IO submission for the NVMe controller to process; ❷ *IO submission notification*: This step is for the host to notify the NVMe controller about new IO arrivals. This is done by writing the new tail position of the `SQ` to the corresponding doorbell register; ❸ *NVMe controller fetching NVMe command from the* `SQ`: In this step, NVMe commands in the queue will be fetched from head and then for ❹ *IO processing at the NVMe controller*, where it will be serviced, either by the caching layer or sent down to back-end storage media (*e.g.*, DRAM or NAND); ❺ *Adding IO completion entry to the* `CQ`: After an IO is done, the NVMe controller will compose an NVMe completion entry and put it to the tail of the

CQ. Then, the device will trigger an interrupt to the host, indicating to the host for IO completion processing; ❻ *Host side IO completion handling*: When the corresponding interrupt handler is scheduled (or when the host side proactively polls on the CQ), the host will fetch an entry from the CQ head and return the IO status back to the upper-level user applications; Lastly, ❼ *Updating the* CQ *head position*: Similar to ❷, this step is for the host to sync the new head position of the CQ with the device to avoid overflows.

NVMe is not just a low-level storage interface, it can be used to fulfill high-level goals. Later, we will show that how NVMe can serve as the key abstraction to bridge the semantic gap between different "worlds" (TEAFA in §3.2), and to offload a wide range of composable storage services (LeapIO in §4.1). Furthermore, we will demonstrate how polling based IO processing for NVMe IOs can improve IO performance significantly under different contexts (FEMU in Chapter 5, and LeapIO in §4.1).

2.3 SSD Primer

As shown in Chapter 1, complex device internals is a major cause of unpredictable latencies. Thus, before presenting our solutions, we first describe SSD internals that are essential for understanding the problem of unpredictability. In particular, this section describes how GC operates from the view of the physical hardware.

Figure 2.3 shows a basic SSD internal layout. Data and command transfers are sent via *parallel channels* ($C_1..C_N$). A channel connects multiple flash *planes*; 1–4 planes can be packaged as a single chip (dashed box). A plane contains *blocks* of flash *pages*. In every plane, there is a 4-KB *register* support; all flash reads/writes must transfer through the plane register. The controller is connected to a *capacitor-backed RAM* used for multiple purposes (*e.g.*, write buffering). For clarity, we use concrete parameter values shown in Table 3.2.

GC operation (4 main steps): When used-page count increases above a certain threshold (*e.g.*, 70%), a GC will start. A possible GC operation reads *valid* pages from an old block, writes them

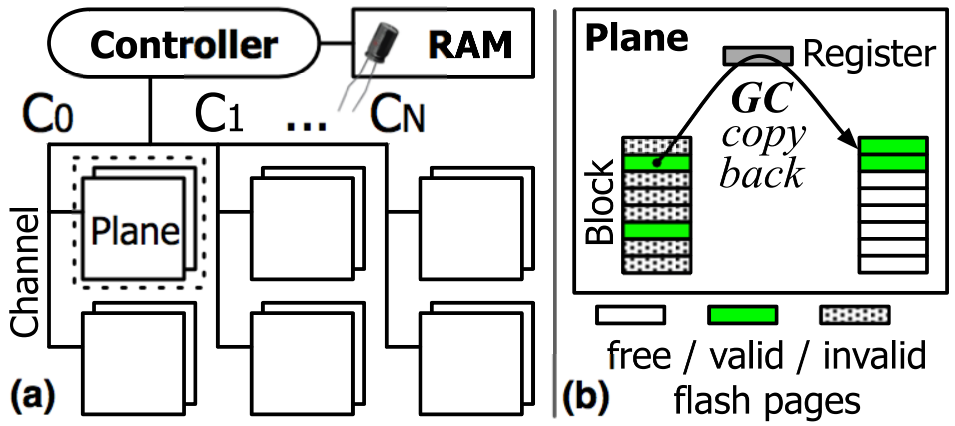


Figure 2.3: **SSD Internals (Section 2.3).**

to a free block, and erases the old block, within the same plane. Figure 2.3 shows two *copybacks* in a GC-ing plane (two valid pages being copied to a free block). Most importantly, with 4-KB register support in every plane, page copybacks happen *within* the GC-ing plane *without* using the channel [101].

The controller then performs the following *for-loop* of four steps for *every page copyback*: **(1)** send a flash-to-register read command through the channel (only 0.2μs) to the GC-ing plane, **(2)** *wait* until the plane executes the 1-page read command (40μs without using the channel), **(3)** send a register-to-flash write command, and **(4)** *wait* until the plane executes the 1-page write command (800μs without using the channel). Steps 1–4 are repeated until all valid pages are copied and then the old block is erased. The key point here is that copyback operations (steps 2 and 4; roughly 840μs) are done *internally* within the GC-ing plane *without* crossing the channel.

GC Blocking: GC blocking occurs when some resources (*e.g.*, controller, channel, planes) are used by a GC activity, which will delay subsequent requests, similar to head-of-line blocking. Blocking designs are used as they are simple and cheap (small gate counts). But because GC latencies are long, blocking designs can produce significant tail latencies.

One simple approach to implement GC is with a blocking controller. That is, even when only *one plane* is performing GC, the *controller is busy* communicating with the GC-ing plane and unable to serve outstanding I/Os that are designated to *any* other planes. We refer to this

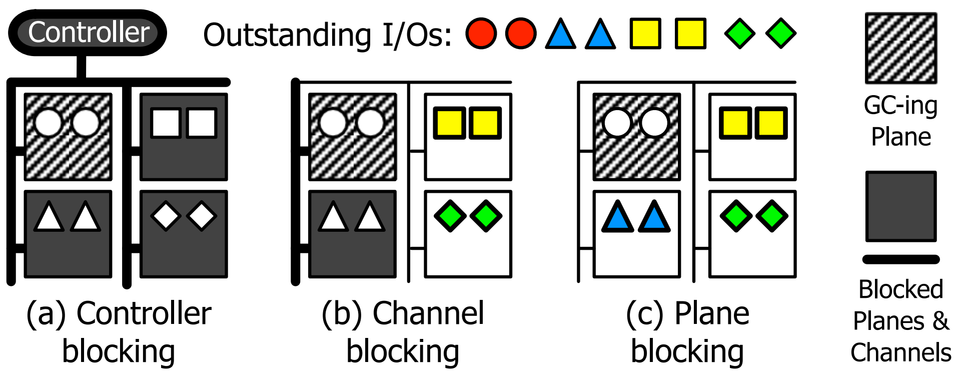


Figure 2.4: **Various levels of GC blocking.** *Colored I/Os in bright planes are servable while non-colored I/Os in dark planes are blocked.* **(a)** *In controller-blocking (§2.3), a GC blocks the controller/entire SSD.* **(b)** *In channel-blocking (§2.3), a GC blocks the channel connected to the GC-ing plane.* **(c)** *In plane-blocking, a GC only blocks the GC-ing plane.*

as *controller-blocking GC*, as illustrated in Figure 2.4a. Here, a single GC (the striped plane) blocks the controller, thus technically all channels and planes are blocked (the bold lines and dark planes). *All* outstanding I/Os cannot be served (represented by the non-colored I/Os). OpenSSD [46], VSSIM [325], and low-cost systems such as eMMC devices adopt this implementation.

Another approach is to support multi-threaded/multi-CPU with channel queueing. Here, while a thread/CPU is communicating to a GC-ing plane (in a for-loop) and blocking the plane's channel (*e.g.*, bold line in Figure 2.4b), other threads/CPUs can serve other I/Os designated to other channels (the colored I/Os in bright planes). We refer this as *channel-blocking GC* (*i.e.*, a GC blocks the channel of the GC-ing plane). SSDSim [169] and disksim$_{+SSD}$ [101] adopt this implementation. Commodity SSDs do not come with layout specifications, but from our experiments (§2.4), we suspect some form of channel-blocking (at least in client SSDs) exists.

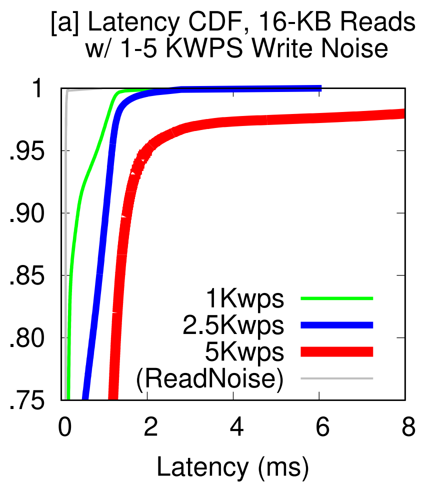
Figure 2.5 also implicitly shows how blocked I/Os create cascading queueing delays. Imagine the "Outstanding I/Os" represents a full device queue (*e.g.*, typically 32 I/Os). When this happens, the host OS cannot submit more I/Os, hence user I/Os are blocked in the OS queues. We show this impact in our evaluation.

2.4 GC-Induced Tail Latency

We present two experiments that show GC cascading impacts, which motivate our work. Each experiment runs on a late-2014 128GB Samsung SM951, which can sustain 70 "KWPS" (70K of 4KB random writes/sec). In Figure 2.5a, we ran a foreground thread that executes 16-KB random reads, concurrently with background threads that inject 4-KB random-write noises at 1, 2.5, and 5 KWPS (far below the max 70 KWPS) across three experiments. We measure L_i, the latency of every 16-KB foreground read. Figure 2.5a plots the CDF of L_i, clearly showing that *more frequent GCs (from more-intense random writes) block incoming reads and create longer tail latencies.* To show the tail is induced by GC, not queueing delays, we ran the same experiments but now with random-read noises (1, 2.5, and 5 KRPS. The read-noise results are plotted as the three overlapping thin lines marked "ReadNoise," which represents a perfect no-GC scenario. As shown, with 5 KWPS noise, read operations become 15×, 19×, and 96× slower compared to no-GC scenarios, at 90th, 95th and 99th percentiles, respectively.

In Figure 2.5b, we keep the 5-KWPS noise and now vary the I/O size of the foreground random reads (8, 16, 32, 64, and 128 KB across five experiments). Supposedly, a 2× larger read should only consume 2× longer latency. However, the figure shows that *GC induces more tail latencies in larger reads*. For example, at 85th percentile, a 64-KB read is 4× slower than a 32-KB read. The core of the problem is this: *if a single page of a large read is blocked by a GC, the entire read cannot complete*; as read size increases, the probability of one of the pages being blocked by GC also increases, as we explain later (§2.3). The pattern is more obvious when compared to the same experiments but with 5-KRPS noises (the five thin gray lines marked "ReadNoise").

For a fairer experiment, because flash read latency is typically 20× faster than write latency, we also ran read noises that are 20× more intense and another where read noises is 20× larger in size. The results are similar.



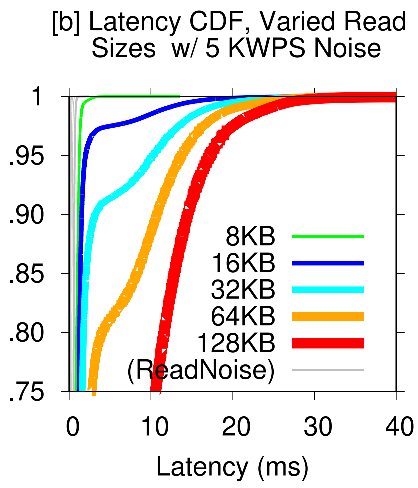


Figure 2.5: **GC-Induced Tail Latencies (Section 2.4) in a single SSD.**

2.5 Tail Latency in AFA

Background operations: From the previous section, we know that SSD firmware must perform many background management operations such as GC, wear leveling, internal write buffer eviction, and ECC checking/scrubbing. While many components inside an SSD exhibit parallelism, there are two places of serialization: channel and chip. When a channel is being used to transfer data, other IOs must wait. When a chip is reading data from the NAND medium to its page register (or writing from register to the NAND), other IOs must wait. Thus, when background operations are queued behind a channel or a chip, these operations will delay user (foreground) IOs, hence creating non-deterministic (background-induced tail) latencies.

Tail in flash array: Imagine a typical sequential large read to block addresses B_1 to B_4 that are striped across multiple SSDs. If one of them is “busy” (must wait for a background operation to finish), then the entire large read will be delayed. Figure 2.6a shows the cascading impact of a busy SSD (doing GC) to large user IOs.

Here, we form a RAID-0 on 4 real SSDs [31] with 4KB chunk size on which we run 16KB full-stripe reads (foreground). To trigger different intensities of GC (background) noises, we also inject random-write noises of 100, 200, 400, and 800 KWPS (kilo-writes-per-second) where “1W”

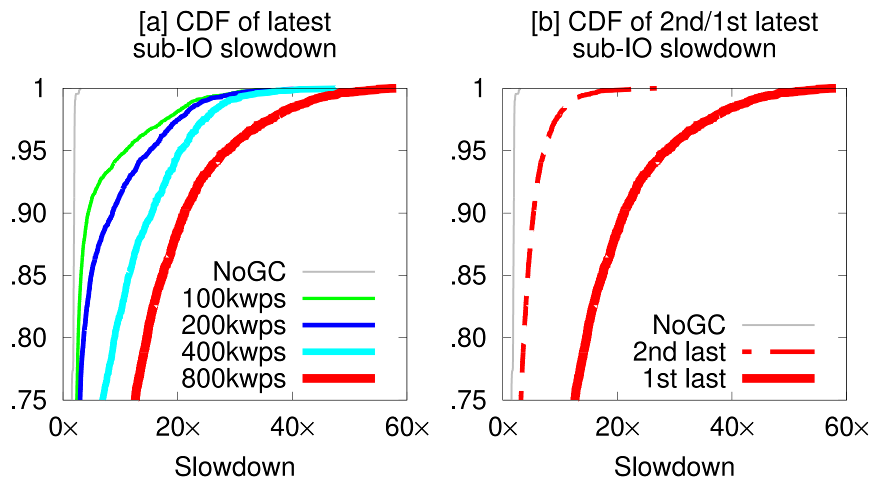


Figure 2.6: **DCSSD GC Impacts in SSD Array (Section 2.5).** *Each experiment runs on a RAID-0 of four Data Center SSDs.*

implies a 4KB random write. Every i^{th} full-stripe read generates 4 page **sub-IOs**.[1] We instrument Linux Software RAID to measure every sub-IO latency. Thus, for every i^{th} read, we measure 4 sub-IO latencies $L_{i1}..L_{i4}$ from the 4 SSDs. We then measure the longest delayed (latest) sub-IO with the following slowdown metric: $S_i{=}Max(L_{ij})/Median(L_{ij})$ where $j{=}1..4$. With 4 drives we use the 2nd earliest time as the median. Put simply, S_i represents the *slowdown to wait* for the latest returned page (the tail) in every full-stripe read i.

Figure 2.6a plots the CDF of all S_i, showing that due to background activities, *the latest sub-IO of a full-stripe I/O can arrive multiple times slower than the earlier ones.* The slowdown becomes worse when GC happens more often (100KWPS green vs. 800KWPS red line). *e.g.*, with 100KWPS, a sub-IO read is 13× slower than the median at p97.[2] Under 800KWPS, we see 23× slowdown at p98.

We emphasize that this slowdown is due to GC and not the random user writes. This is verified by the five (overlapping) thin gray lines marked "NoGC" where we convert the user write to a read noise. The gray lines mostly hovering around `x=1` essentially show that the foreground full-stripe

1. We use the term "sub-IOs" frequently in this paper. In Linux, a large read is broken to multiple *stripe-level reads*, and each stripe-level read is broken down to multiple **sub-IOs**, where a sub-IO represents a chunk read.

2. "**pY**" implies the Yth percentile (*e.g.*, p99 implies the 99th percentile, y=0.99 in the CDF graphs).

reads observe no (1×) slowdown of sub-IO completions. For a fairer experiment, as NAND read latency is around 20× faster than write latency, we also set the read noises to be 20× more intense or 20× larger in size and obtain similar results.

Opportunity: From the above experiment, we find a big opportunity to cut latency tail. To show this, we also record the slowdown of the *2nd latest* returned page: $S_i^2 = 2ndMax(L_{ij})/Median(L_{ij})$. Figure 2.6b compares the distribution of the 1st- and 2nd-latest slowdowns. For readability, we only show the experiment with 800KWPS noise.

As shown, *the probability that* $\underline{\textit{two}}$ *sub-IOs of a stripe read are* $\underline{\textit{simultaneously delayed}}$ *by GC is much lower than only* $\underline{\textit{one page}}$ *being blocked.* For example, >6× slowdown of the latest page happens *14%* of the time (x=6 at p86), but the 2nd-latest page is >6× slower *only 3%* of the time (x=6 at p97). Thus, if we put this finding in the context of RAID-4/5, *11%* of the slow IOs can be made fast by reconstructing the late sub-IOs from another SSD that holds the parity block of the stripe. This finding motivates TEAFA.

2.6 Cloud Storage Deployment Goals

Figure 2.7 paints the deployment goals required for the next-generation storage stack. As shown, the fundamental device abstraction is *NVMe virtual drive*, illustrated with a "●", behind which are the NVMe submission and completion queue pairs for I/O management. The expected deployment/use model can be seen in Figure 2.7a. Here a user mounts a virtual block drive ● to her VM (guest VM) just like a regular NVMe drive. We now elaborate the goals.

ⓐ **Fungibility and portability:** We need to keep servers fungible regardless of their acceleration/offloading capabilities. That is, we treat accelerators as *opportunities* rather than necessities. The storage software stack should be portable – able to run on x86 *or* in the SoC whenever available (*i.e.*, "offload ready") as Figure 2.7a illustrates. The user/guest VMs are agnostic to what is implementing the virtual drive.

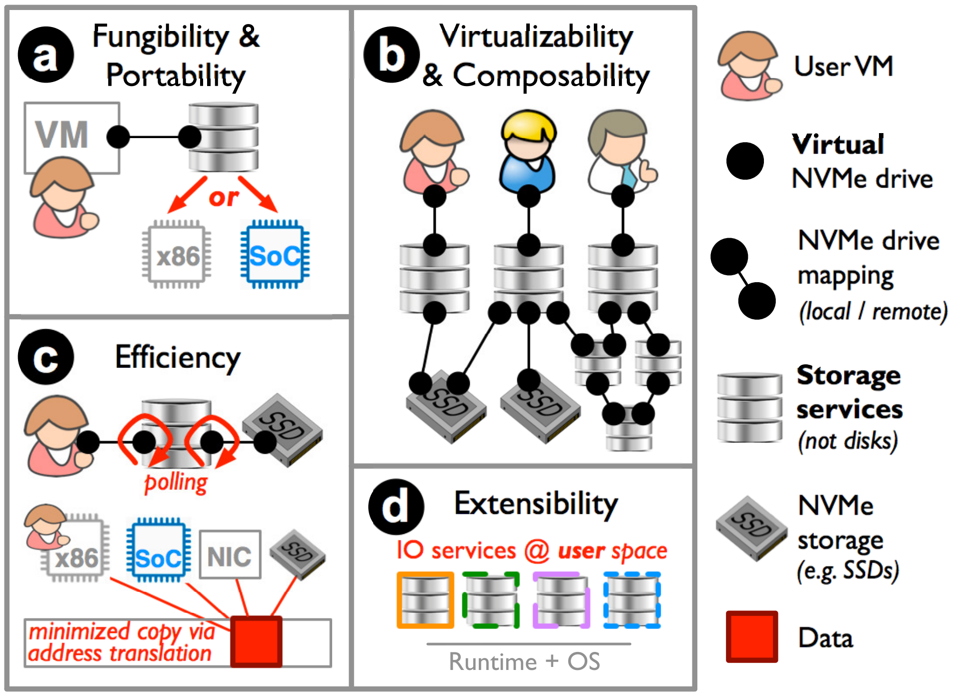

a
Fungibility & Portability
VM
or
x86
SoC
b
Virtualizability & Composability
SSD
SSD
c
Efficiency
SSD
polling
x86
SoC
NIC
SSD
minimized copy via address translation
d
Extensibility
IO services @ user space
Runtime + OS
User VM
Virtual NVMe drive
NVMe drive mapping (local / remote)
Storage services (not disks)
SSD
NVMe storage (e.g. SSDs)
Data

Figure 2.7: **Goals.** *As described in Section 2.6.*

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. As depicted in Figure 2.7b, 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 composed on top of other remote services.

Ⓒ **Efficiency:** It is important to deliver performance close to bare metal. The 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 can be achieved a uniform address space (Figure 2.7c).

ⓓ **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, ideally storage services should be implemented at the *user space* (Figure 2.7d), hence allowing cloud providers to easily manage, rapidly build, and communicate with a variety of (trusted) complex storage services.

CHAPTER 3

"TAIL-FREE" FLASH STORAGE STACK

In this chapter, we describe our holistic tail-free storage stack designs by revisiting the OS, software/hardware interface and hardware architectures to achieve predictable IO latencies. We start with MITTOS (§3.1), an OS level approach with a fast-rejecting SLO interface for data-parallel applications to achieve millisecond level latencies. Moving down to the lower levels of the storage stack, we then present TEAFA (§3.2) and TTFLASH (§3.3) at the software and hardware inteface and controller level to exploit data redundancy for predictable performance. We detail the designs and use cases of these three approaches. In §3.4, we discuss the implementations. Later, we will present detailed evaluations in §3.5, §3.6, and §3.7, respectively. Then we end this chapter with summaries of our three approaches for low and stable latencies in general (§3.8). We leave the discussions to Chapter 6 (§6.1–§6.3). The materials in this chapter are based on our papers "*TeaFA: A Tail-Evading Flash Array with a Gray-box IO Determinism Interface*" (Under submission) [229], "MITTOS*: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface*" (SOSP'17) [164], and "*Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs*" (FAST'17) [318].[1]

3.1 MITTOS: OS Support for Millisecond Tail Tolerance

MITTOS provides operating system support to cut millisecond-level tail latencies for data-parallel applications. In MITTOS, we advocate a new principle that operating system should quickly reject IOs that cannot be promptly served. To achieve this, MITTOS exposes a fast rejecting SLO-aware interface wherein applications can provide their SLOs (*e.g.*, IO deadlines). If MITTOS predicts that the IO SLOs cannot be met, MITTOS will promptly return EBUSY signal, allowing the application

1. I am the 2nd author in the MITTOS and TTFLASH papers. These two works are done in collaboration with other students at University of Chicago and the content of this chapter represents my contributions.

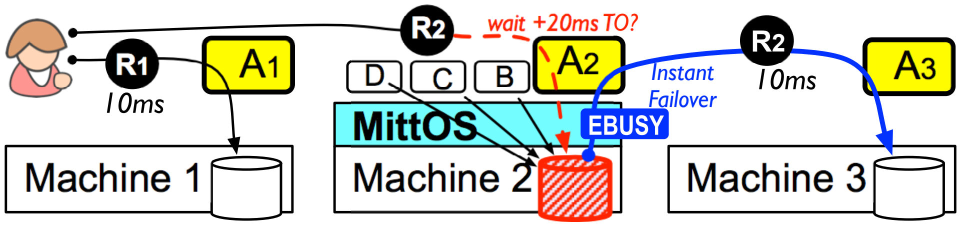


Figure 3.1: **MITTOS Deployment Model (§3.1.1).**

to failover (retry) to another less-busy node without waiting. We build MittOS within the storage stack, but the principle is extensible to CPU and runtime memory managements as well. MittOS' no-wait approach helps reduce IO completion time up to 35% compared to wait-then-speculate approaches.

3.1.1 Deployment Model and Use Case

MITTOS suits the deployment model of data-parallel frameworks running on multi-tenant machines, a common practice, as illustrated in Figure 5.6. Here, every machine has local storage resources (*e.g.*, disk) directly managed by the *host OS*. On top, different *tenants/applications* ($A...D$) share the same machine. Let us consider a single data-parallel framework (*e.g.*, MongoDB) deployed as applications $A_1 - A_3$ across machines #1-3 , and 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 might only be retried after some time (*e.g.*, 20ms) has elapsed, 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 R_2's completion time of only 10+e ms; e is a one-hop network overhead.

In the above model, MITTOS will be 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. We use the direct application-to-host model for faster research prototyping, but MITTOS principles will remain the same across these models.

3.1.2 Use Case

Figure 5.9 shows a simple use-case illustration of MITTOS. ① The application (*e.g.*, MongoDB) creates a deadline for a user. ② The application then tags `read()` calls with the deadline. ③ As the IO request enters a resource queue in the kernel, MITTOS checks if the deadline can be satisfied. ④ If the deadline 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.

As an additional note, a user request can span a stream of multiple `read()` calls, thus we also provide a deadline descriptor that can glue a sequence of `read()` operations (similar to stream/tx `begin()` and `end()` [263, 266]). Later, we discuss to what value a deadline should be set.

3.1.3 Goals / Principles

MITTOS advocates the following principles.

• *Busyness transparency ("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/replicas, thus there is no "shame" for the OS to admit busyness. In large-scale deployments, this principle works well, as the probability of all replicas busy at the same time is extremely low.

• *Instant feedback/failover:* The sub-ms instant feedback gives ms-level operations more flexibility to failover quickly. Making a system call and receiving `EBUSY` only takes $<5\mu s$ (③ and ④ in Figure 5.9). Failing over to another machine (⑤ in Figure 5.9) only involves one more network hop (*e.g.*, 0.3ms in EC2 and our testbed or even $10\mu s$ with Infiniband [256]).

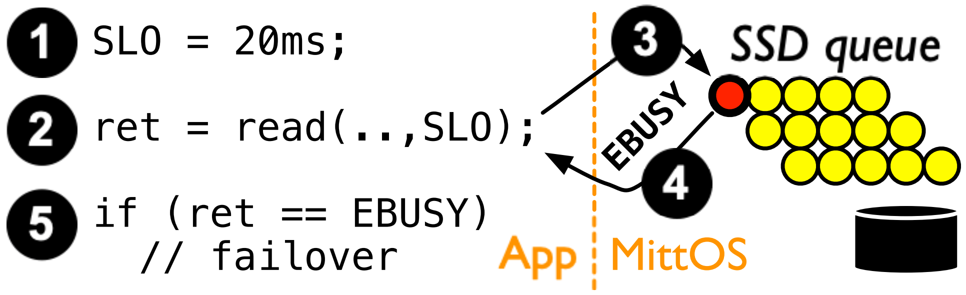


Figure 3.2: **MITTOS use-case illustration (§3.1.2).**

• *Keep existing OS policies:* MITTOS' simple interface extensions allow existing OS optimizations and policies to be preserved. MITTOS does not negate all prior advancements in the QoS literature. We only advocate that applications get notified when OS policies fail to meet user deadlines due to unexpected bursty contentions. For example, even with CFQ, IOs from high-priority processes occasionally must wait for lower-priority ones to finish; in SSD, garbage collection or wear-leveling activities can induce a background noise even with advanced isolation techniques.

• *Keep applications simple:* With MITTOS, prediction, cloning, hedging, and cancellation 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 5.9); 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 5.9).

3.1.4 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 and channels exhibit independent queueing delays. Furthermore, the target resource can be managed by different queuing disciplines (noop/FIFO, CFQ, anticipatory [173], etc.). Thus, EBUSY prediction will vary across different resources and schedulers. **(2)** In terms of performance overhead, latency prediction must 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 (different vendors) have different latency characteristics; seek cost varies across disks and flash programming time varies across SSDs, even across pages within the same chip.

3.1.5 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 (§3.1.6) and SSD (§3.1.7). In each integration, we describe how we address the three challenges (understanding the resource contention nature and fast and accurate latency prediction).

3.1.6 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 by the disk to its 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_{free}), as explained below. The arriving IO's wait time is simply the difference of the current and next free time ($T_{wait}=T_{free}-T_{now}$). If the disk is currently free ($T_{free}<T_{now}$), 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_{free}+=$T_{processNewIO}$). To make T_{free} 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. We omit the detailed discussion for space, as such a methodology is described extensively in disk literature [9, 272, 276]. 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. Our one-time profiling takes 11 hours (disk is slow).

T_{free} will automatically be calibrated when the disk is idle (T_{free}=T_{now}+$T_{processNewIO}$). However, under no-idle period, a slight prediction error in T_{free}+=$T_{processNewIO}$ 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_{free}+=T_{diff}).

We note that the disk firmware can reorder IOs based on their offsets, especially when noop is FIFO. However, with an advanced offset-based scheduling like CFQ, such inaccuracy will be reduced. In the evaluation, we will only evaluate MITTCFQ, which we describe next.

3.1.7 SSD Management (MITTSSD)

Latency variability in SSD is an ongoing problem [28, 35, 141]. 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 [167, 209, 220, 318] or isolation [170, 199]), none of them cover all possible cases. For example, under write bursts or no idle period, read requests can still be delayed significantly [220, §4][318, §6.6]. Even with isolation, occasional wear-leveling page movements will introduce a significant noise [170, §4.3].

Fortunately and typically, not all SSDs are busy at the same time, 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 [22]. 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 [104, 239], 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.

However, to achieve that, only the SSD firmware has the full knowledge of SSD internals. Fortunately, host-managed/software-defined flash [257] is gaining popularity and publicly available (*e.g.*, Linux LightNVM [126] on OpenChannel SSDs. 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. Second, T_{wait}=T_{now}−$T_{chipNextFree}$+ (60μs $\times$#$IO_{SameChannel}$). 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. Writes are discussed below.

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.

3.1.8 A Sample Application: MongoDB

Any application that employs data replication can leverage MITTOS with small modifications. As a sample application, we pick MongoDB (the "top NoSQL" [40]). 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. 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. 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.

3.2 TEAFA: Tail-evading Flash Array for IO Determinism

SSDs entered the industry looking like disks, a totally black-box device to the host. But, through decades of journey, the Storage Interfaces Technical Committees continuously make extensions to host-SSD interfaces, from `UNMAP` (official in 2007) [25], `TRIM` (2011) [27], `ATOMIC_WRITE` (2013) commands [29], to `STREAM` (2017) [44], from `ABC [241]`, `OpenChannel`(2016) [39] to `ZNS` (2020) [81].

After all these years, the committees finally accept the need to have an interface that can help applications observe stable latencies. This interface concept is known as *IO Determinism* (IOD) [84]. We are only aware of two specific commands proposed under this concept: `SET`, which informs the SSD to partition the chips physically, and `WINDOW`, which informs the SSD (to try its best) not to perform any background operation within the time window [260].

These IOD commands were standardized recently in June 2019 [83] but it is unclear how to program them, *e.g.*, what a proper window value is. A long time window will make the SSD run out of provisioned space and start GC, breaking the IOD contract. A short window of stable latencies is not attractive to users with long-running operations. This IOD concept opens up new interesting research questions.

We present TEAFA, a tail-evading flash array. Below we first describe the design principles and overview.

• *Leverage data redundancy in flash array for a timely read reconstruction.* As we target enterprise deployment where flash array is almost a de-facto storage solution, TEAFA exploits data redundancy for tail tolerance.

For example, in a RAID-5 flash array, redundancy exists in the form of parity data that enables a late (busy) chunk to be reconstructed using the parity and the rest of the chunks of the same stripe. Reconstruction is done by the "**host**" – in software-based RAID, the host implies the OS layer (*e.g.*, Linux Software RAID) and in hardware RAID, the host implies the RAID controller. More specifically, let us suppose a stripe consisting of 3 data chunks (B_0, B_1, B_2) and 1 parity chunk (P). If reading B_0 is slow, B_0 can be reconstructed by reading the parity and other chunks within the stripe ($B_0{=}B_1{\oplus}B_2{\oplus}P$). If the read is full stripe and a chunk B_i is slow, then a simple P can be read.

While leveraging redundancy for early reconstruction is straightforward, one must address the question of exactly *when* the host should perform this. If the host waits for a few milliseconds (detect stragglers first), the wait is too long for μs operations. On the other extreme, if the host always proactively spawn the extra reads in the beginning along with user reads, this will generate unnecessarily more load (*e.g.*, the SSDs are actually not busy).

• *Continue breaking the host-SSD semantic gap with the gray-box principle.* To solve the "when" problem above, a timely decision is needed. Continuing the spirit of gray-box approaches, we advocate the SSDs to minimally collaborate with the host on two matters: (a) signal the host when an IO is "busy" behind other background activities, such that the host can perform data reconstruction in a deterministic and timely manner and (b) negotiate with the host on a proper time window value such that the SSDs in the array can perform background operations in non-overlapping time windows. Breaking the semantic gap with these two simple ideas only requires us programming the IOD `WINDOW` value and using one bit in the NVMe submission/completion commands.

• *Do not intrusively modify the SSD (and applications).* We acknowledge that SSD vendors tend not to modify the FTL design too much as the current FTL policies have gone through years of deployment and massive changes would increase the time to deploy the techniques. Thus, it is not in our interest to modify the internal SSD management policies too intrusively. Furthermore, we also do not want to modify applications. To these ends, we carefully design TEAFA by keeping most changes in the host layer while only modifying the SSD side with just simple logics.

In the following subsections, we present TEAFA's three main techniques (§3.2.1-3.2.3).

3.2.1 Fast-Fail and Busy Bits

When an SSD is a sole storage, it must be best effort, serving user IOs as fast as it can, but occasionally still have to queue them behind background operations. However, if the SSD knows there is redundancy in other SSDs managed by the host, it "knows" there is another way to read (or reconstruct) the busy data. Here, we argue that it is acceptable for the SSD to return busy signals rather than queueing IOs for an indeterminate time. We do this in three steps.

First, we extend the base NVMe read *submission* command with a "**fast-fail**" bit (using a slot in the existing 64 reserved bits). When the RAID-based host submits user IOs to the underlying array, it always marks these original read IOs with fast-fail bit enabled.

Second, the SSD firmware processes IOs as usual when fast-fail bit is disabled, but upon seeing IOs with the fast-fail bit enabled, the firmware will quickly return (reject) the IOs when there are expensive background operations queued in front of them (in channel/chip-level queues). For this purpose, we extend the NVMe read *completion* message with a "**busy**" bit (using the reserved bits). In returning the busy reads, the SSD marks those IO completions with the busy bit enabled to distinguish them with normal returned IOs.

Finally, upon receiving back a busy read, the host will run the read reconstruction, by submitting additional IOs that we call *reconstruction IOs* (to differentiate them from the original user IOs). After reconstruction, the host can return the user IOs to upper layers and deem it completed.

A limitation of this approach is that it can only reconstruct k sub-IOs within a stripe where k is the number of parity blocks (*e.g.*, k=1 in RAID-5 and k=2 in RAID-6). Thus, it is still tail-prone when $>k$ sub-IOs are returned busy (*i.e.*, the reconstruction IOs are also returned with busy bits set). The subsequent sections will address this limitation.

Regardless of the limitation, this approach by itself is enough to deliver a large benefit, for two reasons. First, returning a busy read only takes 1μs through PCIe and the `xor`-based reconstruction takes less than 10μs on modern CPUs. Thus, this fast-fail notification (plus reconstruction) can provide orders of magnitude faster response than forcing the user to wait for background operations to complete. Second, as highlighted earlier (Figure 2.6b in §2.5), the probability that two sub-IOs of a stripe being delayed by two simultaneously busy SSDs is significantly smaller than only one sub-IO being delayed. This implies that the fast-fail approach is enough to circumvent the many single busy sub-IOs.

3.2.2 Shortest BG Remaining Time

We now enhance our first technique above for the cases where multiple busy sub-IOs are returned. Let us imagine a user full-stripe read to a RAID-5 (excluding the parity read) and one of the sub-IOs is returned with `busy=1`. The host would then send a reconstruction IO, the parity read with `fast_fail=1`, in order to reconstruct the busy sub-IO. Now, if the parity read also returns `busy=1`, the host must re-submit one of the two busy sub-IOs again, but the question is which one should be sent below.

The simplest is a random way: select one from the two, but with a 50% chance picking the longer delay. The problem here is that the wait time due to a background ("**BG**") operation can span a long time from "almost finish" (near 0 second) or "just begin" (*e.g.*, induce >60× slowdown when moving tens of valid pages as demonstrated earlier in §2.5).

A more effective approach is to re-submit the one with the *shortest BG remaining time*. This requires the SSD to inform the host such BG timing information. In TEAFA, we piggyback the remaining BG time (T_{rem}) in the NVMe completion message (using the 64 reserved bits). In precise definition, T_{rem} of a busy sub-IO is how long this read IO would have waited inside the SSD to get served (*i.e.*, until when the read command reaches the destination NAND chip).

Next, when the host receives two busy sub-IOs, it re-submits the one with the smallest T_{rem}. To generalize this, in the worst case where the RAID-5 host receives busy sub-IOs from all the N drives (including busy parity read), then it would resubmit $N-1$ reads with the least T_{rem}.

This kind of "gray-box information" [109, 158] is valuable because it does not reveal the internal details but yet is helpful for the host. Guessing the remaining time in a black-box way will be challenging due to the many vendor-specific implementations (different FTLs, GC algorithms, etc.). Inside the SSD, the firmware can easily estimate the BG remaining time. An SSD firmware typically maintains multiple queue structures for the parallel flash units. T_{rem} can be simply estimated by counting the number of pending background IOs in front of the user IO within the channel/chip queues. Prior works have shown that such a queue-based wait time is feasible to calculate [148, 149, 164].

We now address the question whether SSD vendors are willing to reveal such information. First, the T_{rem} technique can be made optional. Our earlier approach (§3.2.1) is powerful enough and can be combined with the next method (§3.2.3) where T_{rem} only helps in very corner cases, as explained later. Second, we argue that returning T_{rem} does not reveal more information beyond what users already see. Prior works already show that users can deconstruct many internal SSD layouts by simply deconstructing the user-observed latencies [153, 154, 334]. Third, if slight "obfuscation" is needed, T_{rem} can be designed to be a normalized number to alleviate potential timing channel attacks, similar to the chunk wearing information in the OpenChannel 2.0 specification [53].

3.2.3 BG Time Window

Our earlier approaches are effective when the probability of multiple sub-IOs delayed by concurrent BG operations is low. However, we observe a different case within a flash array design (in one of the co-authors' company) that is deployed by a large number of customers. The design absorbs user writes to a large, separate battery-backed RAM (outside the flash array). The host always flushes this external RAM to the array in the form of large sequential writes of the *same-size* (+/-1 block) to the underlying SSDs (*i.e.*, always full-stripe flushes). Hence, all the SSDs age at the same pace. Because SSDs of an array are usually the same model (same FTL logic), GC operations kick in at relatively the same time. This causes the host to see multiple busy sub-IOs returned with similar T_{rem} values. Handling cases like this is important for other similar flash-array designs that employ an external RAM and full-stripe flushes.

It would be ideal if background operations in the individual SSDs of an array are not overlapping in time. In RAID-5 for example, at most one SSD performs background operations in a given time window, allowing the RAID-5 host to always successfully circumvent at most one busy read.

To our advantage, an initial time window interface in IOD is already officially accepted (§2.5). However, we are not aware of published papers on how to program/set the time window value. Herein lies our contribution, specifically in the context of flash array. To program the time window value, TEAFA introduces two options: a static (SSD managed) and a dynamic window (host managed), described in detail in Section 3.2.4. We summarize them below.

In the SSD-managed **static**-window approach (§3.2.4), the SSDs use the internal proprietary information to inform the host a proper time window value ($\underline{T_{win}}$) to be used. For instance, T_{win}=1*sec* implies that the SSDs will take turns performing BG operations at 1s granularity. The host only needs to tell the SSDs their position in the array (*e.g.*, so that SSD$_1$ only performs BG operations between t=0..1*sec*, SSD$_2$ between t=1..2*sec*, and so on). This configuration is set during the RAID creation.

The host-managed **dynamic**-window approach (§3.2.4) does not require the SSDs to provide the window value. Rather, the window time will be initially set by the host using a random value (*e.g.*, 0.5*sec*) and will be adjusted dynamically live during the runtime to eventually reach a relatively optimal window size. On one hand, a large T_{win} will potentially starve the device (the

Sym.	*Meaning*	*Ex. values*
	Basic properties	
β	Valid page ratio in a block	25%
t_r	NAND page read latency	100μs
t_w	NAND page write latency	1500μs
t_e	NAND block erase latency	6ms
n_{pg}	# of pages in one NAND block	512
c_{blk}	NAND block size	8MB
n_{ch}	# of channels in the SSD	16
C_{ovp}	SSD over-provisioning space (5%)	100GB
N_d	# of devices in the RAID group	16
	Derived values	
T_{gc}	Time to garbage collect one block	0.4s
B_{gc}	GC bandwidth in a `WINDOW`	455MB/s
B_u	User write bandwidth in a `WINDOW`	1GB/s

Table 3.1: **Symbols (§3.2.4).** *The table explains the symbols used in* TEAFA*'s* `WINDOW` *algorithm. The top part are the basic properties of modern datacenter SSDs [77, 126] and the bottom part are the derived values. The right-most column shows example values we use in Section 3.2.4.*

provisioned space), as GC cannot be frequently triggered to reclaim enough free blocks. On the other hand, a small T_{win} (*e.g.*, less than one GC blocking time) is not enough to ensure one GC can finish within the T_{win} period, hence breaking the IOD expectation. A small T_{win} will also increase write amplification [214] as it forces GCs to happen more frequently, not letting the SSDs to absorb as many overwrites as possible.

3.2.4 Time Window Algorithms

We now discuss the algorithms behind our static and dynamic time windows.

• **SSD-Managed Static Window:** In this method, we advocate the SSD firmware to calculate the proper time window (T_{win}). The algorithm is based on the width of the RAID, SSD's NAND characteristics (read/write latencies), and firmware policies (*e.g.*, GC policies), as listed in Table 3.1. The host only needs to tell the SSDs the array width (N_d) and then the firmware gives the resulting T_{win} value to the host without exposing the internal proprietary information.

T_{win} must satisfy the following constraint:

$$T_{win} \leq C_{ovp} \, / \, (N_d \times B_u - B_{gc})$$

Here we consider a period with N_d time windows (*e.g.*, a RAID-5 setup with N_d devices). As only one SSD is allowed to perform GC on its own turn (one time window), the user writes can keep coming within the full "cycle" (a period of $N_d \times T_{win}$) until the SSD has a chance again to do GC. Thus, $N_d \times B_u$ represents the user write load for one SSD within a cycle. Within its time window, the SSD can run GCs freely to reclaim space, say at the speed of B_{gc} (expanded later below). This means $(N_d \times B_u) - B_{gc}$ is the net write load that an SSD should handle in a cycle. In other words, the net incoming write load should not take up all the free over-provisioned space that the SSD has (C_{ovp}).

All combined, the time window length (T_{win}) must be less than the size of the over-provisioned space (C_{ovp}) divided by the net write load, hence the constraint above. Given that C_{ovp} is typically a fixed size, T_{win} is mainly decided by N_d, B_u and B_{gc}. For example, under a wide RAID (large N_d), T_{win} must be set smaller to avoid breaking the IOD contract.

Now, let's further derive the GC speed (B_{gc}), which can be simply calculated as the amount of space reclaimed divided by the GC time. Suppose an average valid page ratio of β in one victim block, we can gain $(1-\beta) \times c_{blk}$ more space by cleaning one block, where c_{blk} is the NAND block size. With n_{ch} parallel channels, $n_{ch}\times$ of such size can be performed at the same time. Hence, B_{gc} is as follows:

$$B_{gc} = (1 - \beta) \times c_{blk} \times n_{ch} \, / \, T_{gc}$$

In the equation above, we introduce T_{gc} which is the time length to clean a block. The equation is as follows where the parameters can be found in Table 3.1. In a nutshell, T_{gc} depends on how many valid pages to move and the NAND read/write time, plus the block erase time.

$$T_{gc} = (t_r + t_w) \times \beta \times n_{pg} + t_e$$

Now, let's use the example parameter values in Table 3.1 to give a more concrete picture. For B_{gc}, the resulting value will be 455MB/s. Next, given a user load (B_u) and RAID size (N_d), we can get the safe range for T_{win}. For example, even under a bursty write load (B_u) of 1GB/s, roughly equal to 42 drive writes per day (DWPD), and 16 drives (N_d), $T_{win} \leq 6.4$s is safe enough. SSD vendors could follow our constraint above to find a proper T_{win} value for every model.

• **Host-Managed Dynamic Window:** In this approach, we do not add more work to the SSD, but rather have the host dynamically set the window value. Upon reboot, the host sets a base value B (*e.g.*, 0.5*sec*) and during runtime dynamically adjusts the value using a simple algorithm below, which is only *possible* given the busy signals supported in TEAFA (§3.2.1), hence showing the power of all of the approaches combined.

Every period of P (*e.g.*, 50ms), the host increases the value by I ms (*e.g.*, 10ms) as long as it does not see more than k busy sub-IOs within a stripe (k=1 in RAID-5; §3.2.1). In other words, as long as the host can always reconstruct up to k busy sub-IO(s) within a stripe, then the window value is deemed "safe," as it allows all the SSDs to have enough time to perform BG operations without overlapping each other in time. As mentioned earlier (§3.2.3), ideally, T_{win} is set as high as possible to reduce write amplification, hence the reason we increase the value gradually.

If more than k busy sub-IOs are observed, it implies that T_{win} is too large for the current user load, hence forcing the SSDs to execute some BG operations within the supposedly deterministic period and breaking the IOD expectation. For example, the internal RAM buffer or the over-provisioned NAND space is almost full, forcing a flush or GC to happen, respectively. When the host sees more than k busy sub-IOs, the host decreases the T_{win} by half[2] and informs the SSDs of the new window value.

We acknowledge that there are potentially many other possibilities to set the window value. For example, device performance likely deteriorates over time, thus even the static method requires window time recalibration. Above are our early attempts to figure out ways to program the window value and we find them simple and effective enough.

A very small T_{win} will reduce TEAFA back to the base case where there is no time window. This is because a too small time window won't be long enough for BG operations to finish, thus

2. Mimicking the TCP AIMD algorithm (additive increase multiplicative decrease) [60].

BG operations started from previous time window will last into next few time windows and still cause concurrent BG operations. Under such cases, TEAFA read construction won't help as it can only deterministically reconstruct one busy sub-IO. Here, we argue that T_{win} should also satisfy the following constraints:

$$T_{win} \geq b \times T_{gc}$$

Where b ($b \geq 1$) is a constant which implies the number of GCs that's allowed to happen per channel per time window. This guarantees that T_{win} is at least as long as one GC time, thus we won't see one GC from one SSD hogs two or more time windows. If more than one GC from each channel GCs ($b > 1$) need to be issued, the SSD vendor could use a relatively larger lower bound (larger b) for T_{win} or they can postpone the additional BG operations to the next time window of the SSD without breaking the IOD contract.

3.2.5 RAID size (N_d)

To guarantee that there won't be concurrent GCs happening in the RAID group at any time point under time window mechanism, we need to ensure that GCs in each SSD can be timely completed within its own time window. SSDs need to balance user writes which consume free pages and GCs which reclaim space, under which a steady state is maintained to guarantee the SSD not to run out of over-provisioning space. Thus, the basic constraint here is that there should be enough free pages cleaned by GCs to satisfy incoming user writes under a certain T_{win}.

Here, we try to answer: how large can N_d be without breaking TEAFA IOD promises?

We derive the maximum RAID size allowable (N_d) such that no concurrent BG operations will happen. That is, as we can only reconstruct one busy sub-IO, there should be *at most one SSD performing BG operations* at all time. Below we show that N_d=21 is safe even under *intensive* write. We first use concrete values. The basic parameters used are the same as Table 1 in our main submission, and then we give a detailed analysis.

We have provided a constraint about T_{win} in our main submission, where the over-provisioning space is used by for buffering bursty user writes. Here, we further *tighten* the constraint (by dropping the help from over-provisioning space) and consider a case where GC speed (B_{gc}) should match user load (B_u) to maintain each SSD's stable status. Thus,

$$N_d \times B_u \leq B_{gc}$$

In each "cycle" of $N_d \times T_{win}$ time, the incoming user write load is $N_d \times B_u \times T_{win}$, and the GC load is $B_{gc} \times T_{win}$, to satisfy the constraint we state above, user load should be equal to or smaller than GC load, thus we can get the above constraint formula.

According to this, we can get $N_d \leq B_{gc}$ / B_u. We have deducted that B_{gc} can be 455MB/s given an average case. As for B_u, even if we consider a 5 drive write per day guarantee from the SSD vendor, assuming a 2TB SSD, 5 DPWD roughly equals to 121MB/s (5×2TB/24hours). This means N_d can only be as high as 455/121<4.

However, with modern ONFI based NAND techniques which support multi-planes and multiple-registers in each plane, the GC speed (B_{gc}) actually can be much higher. To be specific, SSD controller can utilize multi-plane commands to read/write multiple NAND pages at almost the same cost of single t_r/t_w. Further, with multiple registers per plane, NAND die bandwidth can be much higher by interleaving data transfer through the channel and NAND read/program operations. Based on this, let's calculate the new B_{gc} assuming a 4-plane NAND die (*e.g.*, Toshiba A19 MLC NAND, [170]) with multiple-registers.

Given T_{gc} time, we can clean 4× more space since the 4 blocks in neighbor planes on the same die can be garbage collected simultaneously. Thus,

$$B_{gc} = 4 \times (1 - \beta) \times c_{blk} \times n_{ch} \ / \ T_{gc}$$

Similarly, here T_{gc} means the time needed for the above blocking cleaning operations. It's still mainly decided by the time to move valid pages in the victim blocks. But with multiple registers, the channel transfer time can be hidden from consecutive read/write operations. Under a standard ONFI 3.0 interface, transferring 4KB data through the channels takes roughly 15μs (260MB/s channel speed, [126]), thus it takes t_{ch}=60μs for a 16KB page. Hence, we have

$$T_{gc} = (t_r + t_w + 6 \times t_{ch}) \times \beta \times n_{pg} + t_e$$

In the above formula, the "$6\times t_{ch}$" part represents the extra time to transfer 3 extra set of page data from the other 3 planes for both reads and writes ($3\times2\times t_{ch}$). And the rest is the same as the T_{gc} calculation formula in the main submission.

Still plugging in the parameters in Table 1 in the main paper, we can get B_{gc}=1495MB/s. This will allow N_d to be as large as 12 under 5 DWPD or 21 under 3 DWPD, which is enough for building a large RAID in real deployment.

3.3 TTFLASH: Tiny-Tail Flash Controller

TTFLASH 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 copy-back operations.

We now present the design of TTFLASH, a new SSD architecture that achieves guaranteed performance close to a no-GC scenario. We are able to remove GC blocking at all levels with the following four key strategies:

1. Devise a non-blocking controller and channel protocol, pushing any resource blocking from a GC to only the affected planes. We call this fine-grained architecture *plane-blocking GC* (§3.3.1).

2. Exploit RAIN parity-based redundancy (§3.3.2) and combine it with GC information to proactively regenerate reads blocked by GC at the plane level, which we name *GC-tolerant read* (§3.3.3).

3. Schedule GC in a rotating fashion to enforce at most one GC in every plane group, such that no reads will see more than one GC; one parity can only "cut one tail." We name this *rotating GC* (§3.3.4).

Sizes		**Latencies**	
SSD Capacity	256 GB	Page Read	40μs
#Channels	8	(flash-to-register)	
#Planes/channel	8	Page Write	800μs
Plane size	4 GB	(register-to-flash)	
#Planes/chip	** 1	Page data transfer	100μs
#Blocks/plane	4096	(via channel)	
#Pages/block	256	Block erase	2 ms
Page size	4 KB		

Table 3.2: **SSD Parameters.** *This paper uses the above parameters. (**) 1 planes/chip is for simplicity of presentation and illustration. The latencies are based on average values; actual latencies can vary due to read retry, different voltages, etc. Flash reads/writes must use the plane register.*

4. Use capacitor-backed write buffer to deliver fast durable completion of writes, allowing them to be evicted to flash pages at a later time in GC-tolerant manner. We name this *GC-tolerant flush* (§3.3.5).

For clarity of description, the following sections will use concrete parameter values shown in Table 3.2.

3.3.1 Plane-Blocking GC (PB)

Controller- and channel-blocking GC are often adopted due to their simplicity of hardware implementation; a GC is essentially a `for`-loop of copyback commands. This simplicity, however, leads to severe tail latencies as independent planes are unnecessarily blocked. Channel-blocking is no better than controller-blocking GC for large I/Os; as every large I/O is typically striped across multiple channels, one GC-busy channel still blocks the entire I/O, negating the benefit of SSD parallelism. Furthermore, as SSD capacity increases, there will be more planes blocked in the same channel. Worse, GC period can be significantly long. A GC that copybacks 64 valid pages (25% valid) will lead to *54 ms* ($64 \times 840\mu$s) of blocked channel, which potentially leaves *hundreds* of other I/Os unservable. An outstanding read operation that supposedly only takes less than 100 μs is now delayed longer by order(s) of magnitude [28, 141].

To reduce this unnecessary blocking, we introduce *plane-blocking GC*, as illustrated in Figure

2.4c. Here, *the only outstanding I/Os blocked by a GC are the ones that correspond to the GC-ing plane* (○ labels). All I/Os to non-GCing planes (non-○ labels) are servable, including the ones in the same channel of the GC-ing plane. As a side note, plane-blocking GC can be interchangeably defined as chip-blocking GC; in this paper, we use 1 plane/chip for simplicity of presentation.

To implement this concept, the controller must perform a fine-grained I/O management. For illustration, let us consider the four GC steps (§2.3). In TTFLASH, after a controller CPU/thread sends the flash-to-register read/write command (Steps 1 and 3), it will *not* be idle waiting (for 40μs and 800μs, respectively) until the next step is executable. (Note that in a common implementation, the controller is idling due to the simple for-loop and the need to access the channel to check the plane's copyback status). With plane-blocking GC, after Steps 1 and 3 (send read/write commands), the controller creates a future event that marks the completion time. The controller can reliably predict how long the intra-plane read/write commands will finish (*e.g.*, 40 and 800 μs on average, respectively). To summarize, with plane-blocking GC, TTFLASH *overlaps* intra-plane copyback and channel usage for other outstanding I/Os. As shown in Figure 2.4c, for the duration of an intra-plane copyback (the striped/GC-ing plane), the controller can continue serving I/Os to other non-GCing planes in the corresponding channel (▲ I/Os).

Plane-blocking GC potentially frees up hundreds of previously blocked I/Os. However, there is an unsolved GC blocking issue and a new ramification. The unsolved GC blocking issue is that the I/Os to the GC-ing plane (○ labels in Figure 2.4c) are *still blocked* until the GC completes; in other words, with only plane-blocking, we cannot entirely remove GC blocking. The new ramification of plane-blocking is a potentially *prolonged* GC operation; when the GC-ing plane is ready to take another command (end of Steps 2 and 4), the controller/channel might still be in the middle of serving other I/Os, due to overlaps. For example, the controller cannot start GC write (Step 3) exactly 40 μs after GC read completes (Step 1), and similarly, the next GC read (Step 1) cannot start exactly 800 μs after the previous GC write. If GC is prolonged, I/Os to the GC-ing plane will be blocked longer. Fortunately, the two issues above can be masked with RAIN and GC-tolerant

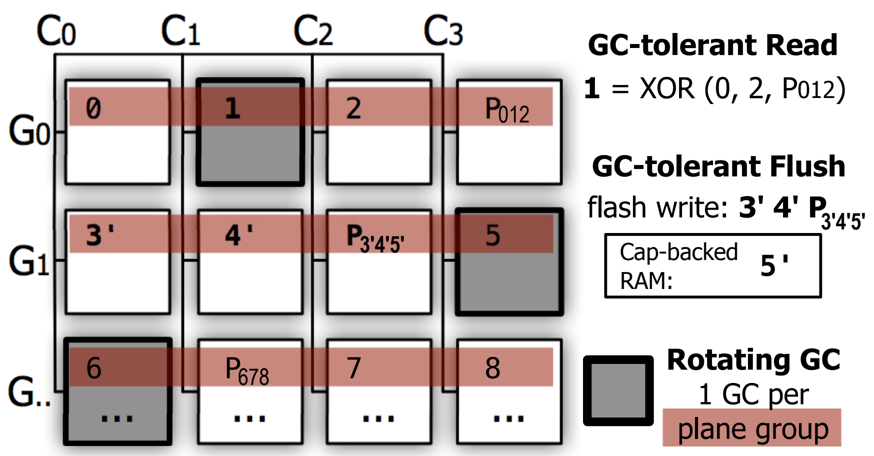


Figure 3.3: **TTFLASH Architecture.** *The figure illustrates our RAIN layout (§3.3.2), GC-tolerant read (§3.3.3), rotating GC (§3.3.4), and GC-tolerant flush (§3.3.5). We use four channels ($C_0 - C_3$) for simplicity of illustration. Planes at the same "vertical" position form a plane group (G_0, G_1, etc.). A RAIN stripe is based on $N-1$ LPNs and a parity page (e.g., $012P_{012}$).*

read.

3.3.2 RAIN

To prevent blocking of I/Os to GC-ing planes, we leverage RAIN, a recently-popular standard for data integrity [13, 21]. RAIN introduces the notion of parity pages inside the SSD. Just like the evolution of disk-based RAIDs, many RAIN layouts have been introduced [171, 200, 221, 223], but they mainly focus on data protection, write optimization, and wear leveling. On the contrary, we design a RAIN layout that also targets tail tolerance. This section briefly describes our basic RAIN layout, enough for understanding how it enables GC-tolerant read (§3.3.3).

Figure 3.3 shows our RAIN layout. For simplicity of illustration, we use 4 channels ($C_0 - C_3$) and the RAIN *stripe width* matches the channel count (N=4). The planes at the same position in each channel form a *plane group* (*e.g.*, G_1). A stripe of pages is based on logical page numbers (LPNs). For every stripe ($N-1$ consecutive LPNs), we allocate a parity page. For example, for LPNs 0-2, we allocate a parity page P_{012}.

Regarding the FTL design (LPN-to-PPN mapping), there are two options: dynamic or static. Dynamic mapping, where an LPN can be mapped to *any* PPN, is often used to speed-up writes (flexible destination). However, in modern SSDs, write latency issues are absorbed by capacitor-backed RAM (§3.3.5); thus, writes are spread across multiple channels. Second, dynamic mapping works well when individual pages are independent; however, RAIN pages are *stripe dependent*. With dynamic mapping, pages in a stripe can be placed behind one channel, which will underutilize channel parallelism.

Given the reasons above, we create a page-level hybrid static-dynamic mapping. The static allocation policies are: **(a)** an LPN is statically mapped to a plane (*e.g.*, LPN 0 to plane G_0C_0 in Figure 3.3), **(b)** $N-1$ consecutive LPNs and their parity form a stripe (*e.g.*, 012P$_{012}$), and **(c)** the stripe pages are mapped to planes across the channels within one plane group (*e.g.*, 012P$_{012}$ in G_0). Later, we will show how all of these are crucial for supporting GC-tolerant read (§3.3.3) and rotating GC (§3.3.4).

The dynamic allocation policy is: inside each plane/chip, an LPN can be dynamically mapped to *any* PPN (hundreds of thousands of choices). An overwrite to the same LPN will be redirected to a free page in the same plane (*e.g.*, overwrites to LPN 0 can be directed to any PPN inside G_0C_0 plane).

To prevent parity-channel bottleneck (akin to RAID-4 parity-disk bottleneck), we adopt RAID-5 with a slightly customized layout. First, we treat the set of channels as a RAID-5 group. For example, in Figure 3.3, P$_{012}$ and P$_{345}$ are in different channels, in a diagonal fashion. Second, as SSD planes form a 2-dimensional layout (G_iC_j) with wearout issues (unlike disk's "flat" LPNs), we need to ensure hot parity pages are spread out evenly.

3.3.3 GC-Tolerant Read (GTR)

With RAIN, we can easily support *GC-tolerant read* (*GTR*). For a full-stripe read (which uses $N-1$ channels), GTR is straightforward: *if a page cannot be fetched due to an ongoing GC, the page content is quickly regenerated by reading the parity from another plane*. In Figure 3.3, given a full-stripe read of LPNs 0–2, and if LPN 1 is unavailable temporarily, the content is rapidly regenerated by reading the parity (P_{012}). Thus, the full-stripe read is *not* affected by the ongoing GC. The resulting latency is *order(s) of magnitude faster* than waiting for GC completion; parity computation overhead only takes less than 3μs for $N \leq 8$ and the additional parity read only takes a minimum of 40+100μs (read+transfer latencies; Table 3.2) and does not introduce much contention.

For a partial-stripe read (R pages where $R < N-1$), GC-tolerant read will generate in total $N-R$ *extra* reads; the worst case is when $R=1$. These $N-R$ extra parallel reads will add contention to each of the $N-R$ channels, which might need to serve other outstanding I/Os. Thus, we only perform extra reads if $T_{GCtoComplete} > B \times (40+100)\mu$s where B is the number of busy channels in the $N-R$ extra reads (for non-busy channels the extra reads are free). In our experience, this simple policy cuts GC tail latencies effectively and fairly without introducing heavy contention. In the opposing end, a “greedy” approach that always performs extra reads causes high channel contention.

We emphasize that unlike tail-tolerant speculative execution, often defined as an optimization task that may *not* be actually needed, GC-tolerant read is *affirmative*, not speculative; the controller knows exactly when and where GC is happening and how long it will complete. GTR is effective but has a limitation: it does *not* work when *multiple* planes in a plane group perform GCs simultaneously, which we address with rotating GC.

3.3.4 Rotating GC (RGC)

As RAIN distributes I/Os evenly over all planes, multiple planes can reach the GC threshold and thus perform GCs simultaneously. For example, in Figure 3.3, if planes of LPNs 0 and 1 (G_0C_0 and G_0C_1) both perform GC, reading LPNs 0–2 will be delayed. The core issue is: one parity can only cut “one tail”. Double-parity RAIN is not used due to the larger space overhead.

To prevent this, we develop *rotating GC (RGC)*, which enforces that *at most one plane in each plane group can perform one GC at a time*. Concurrent GCs in different plane groups are still allowed (*e.g.*, one in each G_i as depicted in Figure 3.3). Note that rotating GC depends on our

RAIN layout that ensures every stripe to be statically mapped to a plane group.

We now emphasize our most important message: *there will be zero GC-blocked I/Os if rotating GC holds true all the time*. The issue here is that our rotating approach can delay a plane's GC as long as $(N-1)\times T_{gc}$ (the GC duration). During this period, when all the free pages are exhausted, *multiple* GCs in a plane group *must* execute concurrently. This could happen depending on the combination of N and the write intensity.

Employing a large stripe width (*e.g.*, N=32) is possible but can violate rotating GC, implying that GC tail latencies cannot be eliminated all the time. Thus, in many-channel (*e.g.*, 32) modern SSDs, we can keep N=8 or 16 (*e.g.*, create four 8-plane or two 16-plane groups across the planes within the same vertical position). Increasing N is unfavorable not only because of rotating GC violation, but also due to reduced reliability and the more extra I/Os generated for small reads by GTR (§3.3.3). In our evaluation, we use N=8, considering 1/8 parity overhead is bearable.

3.3.5 GC-Tolerant Flush (GTF)

So far, we only address read tails. Writes are more complex (*e.g.*, due to write randomness, read-and-modify parity update, and the need for durability). To handle write complexities, we leverage the fact that flash industry heavily employs *capacitor-backed RAM* as a durable write buffer (or "cap-backed RAM" for short) [24]. To prevent data loss, the RAM size is adjusted based on the capacitor discharge period after power failure; the size can range from tens to hundreds of MB, backed by "super capacitors" [20].

We adopt cap-backed RAM to absorb all writes quickly. When the buffer occupancy is above 80%, a *background flush* will run to evict some pages. When the buffer is full (*e.g.*, due to intensive large writes), a *foreground flush* will run, which will *block* incoming writes until some space is freed. The challenge to address here is that foreground flush can induce write tails when the evicted pages must be sent to GC-ing planes.

To address this, we introduce *GC-tolerant flush (GTF)*, which ensures that *page eviction is* *free from GC blocking, which is possible given rotating GC*. For example, in Figure 3.3, pages belonging to 3', 4' and $P_{3'4'5'}$ can be evicted from RAM to flash while page 5' eviction is delayed until the destination plane finishes the GC. With proven rotating GC, GTF can evict $N-1$ pages in every N pages per stripe without being blocked. Thus, the minimum RAM space needed for the pages yet to be flushed is small.

For partial-stripe writes, we perform the usual RAID read-modify-write eviction, but still without being blocked by GC. Let us imagine a worst-case scenario of updates to pages 7' and 8' in Figure 3.3. The new parity should be $P_{67'8'}$, which requires read of page 6 first. Despite page 6 being unreachable, it can be regenerated by reading the old pages P_{678}, 7, and 8, after which pages 7', 8', and $P_{67'8'}$ can be evicted.

We note that such an expensive parity update is rare as we prioritize the eviction of full-stripe dirty pages to non-GCing planes first and then full-stripe pages to mostly non-GCing planes with GTF. Next, we evict partial-stripe dirty pages to non-GCing planes and finally partial-stripe pages to mostly non-GCing planes with GTF. Compared to other eviction algorithms that focus on reducing write amplification [198], our method adds GC tail tolerance.

3.4 Implementation

3.4.1 MITTOS Implementation

MITTOS is implemented in 3260 LOC in Linux v4.10 (MITTNOOP, and MITTSSD in 1810, 1450 lines respectively, and an additional 50 lines for propagating deadline through the IO stack. These changes spread across 28 kernel files in 5 directories.

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 must add cloning and hedged-request features to MongoDB for another 210 LOC.

3.4.2 TEAFA Implementation

We first describe the platforms we use, and then the breakdown of TEAFA implementation, and other re-implementation of related works.

• **Platforms:** To implement TEAFA, we exhaustively tried many popular SSD research platforms from OpenSSD [46], DFC Card [41], FEMU [228], to LightNVM/OpenChannel-SSD (OCSSD) [126]. At the end only FEMU and LightNVM work as suitable platforms. We first discuss why OpenSSD and DFC are not an appropriate platform to implement TEAFA, and then we further describe our changes to LightNVM for TEAFA.

OpenSSD: The most ideal platform to implement TEAFA is the OpenSSD platform where we can modify the FTL logic and the NVMe interface implementation. However OpenSSD front-end programming framework is a *single-threaded* C implementation of the controller, which on the positive side speeds up FTL research development, but on the negative side not enabling more complex implementations. For example, in OpenSSD, when the SSD is doing a GC, the controller cannot be programmed to concurrently read the submission queue and return a busy signal. This simple programming model doesn't confirm with the ONFI controller specification that shows when a controller sends read/write device-level commands to the NAND chips, the event-based controller can proceed doing other concurrent operations. We tried many approaches to work around this, but at the end discard using OpenSSD.

DFC Card: Dragon Fire Card [41] is another SoC-based platform where the firmware changes can be implemented in the "mini" Linux on running on the SoC. Earlier DFC cards can directly manage NAND chips (the on-SoC Linux has an FTL driver), but it's no longer supported; The latest version of DFC cards directly attach to off-the-shelf SSDs where the FTL now resides in the SSD firmware, hard to modify. DFC cards lately are used as research platform to show near-storage processing, rather than pure FTL research.

FEMU (upgraded): This is a recent flash emulation platform (QEMU-based and DRAM-backed) [228] that allows applications and OS to run on top. It is reported to be more scalable than the popular VSSIM [325]. However, we must perform some fundamental modifications before we can build TEAFA here. First, FEMU default FTL's high computation overhead (adapted from VSSIM) causes inaccurate emulation under high user load. Thus, we (1) implemented a new page-level FTL optimized for FEMU emulation model, (2) offloaded the FTL logic into a separate polling thread to avoid interference from other management logics, and (3) re-implemented the data placement and GC policies taken from modern SSD designs [126, 265]. This whole upgrade requires *980 LOC*, but now FEMU can deliver a low and stable read latency of 40μs and 400 KIOPS. Second, we had to extend the firmware emulation with more basic features such as write buffering and flushing policies (*e.g.*, LRU with a balanced binary search tree) and preemptive GC policy in *740 LOC*, all of which was not supported in base FEMU. All of the upgrades and extensions now allow us to build TEAFA's firmware logic in FEMU as well as to rapidly re-implement other related works for evaluation purposes.

LightNVM+OCSSD: FEMU allows fast prototyping but one drawback is its DRAM-backed emulation nature (*i.e.*, not a real SSD). Because OpenSSD and DFC Card do not work for us, we have to resort to LightNVM+OCSSD. Thus, we also build TEAFA's firmware logic in LightNVM to show a real SSD evaluation. TEAFA implentation on the OCSSD is based on CNEX WestLake SDK. Our OCSSD uses the OpenChannel Spec 1.2 and runs LightNVM (the FTL) to manage NAND flash. LightNVM uses a state-of-the-art preemptive GC and a rate limiter to balance user and GC writes. OCSSD features host-managed IO scheduling, data placement and flexible preemptive GC policies, thus rendered a promising solution for solving the unpredictability problem associated with NAND flash. However, we find OCSSD/LightNVM has the following problems: (1) firmware prioritizes reads over writes (2) LightNVM limits inflight writes to each LUN (a.k.a, NAND die) to be at most 1 while allowing as many as reads submitted to the OC controller. Under this design, when using a mixed workload with r/w ration 2:1, we observe write throughput drop to **3MB/s** for whole SSD, if we kill the read workloads, the write throughput will be back to 1GB/s (under frequent GCs). We argue that base LightNVM design achieves best read performance at the cost of write bandwidth. Even in this case, user read might end up being queued behind by GC requests and experience tens of ms latencies. As NAND moving to 3D TLC And QLC with even 196 layers , program and erase latencies are getting worse, reaching 4ms and 10ms respectively by themselves. In this sense, it still makes sense to help save a read being blocked by writes or erases, using TEAFA reconstruction method. We argue the firmware's bias over writes doesn't follow the OpenChannel standard where the controller should execute command in FIFO and leave I/O scheduling to the host. Thus, our first effort is to work around the firmware problem by maintaining FIFO queue in LightNVM in 600 LOC. To be specific, LightNVM uses vector I/O commands, carrying at most 64 4KB sectors. While this reduces submission overhead, it makes performance monitoring to the underlying OC controller hard. Thus, we enhance LightNVM with a per-LUN queue. We insert a shim layer between pblk and NVMe driver to intercept pblk level IO requests and divide them into per-LUN level requests which will be managed by our per-LUN queue. In the IO completion path, we aggregate all the per-LUN requests back to user the original pblk request. This design doesn't introduce much overhead on the host as host CPU is powerful. LightNVM uses vector I/O commands to reduce command transmission overhead, however, this might impose load imbalance over all the parallel LUNs. The per-LUN queue design also makes it easier for people to design sophisticated I/O scheduling and data placement algorithms. The per-LUN queue works in FIFO manner to guarantee fairness. Due to the OC controller limitation on prioritizing reads over writes, we limit the number of inflight reads to be 1 if there is a write operation in front, thus making sure that write won't be starved by read operations. In addition, we also solved a severe bug of LightNVM when working with Linux software RAID which will cause system hangup. The problem is reported to and confirmed by LightNVM maintainers and we also submitted patches to fix it.

• **TEAFA software implementation:** This is broken down to the following layers.

The host (Linux Software RAID): Here we implement the TEAFA's host-side logic in 806 LOC in Linux 4.15. While the complexity is small, it took us months to address a couple of hurdles. First, when a busy read is returned, Linux triggers a complex timeout thread and a series of retries in the NVMe layer, hence we have to augment the driver code to distinguish between busy and real failure signals to bypass unnecessary IO retries. Second, to change the RAID-level IO processing logic for tracking the status of each busy IOs and their BG remaining time, we must change the complex Linux RAID state machine where originally every sub-IO path was highly diverged. In other words, each sub-IO is treated "independent", but in TEAFA, the status of every sub-IO must be monitored with respect to the other sub-IOs within the same stripe (for busy sub-IO counting and retry decision with/without fast-fail bits enabled).

The interface: We modify the Linux 4.15 NVMe driver to support TEAFA interface changes in just 18 LOC. As mentioned, the fast-fail and busy bits do not require a new bit position, but rather are "piggybacked" using the existing 64 free reserved bits in command/completion messages. While we focus on NVMe in this paper, we have also experimented with SATA, virtio and IDE interfaces, without breaking the RAID layer and FTL logic, demonstrating the simplicity of our interface changes. For SATA/IDE, due to lack of re-usable bits in the IO command structure (only 8-bit usable), we have to divide T_{rem} into 2^8 buckets, instead of passing back a real remaining time value.

The SSD "firmware": Modifying a real firmware can be done with OpenSSD or through collaboration with SSD vendors. Because of the limitations mentioned above, we implement the TEAFA firmware logic in 294 LOC in FEMU and 186 in LightNVM. The firmware logic is essentially for making fast-fail decisions and processing the fast-fail/busy bits. For computing the BG remaining time, we add a fine-grained IO status monitoring at the channel/chip queue, which is as simple as counting the number of requests in each queue and the remaining time of in-flight requests. As emphasized numerous times, TEAFA does *not* force any major policy changes such as re-architecting the GC process. As an additional note, although our firmware prototype is in software (FEMU and LightNVM), our industry partners agree that the simplicity of TEAFA's firmware logic can be easily added to a real firmware.

• **Re-implementation of other works** In our evaluation, we compare TEAFA with other works. But because other works use varying platforms (some of which even cannot run), "apples-to-apples" comparison would be difficult to make. Fortunately, with the upgraded FEMU platform, we were able to re-implement other works such as Harmonia [209], Flash on Rails [292], a proactive approach (more later) and Preemptive GC [220] in around 1400 LOC. Here we provide more details on how we implement related-work on FEMU platform.

Proactive: To avoid the uncertainty caused by the timeout threshold, Proactive method is more aggresive. Despite user IO size, Proactive always sends full-stripe reads down to all the devices in the RAID group. And it only waits for the first few returned IOs before marking the user IO as done and returning to upper layers. Here, if all the user IOs going to different devices return sooner than the rest extra issued IOs, we directly complete user IO and discard results from extra reads. Otherwise, if some user IOs are blocked by device-level BG operations and return late, we wait for enough parity IOs returning and proactively reconstruct the user read data. After reconstruction is done, we complete user IO and later discard late-returned user IOs. This simplies the overall design and is potentially more deterministic to ensure fast IO latencies, but at the cost of extra IO overhead to all the drives in the RAID group and some of the reads might overload the device considering their read data won't be used at all. We implement Proactive method in Linux software RAID, no other changes are needed.

Harmonia: To alleviate concurrent GC caused tail latencies, Harmonia [209] utilizes a global GC policy to coordinate GCs in different SSDs to start at the same time. Compared to uncoordinated GCs, Harmonia helps lower the overall possibility that IOs will be blocked by GCs, thus improving performance. In Harmonia, all GCs happen at the same time, we simply program the T_{win} appropriately. We implement Harmonia in FEMU with a global GC control logic. Whenever it detects GC is kicking in on SSD, it will proactively trigger GCs on other SSDs in the same RAID group. This way, we trigger GCs across all the SSDs in the RAID at the same time. Harmonia only requires FEMU modifications, where on top of each FEMU FTL structures, we simulate a global RAID controller logic for such GC synchronization logics.

Preemptve GC: With preemptive GC (PGC) [220], user reads can be interleaved with GC reads, writes and erases to alleviate GC inteference to user requests. This helps improve user request latency compared to conventional non-preemptive GC where user IOs might need to wait for the completion of garbage collection one or more NAND blocks. Under low capacity utilization, PGC can keep postponing GC requests to achieve low lantency for user requests. One user read can still be unlucky and delayed behind a write or erase, which takes at least several millisecond. In the worst case, when the amount of free space on the device runs critical, PGC has to be frequently kicked in and block one user request with multiple GC IOs. We add PGC into FEMU. While PGC logics sound simple, it brings several challenges when we integrate it to FEMU platform. First, FEMU utilizes a simplified NAND timing model, which makes it difficult for timing emulation in the PGC case. Second, FEMU doesn't have a low-level queue structure and asynchronous event model which can be used to "queue" PGC requests. We argument FEMU with support such support and queue user and GC requests fairly with a similar approach like rate-limiter in LightNVM. PGC only requires FEMU modifications.

Flash on Rails: Flash on Rails ("Rails" for short) [292] utilizes redundant data to improve performance. It put a subset of SSDs into read-only and write-only mode and switch their roles alternatively with a fixed time window size (*e.g.*, 5s). Rails requires a large write buffer to stage incoming writes before it can be safely flushed to the write-mode drives. Since all the reads are directed to the read-only SSD drives, Rails claimes to achieve read-only performance, without experiencing write or GC interferences. We implement Rails using two emulated FEMU SSD instances working in RAID 1 mode with a 10s time window size as used in the original paper. We used a large write buffer as required by Rails in front of FEMU FTL logics. While writes will be directed to and cached by the write buffer, and also written to the write-only FEMU drive directly, reads will be directed directly to the read-only FEMU drive, with no interference from writes at all. Later during role swapping, we flush the writes in the write buffer into the FEMU drive which was previously in read-only mode to maintain data consistency. After this, the two FEMU drives exchange their roles, *i.e.*, the previous read-only mode drive will now be the write-only drive. Rails only requires changes in FEMU. While Rails is better implemented in the host level, *e.g.*, the RAID controller or OS level as it needs to control user write buffering, we find doing on on top of FEMU is much easier with only user-level code changes, and we achieve similar read-only like performance for the workloads, as claimed in the original Rails papers.

This section describes our implementations of TTFLASH, TEAFA, and MITTOS, which are all publically available.

3.4.3 TTFLASH Implementation

• `ttFlash-Emu` **("VSSIM++"):** To run Linux kernel and file system benchmarks, we also port TTFLASH to VSSIM, a QEMU/KVM-based platform that "facilitates the implementation of the SSD firmware algorithms" [325]. VSSIM emulates NAND flash latencies on RAM disk. Unfortunately, VSSIM's implementation is based on 5-year old QEMU-v0.11 IDE interface, which only delivers 10K IOPS. Furthermore, as VSSIM is a single-threaded design, it essentially mimics a controller-blocking SSD (1K IOPS under GC).

These limitations led us to make major changes. First, we migrated VSSIM's single-threaded logic to a multi-threaded design within the QEMU AIO module, which enables us to implement channel-blocking. Second, we migrated this new design to a recent QEMU release (v2.6) and connected it to the PCIe/NVMe interface. Our modification, which we refer as *"VSSIM++"*, can sustain 50K IOPS. Finally, we port TTFLASH features to VSSIM++, which we refer as `ttFlash-Emu`, for a total of 886 LOC of changes.

Finally, we also investigated the LightNVM (OpenChannel SSD) QEMU test platform [45].

LightNVM [126] is an in-kernel framework that manages OpenChannel SSD (which exposes individual flash channels to the host, akin to Software-Defined Flash [257]). Currently, neither OpenChannel SSD nor LightNVM's QEMU test platform support intra-SSD copy-page command. Without such support and since GC is managed by the host OS, GC-ed pages must cross back and forth between the device and the host. This creates heavy background-vs-foreground I/O transfer contention between GC and user I/Os. For example, the user's maximum 50K IOPS can downgrade to 3K IOPS when GC is happening. We leave this integration for future work after the intra-SSD copy-page command is supported.

3.5 MittOS Evaluation

We use YCSB [139] 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 Mitt-SSD, 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 3.4 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.

3.5.1 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

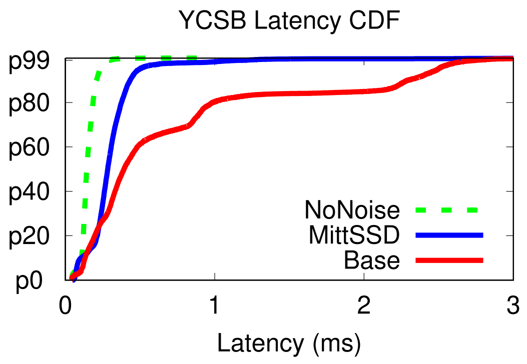


Figure 3.4: **Latency CDFs from microbenchmarks.** *The figures are explained in Section 3.5.1.*

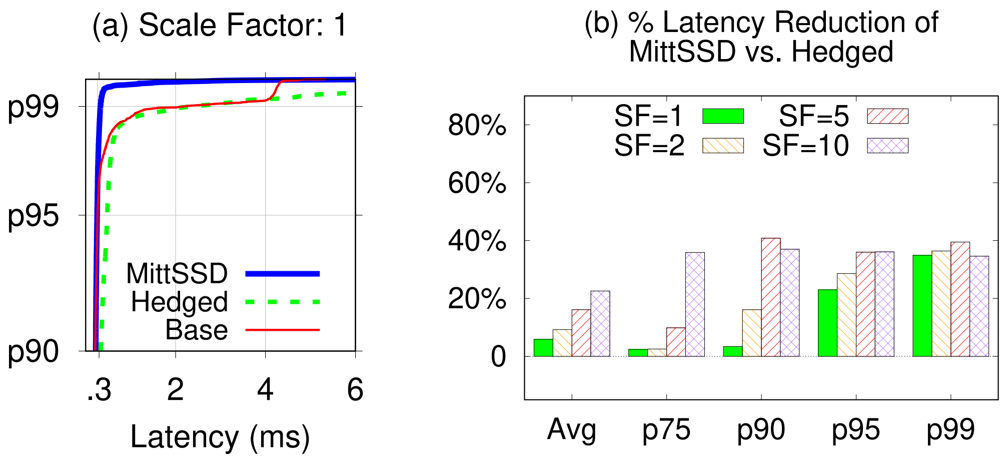


Figure 3.5: **MITTSSD vs. Hedged.** *The figures are explained in Section 3.5.2.*

directed to the noisy node.

Figure 3.4 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).

3.5.2 MITTSSD Results with Amazon EC2 Noise

For MITTSSD, we can only use our single OpenChannel SSD in one machine with 8 core-threads. We carefully (a) partition the SSD into 6 partitions with no overlapping channels, hence no con-

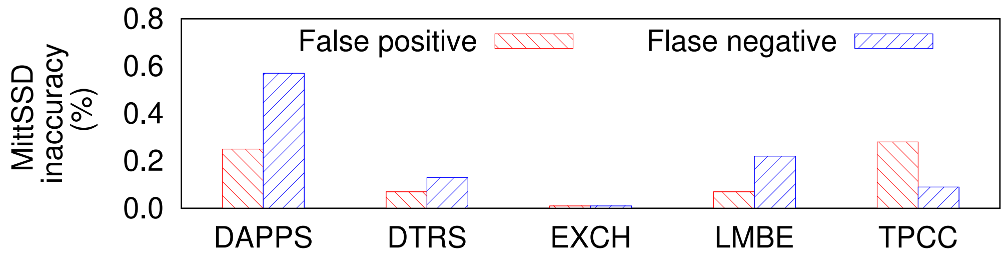


Figure 3.6: **Prediction inccuracy.** *(As explained in §3.5.3).*

tention 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, 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 3.5a), 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 thre-ads 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 3.5b shows the resulting % of latency reduction.

3.5.3 Prediction Accuracy

Figure 3.6 shows the results of 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 [193, §III][19]), 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 128× more 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 3.6 shows the % of false positives and negatives over all IOs. In total, MITTSSD inaccuracy is only up to 0.8%. Without the improvements (§3.1.7), 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 <1ms on average, for SSDs . We leave further optimizations as future work.

3.6 TEAFA Evaluation

TEAFA evaluation is divided into three sections: We first show the basic results of tail latency improvement brought by TEAFA approaches (§3.6.1). Then, we compare TEAFA with other related works (§3.6.4). Finally, we perform other extended evaluations (§3.6.5).

Platform setup: Most experiments are done on FEMU (for reasons mentioned in §3.4) running on Emulab D430 machines [68], equipped with two 8-core Intel Xeon E5-2630v3 CPUs at 2.4 GHz and 64GB DDR4 DRAM. LightNVM+OCSSD experiments are done on a local machine with a similar specification.

We configure a RAID-5 with Linux `mdadm` tool with a large stripe cache size and thread count to avoid software RAID-5 daemon being the performance bottleneck [30, 87]. The default 4KB chunk size is used as recommended for SSD arrays [23, 138].

We deploy 4 FEMU NVMe SSDs for the RAID-5. Each FEMU drive emulates 8 channels and 8 NAND chips per channel (backed by DRAM for the storage) with a total capacity of 12GB per drive (limited by the DRAM size). The SSD parameters are set to 1 plane/chip, 192 blocks/plane, 256 pages/block, 4KB page size, 40μs page-read, 200μs page-write, and 2ms block-erase. We use a standard baseline GC algorithm [169, 265, 325] with the optimum chip/plane-level GC blocking [228].

BG operations: We only cover the two most common background operations: GC and internal buffer flush. In most of the evaluation, we disable write buffering to cleanly show the impact of GC, but later we show the negative impacts of internal buffer flush and how TEAFA evades it. We did not induce other BG operations such as wear leveling as it is essentially similar to GC (internal data movement).

Workloads: We use the standard SNIA disk block traces [91] that we have re-rated 8-32 times more intense to reflect SSD workloads. We also use four new SSD traces from Microsoft data center, spanning cloud storage, search engine and database workloads. In these traces, we pick the 1-hour busiest period. Before running the workload, we follow the standard SNIA performance testing specification [92] to make FEMU SSDs into steady state.

3.6.1 Basic results of TEAFA Approaches under TPCC workload

This section shows the improvement made by the combination of the three TEAFA strategies, one at a time: "$\underline{\texttt{Tea}_1}$" represents only the fail-fast and busy-signal approach (§3.2.1), "$\underline{\texttt{Tea}_2}$" the one with background remaining time (§3.2.2), and "$\underline{\texttt{TeaFA}}$" the complete approach.

The complete approach `TeaFA` uses a static time window of 100 ms (§3.2.4), which is chosen to meet the constraint of at most one busy sub-IO per stripe given the SSD parameters we use above. Note that the 100ms value is lower than the example time window value in §3.2.4 because our FEMU drive is small (DRAM backed) while in §3.2.4, we use typical SSD parameters for general analysis.

For simplicity of figure presentation, we first show only the results of one workload, TPCC. Figure 3.7a shows the read latencies at major percentile values (p75 to p99.99) of five different approaches: **(1)** The red `Base` line represents the TPCC workload running on a 4-SSD RAID-5 without any tail evading strategies. As shown, starting at p95 (x=95) the `Base`'s latency starts increasing, which is consistent with what we see on real commodity SSDs (§2.5). **(2)** The orange $\texttt{Tea}_1$ line shows that by just cutting the longest tail (via data reconstruction as signaled by the

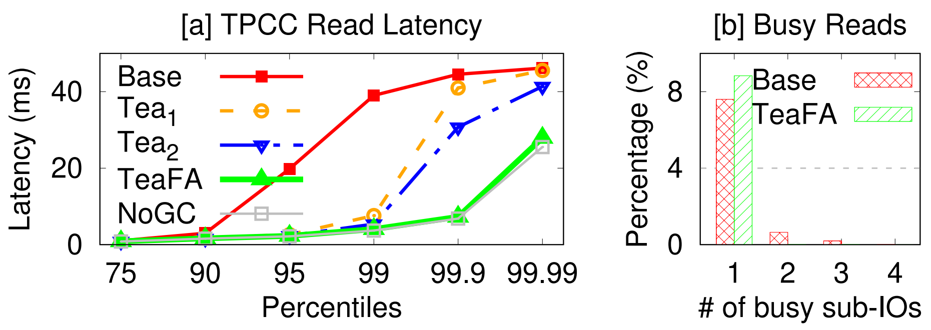


Figure 3.7: **TEAFA percentile latencies and #busy sub-IOs with TPCC (§3.6.1).** *Figure (a) shows the TPCC read latencies (in y-axis) at major percentiles p75 to p99.99 (in x-axis) with various* TEAFA *strategies (*`Tea`$_1$*,* `Tea`$_2$*, and* `TeaFA`*) compared to the* `Base` *line and the ideal* `NoGC` *case. Figure (b) shows the percentage of stripe-level reads (in y-axis) that experience 1 to 4 busy sub-IOs (in x-axis). It shows that* TEAFA *converts multiple concurrent busy sub-IOs to at most one busy sub-IO per stripe in RAID-5, thus always guaranteeing reconstructability of busy reads.*

busy bits), we significantly evade the latency tail up to p99, this is also consistent with our earlier opportunity analysis (§2.5). **(3)** The blue `Tea`$_2$ line shows that the background-remaining-time approach helps, although not significantly, because we cannot completely evade two busy sub-IOs, but only optimize the latency by choosing the fastest one. **(4)** The bold green `TeaFA` line shows that the time window approach can significantly cut further the tail area, even up to p99.99. Again, this is because the time-window approach attempts to guarantee at most one busy sub-IO per stripe. **(5)** The thin gray `NoGC` line shows the "ideal" read performance with GC delay emulation disabled in FEMU. The thin gap between the `NoGC` and `TeaFA` lines show the power of TEAFA in evading latency tail. Even at p99.99, TEAFA is only 9% slower than this ideal performance.

Figure 3.7b reveals the reason behind this tail-evading success. The x-axis shows how many sub-IOs of a stripe are returned with `busy=1`. As a reminder, in Linux, a large read is broken to many stripe-level reads, and each stripe-level read is broken down to chunk-level sub-IOs. Thus, with 4-drive RAID, we will only see a maximum of 4 sub-IOs (x=4) that can be busy-returned.

In Figure 3.7b, at x=1, the `Base` bar shows that roughly 7% of stripe-level reads experience 1 busy sub-IO, but since the base approach just waits (does not reconstruct) busy sub-IOs, we can see the `Base` line in Figure 3.7a starts to increase between the p90 and p99 values (100-7%). However, in `Tea`$_1$, `Tea`$_2$ to `TeaFA` approaches, this is not an issue, because they can reconstruct easily one busy sub-IO.

Still in Figure 3.7b, at x=2, the Base bar shows that almost 1% of the stripe-level reads experience 2 busy sub-IOs. The $\texttt{Tea}_2$'s SBRT approach cannot completely evade these ≥2 busy sub-IOs, hence the $\texttt{Tea}_2$ line in Figure 3.7a starts increasing between the p99 and p99.9 values.

Now with the complete TeaFA's time window approach, the green TeaFA bar in Figure 3.7b shows that the *time-window approach successfully* ***shifts*** *the concurrent GCs across time* such that *at most* there is only *one* busy sub-IO per stripe. Hence, the TeaFA bar is higher than the Base bar, reaching y=8% at x=1 but y=0 at x>1. In other words, it is acceptable to see a higher percentage of one busy sub-IO (reconstructable) as long as we evade multiple busy sub-IOs.

In Figure 3.8, we present the tail latencies at p99.99. This is the complementary results to p99 and p99.9 tail latencies which are already shown in our main submission. Here, again, the p99.99 matches the conclusion we get from p99 and p99.9, that by integrating all TEAFA strategies, we can effectively trim the tail latencies at p99.99. Tea_1 and Tea_2 don't help much because at this high percentile, the tails cannot be easily trimmed without the help from time window.

3.6.2 Basic Results of TEAFA Approaches under All Workloads

Figure 3.8 shows the overall result with all the workloads. To simplify the figure, we only show latency values at p99 (top graph) and p99.9 (bottom one). Overall, the TeaFA bars summarize that TEAFA, between the p99 and p99.9 percentiles, delivers on average 3.5×, and up to 9.6× faster latencies compared to the base approach, and only 1.1× to 2.1× slower than the ideal NoGC case.

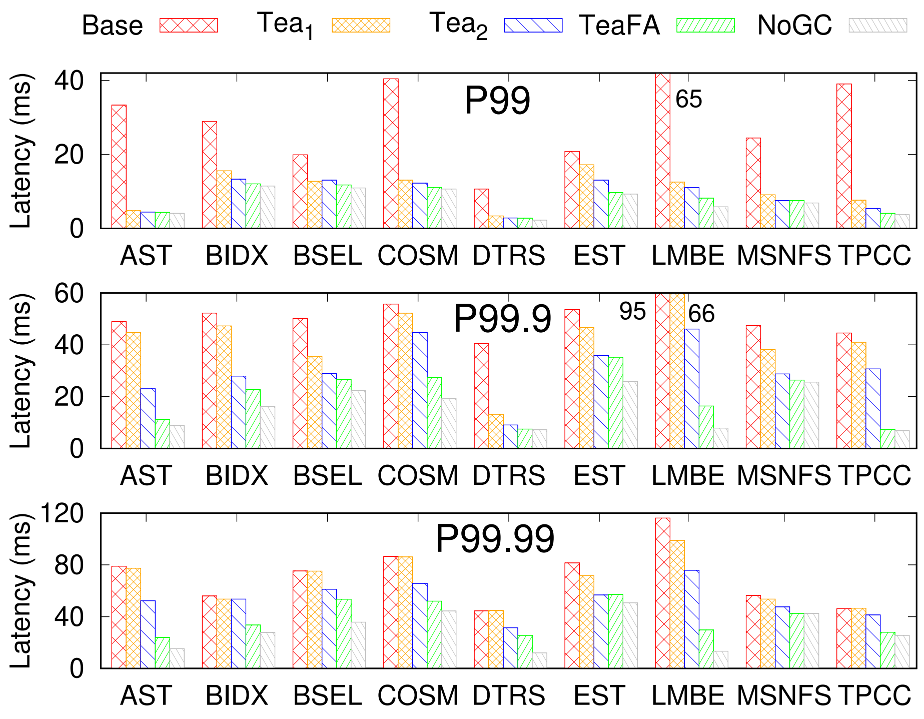


Figure 3.8: **TEAFA p99, p99.9, and p99.99 latencies with all the workloads (§3.6.2).** *The figures show the p99 (top figure), p99.9 (center) and p99.99 (bottom) latencies of all the workloads under different* TEAFA *strategies including the* `Base` *and* `NoGC` *cases. It shows that* TEAFA *successfully evades tail latencies and almost reaches the* `NoGC` *values at high percentiles.*

In Table 3.3, we present the detailed characteristics of the trace workloads used in our main submission. The table covers major characteristics such as read/write ratio, average read/write size and IO intensity (IO inter-arrival time and IOPS). Note that the numbers in the table only captures the average view of the whole workload as each workload characteristics change with time. *e.g.*, for BSEL, even if the average inter-arrival time is over 20ms, it doesn't imply this workload is not intensive at all, instead, this workload contains many large IOs (>1MB) and the IOPS at certain time range can reach thousands.

While SNIA block traces are well described, our 4 new traces from Microsoft data centers need further description. "AST" represents AzureStorage traces collected from Microsoft Azure storage backend, "BIDX/BSEL" are the Bing search engine indexing workloads, and "COSM" are the workloads from Microsoft CosmosDB KV store.

For readers who are interested to see the full CDF graphs, we put them in Figure 3.9. Note that

Workload	#IOs	Read%	Read size	Write size	IO Max	Inter-time
AST	320K	18%	24KB	20KB	64KB	1417μs
BIDX	21K	36%	60KB	104KB	288KB	6974μs
BSEL	164K	4%	260KB	78KB	11MB	21951μs
COSM	403K	8%	214KB	91KB	16MB	8936μs
DTRS	147K	72%	42KB	53KB	64KB	2034μs
EST	71K	24%	15KB	43KB	1MB	8445μs
LMBE	1112K	89%	12KB	191KB	192KB	539μs
MSNFS	487K	74%	8KB	128KB	128KB	3696μs
TPCC	513K	64%	8KB	137KB	4MB	721μs

Table 3.3: **Block trace charateristics.** *This table show the detailed characteristics of the blocks traces we use. "#IOs" represents the total number of IOs in the trace, "Read%" means the percentage of reads in each trace, and "Read size, Write size, IO Max" represents the average read, write and max IO size. "Inter-time" means the average inter-arrival time between two consecutive IOs in each trace*

different workloads have different characteristics, hence the various gap sizes between the lines. However, all of them point to a consistent result that each of the TEAFA strategies move the CDF line to the upper left (better) with the `TeaFA` line closest to the `NoGC` line.

Figure 3.10 shows the percentage of stripe-level reads that observe busy sub-IOs (from `1busy` to `4busy`). The top figure 3.10a shows that the `Base` approach observes many multiple busy sub-IOs in many of the workloads, *e.g.*, a high percentage of concurrent `3busy` sub-IOs appears in `COSM` and `TPCC` workloads (see the orange bars). The bottom figure 3.10b again shows that TEAFA successfully shifts the concurrent GCs across time (higher `1busy` bars with almost no `2-4busy` bars).

In Figure 3.10b, one can see that for `COSM` and `LMBE`, we can see small `2-4busy` bars ($y<0.05\%$ on average). Upon further investigation, this is *not* because of a too-high time-window value (100ms in this case). Rather, because the workloads are write intensive and GCs kick in most of the time, there are left-over GCs that started *just before* and finished *slightly after* the time-window expires in the corresponding SSD. This can be fixed easily in the future by making the GC page movement and block erase only start if the time estimate does not pass the expiration time.

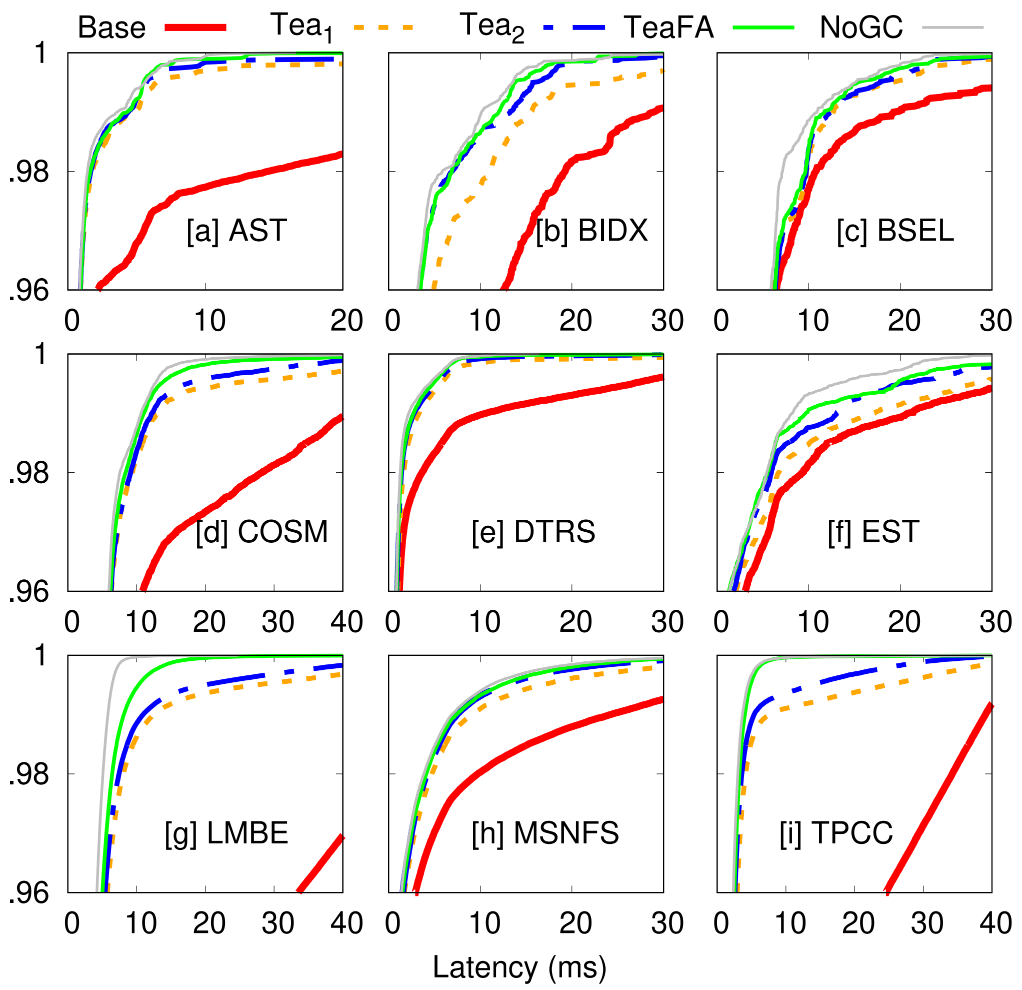


Figure 3.9: **Tail Latencies in CDF (§3.6.2).** *The figures represent the same experiments in Figure 3.8. The figures basically show a more complete picture (in CDF above p96) of the p99 and p99.9 values shown in Figure 3.8.*

3.6.3 Results from Real Storage Applications

This subsection shows the results from running 6 different FileBench workloads and demonstrates that TEAFA can help with application perceived latencies. We report the average read latencies here as FileBench doesn't support reporting per-IO latency. As depicted in Figure 3.11, we see latencies get consistently better after applying different TEAFA techniques. This aligns well with previous trace-based experiments to prove TEAFA's ability to benefit applications in achieving better performance.

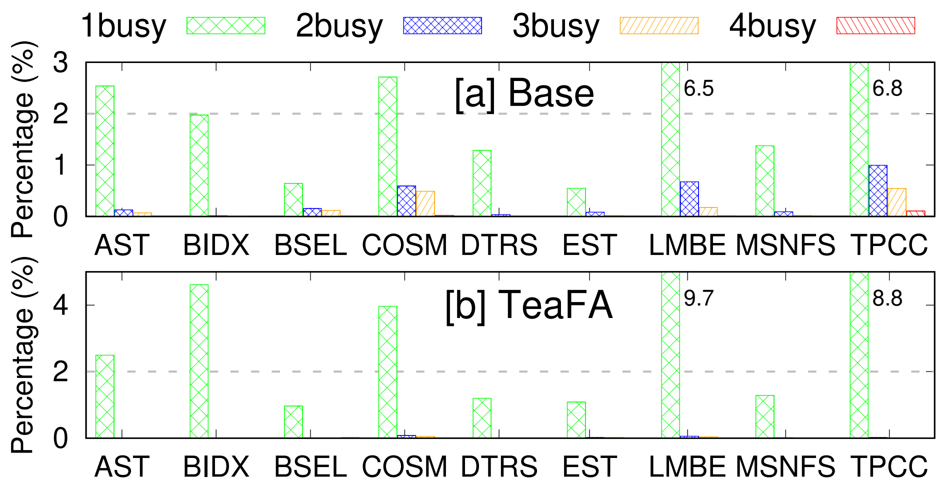


Figure 3.10: **#Busy sub-IOs in all the workloads (§3.6.2).** *These bar figures represent the same bar figure in Figure 3.7b (that only shows TPCC), but now we show the results from all workloads. The* `1busy` *to* `4busy` *bars represent the number of busy sub-IOs per stripe, and the y-axis shows the percentage of stripe-level reads that experience busy sub-IOs. The bottom figure essentially shows that* TEAFA *successfully shifts multiple concurrent* `2-4busy` *sub-IOs to* `1busy` *sub-IOs.*

3.6.4 Versus Other Works

We now compare TEAFA with other related works. Due to space constraints, we only show one benchmark TPCC (we reach the same observation in other benchmarks).

Comparison with Proactive (Always Full Stripe): A simple black-box way to cut 1-busy sub-IOs is to always proactively send a full-stripe read including the parity read. Hence, out of every N sub-IOs, the stripe-read completes when the first $N-1$ sub-IOs finishes. Figure 3.12a shows the comparison. `Proactive` is effective as it can reconstruct 1-busy sub-IOs but it still loses to TEAFA at high percentiles due to its inability to evade concurrent busy sub-IOs. `Proactive` also negatively adds more load; Figure 3.12b shows that `Proactive` sends down 2.4× more IOs to the base case, while the `TeaFA` bar only issues 6% more reads.

Comparison with Harmonia: Harmonia [220] is an approach that tells the SSDs in an array to do GCs at the same time (*i.e.*, a synchronized GC in flash array). We observe that Harmonia is able to improve the overall average latency by 27% compared to the baseline, but it fails to cut tail latencies as effective as TEAFA, as shown in Figure 3.12c. In Harmonia, a small percentage of

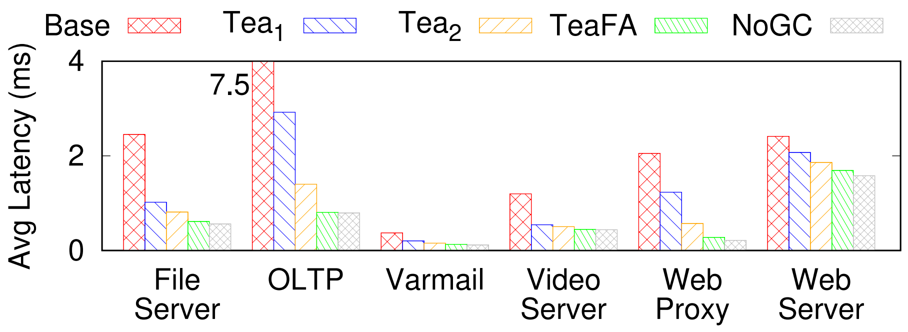


Figure 3.11: **Average latencies of different Filebench workloads (§3.6.3).** *The figures represent experiments by running 6 different storage workloads using Filebench on top of ext4 file system. We report the average read latencies as Filebench doesn't support per-IO latency tracking.*

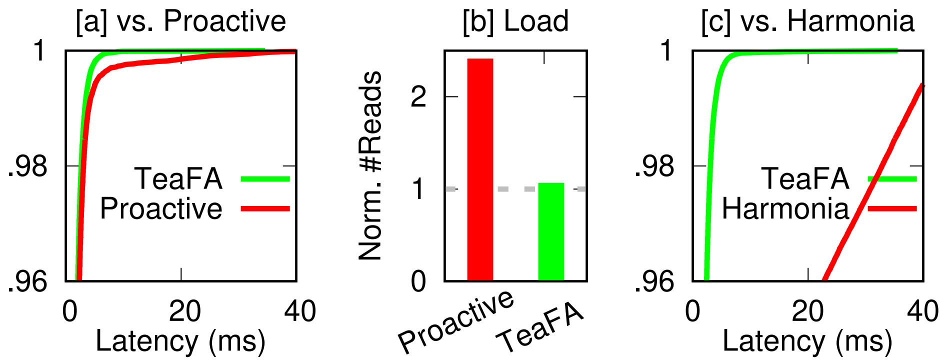


Figure 3.12: **vs. Proactive and Harmonia (§3.6.4-3.6.4).** *Figure (a) compares the TPCC CDF latencies on* TEAFA *vs. Proactive. Figure (b) shows the normalized number of IOs that Proactive and* TEAFA *send down to the flash array (2.4× vs. 6%). Figure (c) compares the TPCC latencies on* TEAFA *vs. Harmonia.*

IOs must wait when all the SSDs are doing GC concurrently.

Comparison with Flash-on-Rails: Flash on Rails [292] is a tail-cutting technique that divides the SSDs of an array into read-only and write-only SSDs, and performs read-write role swapping periodically. A similar strategy can also be found in Gecko [286] and SWAN [201]. Figure 3.13a shows that Rails is indeed able to deliver a pure read-only latency to users (the left-most line). Here, TEAFA loses to Rails because in our current setup we have not enabled internal write buffering, hence user reads in TEAFA can be queued behind the longer *user* writes. Rails on the other hand ensures pure read latency.

However, Rails has downsides. The first is reduced read throughput. As Rails break the read-write role of the SSDs separately, there are fewer number of devices to serve reads. Figure 3.13b

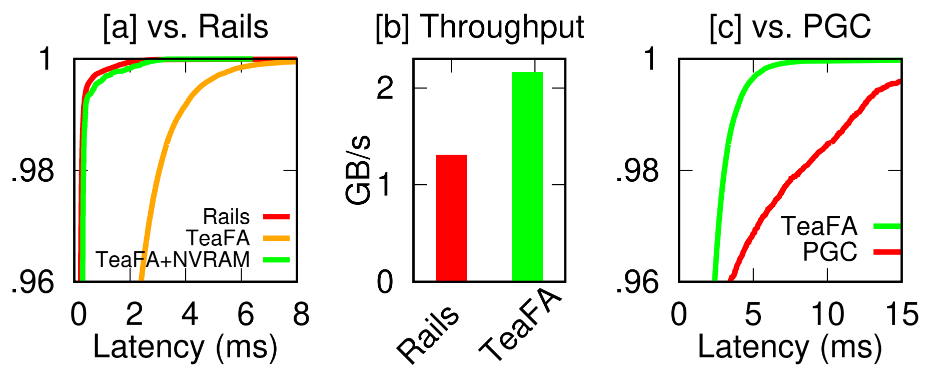


Figure 3.13: **vs. Rails and Preemptive GC (§3.6.4-3.6.4).** *Figure (a) shows the TPCC read CDF latencies on* TEAFA *vs. Rails. Figure (b) shows user-perceived read throughput on Rail vs.* TEAFA. *Figure (c) shows the TPCC read latencies on* TEAFA *vs. Preemptive GC.*

shows that Rails' read throughput is significantly lower compared to TEAFA. In other words, Rails does not leverage the purpose of a flash array in delivering higher bandwidth, while TEAFA behaves the same as a typical RAID. Rails' other downsides include the need for a large external buffer to cache incoming writes and the induced IO contention during the read-write role swapping.

Comparison with Preemptive GC: Preemptive GC (PGC) [220] is an approach that allows user reads to be interleaved in between GC individual read/write/erase operations, hence user reads are not queued far behind. Figure 3.13c shows that the PGC line cuts a huge area of the latency tail as compared to the Base line in Figure 3.9i. But the TeaFA line in Figure 3.13c shows that TEAFA wins over preemptive GC. The reason is straightforward: in TEAFA, users reads do not need to wait behind any GC operations as they will be busy returned and reconstructed. More recent works attempt to preempt in the middle of *a* GC write or erase [205, 232, 309], which can make the PGC line closer, the same or slightly better to the TeaFA line, but require more special hardware support.

Comparison with Other Related Works (Not Evaluated): We unfortunately did not have enough time to re-implement many other tail-cutting works into the same FEMU platform. Thus, we only perform a qualitative comparison below.

Speculative IOs [302, 314, 317] can be integrated into RAID where parity/redundant IOs are sent when the original data IOs have not returned after a certain timeout threshold. For disks, such a method is fitting because the timeout is usually set in the granularity of seconds (*e.g.*, 5*sec*). But

SSD operates at *micro*-second level. Setting a low timeout will essentially make this approach into a pro-active approach (§3.6.4). Setting it to high will lead to a long wait-time.

Partitioning such as FlashBlox [170], OPS isolation [199], and IOD SET [83] methods basically partition the SSDs such that user-vs-GC, user-vs-write, or user-vs-user contention is reduced. While simple and effective, the drawback of partitioning methods is the same as of Rails' (§3.6.4), that the aggregate throughput is sacrificed.

Tiny-Tail Flash [318] is a white-box approach that requires RAIN [16] inside the SSD in order to perform internal read reconstruction, and it also performs a rotating GC. But because TEAFA is a gray-box approach, the challenges we face (interfaces, time window, etc.) are different than those addressed in white-box approaches.

MittOS/MittSSD [164] is an OS approach that predicts how long every IO to the SSD will take. If the predicted latency is higher than the deadline SLO, the OS returns a busy error code such that the application can retry to another less busy node. MittOS and TEAFA are in the same spirit, but the MittSSD part requires "open" SSD devices such as OpenChannel SSDs that expose the internal NAND layout such that the latency estimation can be made accurate, hence also a white-box approach.

Similar to Gecko [286] in spatial separation of device roles, SWAN [201] is an AFA design with two-dimensional data organization. TEAFA differs from them in that it utilizes a small interface change for fast IO responses through temporal coordination of background operations.

3.6.5 Other Evaluations

This last section shows further evaluations.

Dynamic Window Time: Here we evaluate TEAFA's dynamic time window algorithm. Figure 3.14a shows that the dynamic-window approach is very close to the static one. As discussed before, because the dynamic window calculation is host managed, based on the busy signals the host receives from the SSDs, there will be instances where the window grows too large causing

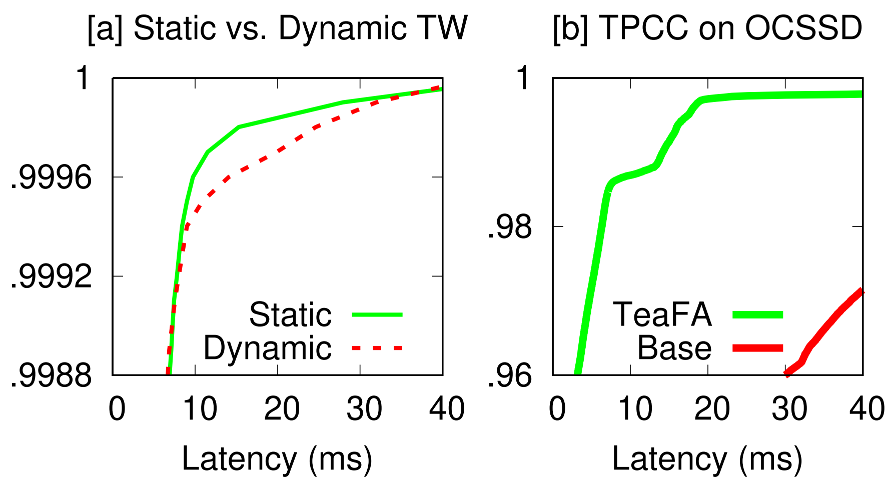


Figure 3.14: **Extended evaluations on Dynamic time window.** *Dynamic Time Windows*

multiple stripe-level sub-IOs to be returned with busy bits enabled. This is why the performance of the dynamic time window is slightly worse than the static one as the figure shows. While in the prior evaluation sections, we have shown the robustness of the static time window across many different workload intensities, in the near future work, we will further explore the robustness of the dynamic time window under various write bursts.

TEAFA on LightNVM+OCSSD: While all prior experiments are based on an emulator (FEMU), we show that TEAFA approach also runs well on real SSD hardware. As discussed before (§3.4), LightNVM+OCSSD [126] is the only platform we could use for this purpose. Our extension in LightNVM represents the TEAFA's firmware changes, and our modified Linux Software RAID remains the same. These two layers run on top of OCSSD. Figure 3.14b compares TEAFA vs. the `Base` approach running TPCC on this real hardware platform. As shown, we obtained a similar improvement as in the FEMU platform in Figure 3.9i.

Window Time and Write Amplification: Finally, to show the implication of window time to write amplification, we ran a further analysis using SSDSim [169]. Figure 3.15 shows the results of this analysis across different workloads. As expected, even a small time window value such as less than 50ms will help reduce write amplification as it performs less frequent GCs (delaying unnecessary GCs until next time window). A large time window is healthy for reducing write

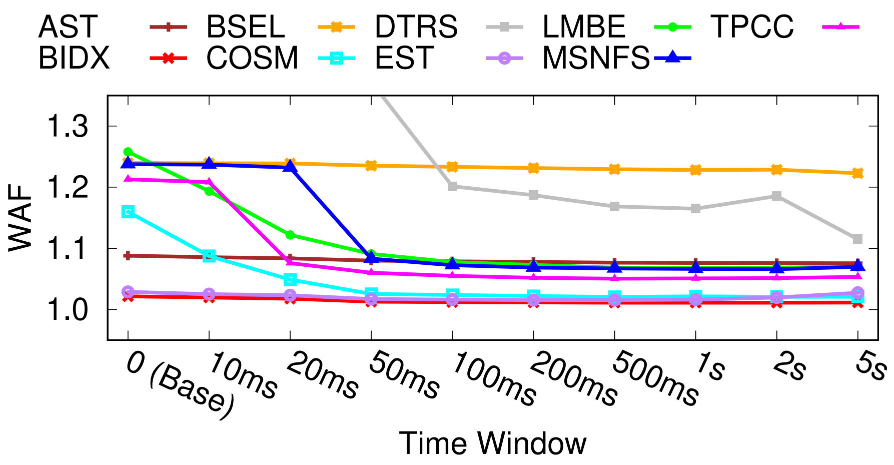


Figure 3.15: **Time window impacts to write amplification (§3.6.5).** *The figure shows the resulting write amplification factor (WAF) in the y-axis when different static time window values are used (in the x-axis) for a variety of workloads.*

amplification, but again it does not allow the cleaning to keep up with the incoming writes, hence will force the GC to break the time window contract, causing multiple busy sub-IOs that creates tail latencies.

3.7 TTFLASH Evaluation

We now present extensive evaluations showing that TTFLASH significantly eliminates GC blocking. We evaluate `ttFlash-Emu` (on VSSIM++), as described in Section 3.4. We use filebench [10] with six personalities as listed in the x-axis of Figure 3.16. `ttFlash-Emu` emulates 48GB SSDs (limited by the machine's DRAM). We use a machine with 2.4GHz 8-core Intel Xeon Processor E5-2630-v3 and 64-GB DRAM. The simulated and emulated SSD drives are pre-warmed up with the same workload.

Figure 3.16 shows the average latencies of filebench-level read operations (including kernel, file-system, and QEMU overheads in addition to device-level latencies) and the percentage of GC-blocked reads measured inside `ttFlash-Emu`. We do not plot latency CDF as filebench only reports average latencies. Overall, `ttFlash-Emu` shows superior performance compared to the baseline.

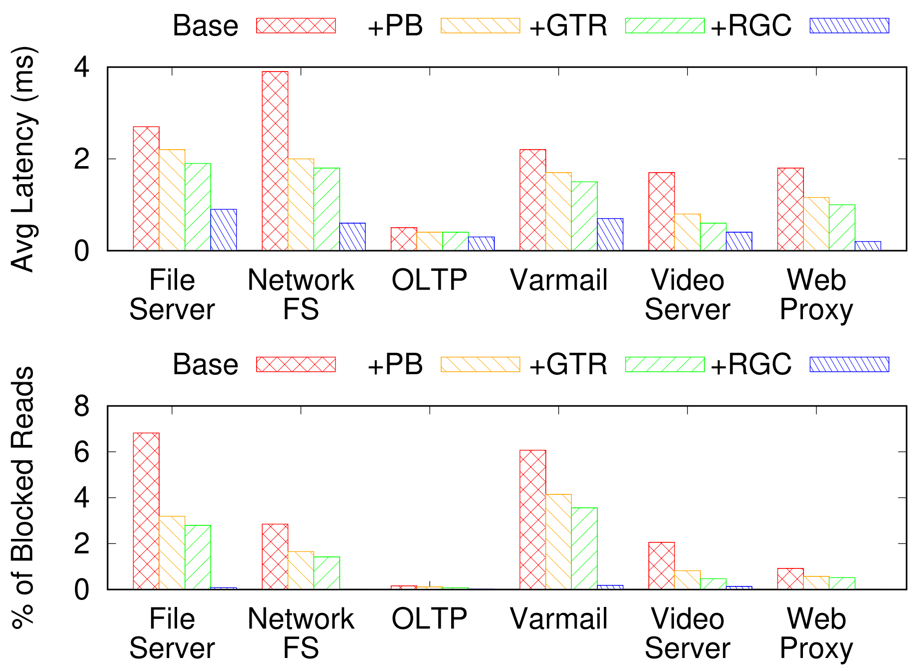


Figure 3.16: **Filebench on `ttFlash-Emu`.** *The top and bottom figures show the average latencies of read operations and the percentage of GC-blocked reads, respectively, across six filebench personalities. "`Base`" represents our VSSIM++ with channel-blocking (§3.4).*

3.8 Summary

Existing application-level tail-tolerant solutions can only guess at resource busyness, as such information is not exposed by the OS. We propose a fundamentally different philosophy: transparency of resource busyness, and have demonstrated the effectiveness of this approach in MITTOS, and documented its benefits for a popular NoSQL application. The MITTOS design pushes latency prediction into the OS layer, where it can deliver good prediction that simplifies applications and eases their tail and other performance management. As cloud systems continue the drive for cost-efficiency, we expect consolidation and sharing to be a fundamental reality. In such a world, busyness transparency, as embodied in the MITTOS principles, should only grow in importance.

The spirit of gray-box approaches are now accepted by the community, becoming official interfaces. We propose a "sweet" design spot to use and extend slightly the IOD interface, and to minimally modify the SSD firmware so not to be intrusive to vendors, but keep most of the tail evading logic in the host. We hope this simple and elegant design will increase the chance of TEAFA adoption. We also believe there are many exciting future research questions such as how to use IOD interface (*e.g.*, program the window time), and hope TEAFA can spur more solutions in this space.

SSD technologies have changed rapidly in the last few years; faster and more powerful flash controllers are cable of executing complex logic; parity-based RAIN has become a standard means of data protection; and capacitor-backed RAM is a de-facto solution to address write inefficiencies. In our work, we leverage a combination of these technologies in a way that has not been done before. This in turn enables us to build novel techniques such as plane-blocking GC, rotating GC, GC-tolerant read and flush, which collectively deliver a robust solution to the critical problem of GC-induced tail latencies.

Chapter 4

LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs

Today's cloud storage stack is extremely resource hungry, burning 10-20% of datacenter x86 cores, a major "storage tax" that cloud providers must pay. Yet, the complex cloud storage stack is not completely offload-ready to today's IO accelerators. While our solutions in Chapter 3 successfully attacked the challenges for predictable performance, they couldn't help with the "storage tax" challenge to make the storage stack run efficiently. In this chapter, we present LeapIO, a new cloud storage stack that leverages ARM-based co-processors to offload complex storage services. LeapIO addresses many deployment challenges, such as hardware fungibility, software portability, virtualizability, composability, and efficiency. It uses a set of OS/software techniques and new hardware properties that provide a uniform address space across the x86 and ARM cores and expose virtual NVMe storage to unmodified guest VMs, at a performance that is competitive with bare-metal servers.

The organization of this chapter is as follows. We first describe LeapIO design in §4.1, by visiting LeapIO from different angles. Then, we discuss the implementation in §4.2 and present evaluation results in §4.3. Lastly, we summarize in §4.4. We leave futher discussion to Chapter 6 (§6.4). The content of this chapter is adapted from our paper *"LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs"* [227] published at ASPLOS'20.

4.1 LeapIO Design

We now present the design of LeapIO from different angles: hardware (§4.1.1), software (§4.1.2), control flow (§4.1.3), data path (§4.1.4), and x86/ARM portability (§4.1.5).

We first clarify several terms: *"ARM"* denotes cheaper, low-power processors suitable enough for storage functions (although their architecture need not be ARM's); *"SoC"* means ARM-based co-processors with ample memory bundled as a PCIe SoC; *"x86"* implies the powerful and expensive host CPUs (although can be non x86); *"rNIC"* stands for RDMA-capable NIC; *"SSD"* means NVMe storage; *"functions"* and *"services"* are used interchangeably.

4.1.1 The Hardware View

We begin with the hardware view.

In the left side of Figure 4.1, the top area is the host side with x86 cores, host DRAM, and IOMMU. In the middle is the PCIe bus connecting peripheral devices. In the bottom right is our SoC card (bold blue edge) containing ARM cores and on-SoC DRAM. Our SoC and rNIC are co-located on a single PCIe card as explained later. The right side in Figure 4.1 shows a real example of our SoC deployment. In terms of hardware installation, the SoC is simply attached to a PCIe slot. However, easy offloading of services to the SoC while maintaining performance requires four hardware capabilities (labels ① to ④ in Figure 4.1), which all can be addressed from the SoC vendor side.

① **HW$_1$: Host DRAM access by SoC**. The SoC must have a DMA engine to the host DRAM (just like rNIC). However, it must allow the user-space LeapIO runtime (running in the ARM SoC) to access the DMA engine to reach the location of all the NVMe queue pairs mapped between the on-x86 user VMs, rNIC, SSD, and in-SoC services (§4.1.3).

It's a prevalent technology for modern PCIe devices to access host DRAM coherently via DMA engines. Moreover, PCIe controllers can also expose DMA engines to user space, thus allowing user-space programs utilizing DMA engines to communicate with host DRAM directly. This is already well supported by today 's programmable hardware with ready-to-use open-source drivers [76, 96].

② **HW$_2$: IOMMU access by SoC**. The trusted in-SoC LeapIO runtime must have access to an IOMMU coherent with the host in order to perform page table walk of the VM's address space

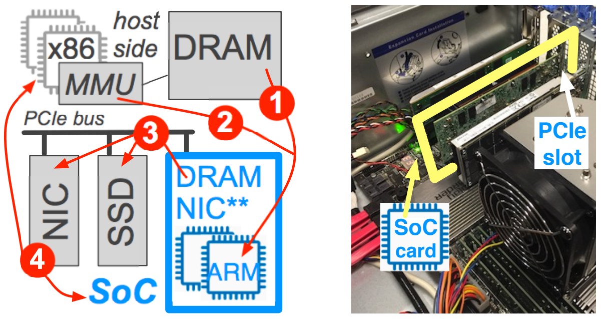


Figure 4.1: **Hardware requirements.** *The figure is explained in Section 4.1.1. "NIC**" means an optional feature of the SoC. x→y means x should be exposed or accessible to y.*

that submitted the IO. When an on-x86 user VM accesses a piece of data, the data resides in the host DRAM, but the VM only submits the data's guest address. Thus, the SoC must facilitate the LeapIO runtime to translate guest to host physical addresses via the IOMMU (§4.1.4).

Modern computer systems utilize IOMMU between host DRAM and PCIe devices for address translations. When PCIe devices try to access host DRAM, it sends PCIe read/write transactions to IOMMU and IOMMU will perform address translation for addresses in PCIe transactions [252]. Modern IO devices, such as NVMe SSDs, RDMA NICs, FPGAs, GPUs are all designed to comply with IOMMU. In LeapIO, we just need the SoC vendor to expose such capability to our runtime at user-space level in SoC OS.

③ **HW$_3$: SoC's DRAM mapped to host**. The on-SoC DRAM must be visible by the rNIC and SSD for zero-copy DMA. For this, the SoC must expose its DRAM space as a PCIe BAR (base address register) to the host x86. The BAR will then be mapped as part of the host physical address space by the host OS. With this capability, main hardware components such as rNIC, SSD, host x86, and the SoC can read/write data via the host address space without routing data back and forth (§4.1.4).

In LeapIO, we require SoC to expose its DRAM space to x86 as a PCIe BAR. We argue that this is also an easy-to-achieve technique. For example, NVMe features such as Controller Memory Buffer (CMB) allows the SSD controller to share its device memory to x86 [54]. There are other devices which expose all of its DRAM space to x86 host as part of its PCIe configuration space [50]. With byte-addressable SSD on the horizon [97, 112], it will be much easier to use device DRAM directly from host. Similar technology can be applied to ARM SoC designs.

④ **HW_4: NIC sharing**. The NIC must be "shareable" between the host x86 and ARM SoC because on-x86 VMs, other host agents, and in-SoC services are all using the NIC. NIC can be used by the host to serve VM traffic as well as by the SoC for offloaded remote storage functions. One possibility is to co-locate the ARM cores and the NIC on the same PCIe card ("NIC**" in Figure 4.1), hence not dependent on the external NIC capabilities (§3.4).

4.1.2 The Software View

Now we move to the software view. To achieve all the goals in §2.6, LeapIO software is relatively complex, thus we decide to explain it by first showing the high-level stages of the IO flows, as depicted in stages ⓐ to ⓕ in Figure 4.2.

ⓐ **User VM**. On the client side, a user runs her own application and guest OS of her choice on a VM where *no* modification is required. For storage, the guest VM runs on the typical NVMe device interface (*e.g.*, `/dev/nvme0n1`) exposed by LeapIO as a queue pair (represented by ●) containing submission and completion queues (SQ and CQ). This NVMe drive which can be a local (ephemeral) drive or something more complex will be explained later.

ⓑ **Host OS**. We add a capability into the host OS for building queue-pair mappings (more in 4.1.3) such that the LeapIO runtime ⓒ sees the same NVMe command queue exposed to the VM. The host OS is *not* part of the datapath.

ⓒ **Ephemeral storage**. If the user VM utilizes local SSDs (*e.g.*, for throughput), the requests will be put into the NVMe queue mapped between the LeapIO runtime and the SSD device (the downward ●—●). Because the SSD is not in the SoC (not inside the bold edge), they need to share the NVMe queue stored in the host DRAM (more in §4.1.3).

ⓓ **Client-side LeapIO runtime and services**. The client-side runtime (shaded area) represents the LeapIO runtime running on the ARM SoC (bold blue edge). This runtime "glues" all the NVMe queue pairs (●—●) and end-to-end storage paths over a network connection (◆—◆). To

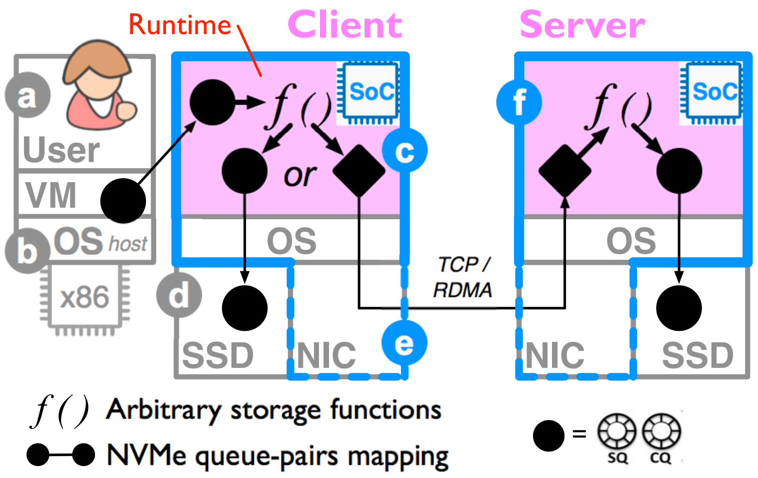


Figure 4.2: **Software view.** *The figure shows the software design of LeapIO (Section 4.1.2). For simplicity, we use two nodes, client and server, running in a datacenter. The arrows in the figure only represent the logical control path, while the data path is covered in §4.1.4. Our runtime and storage services (the shaded/pink area) can transparently run in the SoC (as shown above) or on the host x86 via our "SoC$_{VM}$" support (in §4.1.5).*

quickly process new IOs, the LeapIO runtime polls the VM-side NVMe submission queue that has been mapped to the runtime address space (●—●). This runtime enables services to run arbitrary storage functions in user space ("$f()$", see Table 1.1) that simply operate using NVMe interface. The functions can then either forward the IO to a local NVMe drive (●) and/or a remote server with its own SoC via RDMA or TCP (◆). Later, §4.1.4 will provide details of the data path.

At this stage, we recap the aforementioned benefits of LeapIO. First, the cloud providers can develop and deploy the services in user space (*extensibility*). The LeapIO runtime also does not reside in the OS, hence all data transfers bypass the OS level (both host OS and SoC-side OS are skipped). The SoC-side OS can be any standard OS. Second, with mapped queue pairs, the runtime employs polling to maintain fast latency and high throughput (*efficiency*). Third, VMs can obtain a rich set of block-oriented services via virtual NVMe drives (*virtualizability/composability*). Most importantly, although in the figure, the LeapIO runtime and services are running in the SoC, they are also designed to transparently run in a VM on x86 to support older servers (*portability*), which we name the "SoC$_{VM}$" feature (more in §4.1.5).

ⓔ **Remote access (NIC).** If the user stores data in a remote SSD or service, the client runtime

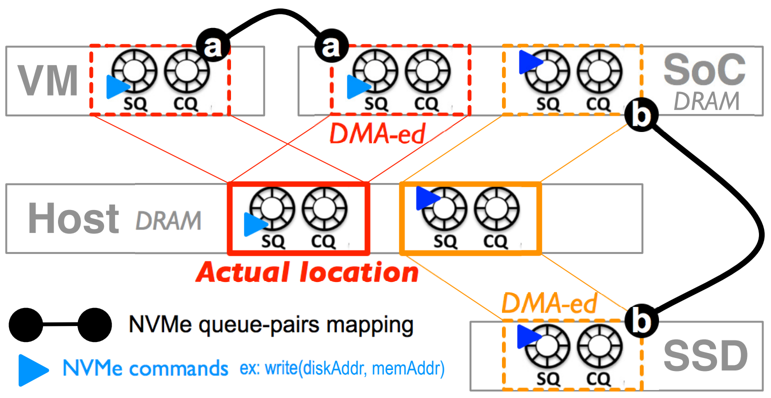


Figure 4.3: **Control setup.** *The figure shows the mappings of NVMe queue pairs to pass IO commands across the hardware and software components in LeapIO, as described in §4.1.3.*

simply forwards the IO requests to the server runtime via TCP or RDMA through the NIC (◆—◆). Note that the NIC is housed in the same PCIe slot (dotted bold edge) as the ARM SoC in order to fulfill property $\mathbf{HW}_4$ (shareable rNIC).

ⓕ **Server-side LeapIO runtime and services**. The server-side LeapIO runtime prepares the incoming command and data by polling the queues connected to the client side (◆). It then invokes the server-side storage functions $f()$ that also run in the user level within the SoC. The server-side service then can forward/transform the IOs to one or more NVMe drives (●) or remote services. The figure shows the access path to its local SSD (the right-most ●—●).

4.1.3 The Control Setup

We now elaborate on how LeapIO provides the *NVMe queue-pair mapping support* to allow different components in LeapIO to use the same NVMe abstraction to communicate with each other. To illustrate this, we use the two logical queue-pair mappings (two ●—●) in the client side of Figure 4.2 and show the physical mappings in Figure 4.3a-b.

ⓐ**VM-runtime queue mapping**. This is the mapping between the user VM and the in-SoC client runtime (red lines). The actual queue-pair location is in the host DRAM (middle row). The user VM (upper left) can access this queue pair via a standard mapping of guest to host address (via hypervisor-managed page table of the VM). For the in-SoC runtime to see the queue pair, the location must be mapped to the SoC's DRAM (upper right), which is achieved via DMA. More specifically, our modified host hypervisor establishes an NVMe *admin* control channel with LeapIO runtime. There is a single admin NVMe queue pair that resides in the host DRAM but it is DMA-ed by the host to the runtime address space, thus requiring property $\mathbf{HW}_1$ (§4.1.1).

ⓑ**Runtime-SSD queue mapping**. This is the mapping between the in-SoC runtime with the SSD (orange lines). Similar to the prior mapping, the hypervisor provides to the SSD the address ranges within the memory mapped region. The SSD does not have to be aware of the SoC's presence. Overall, the memory footprint of a queue pair is small (around 80 KB). Thus, LeapIO can easily support hundreds of virtual NVMe drives for hundreds of VMs in a single machine without any memory space issue for the queues.

With this view (and for clarity), we repeat again the control flow for local SSD write operations, using Figure 4.3. First, a user VM submits an NVMe command such as `write()` to the submission queue (red `SQ` in VM space, ⓐ). Our in-SoC runtime continuously polls this `SQ` in its address space (red `SQ` in SoC's DRAM, ⓐ) and does so by *only* burning an ARM core. The runtime converts the previous NVMe command, submits a new one to the submission queue in the runtime's address space (orange `SQ` in SoC's DRAM, ⓑ), and rings the SSD's "doorbell". The SSD controller reads the NVMe write command that has been DMA-ed to the device address space (orange `SQ` in the SSD, ⓑ). Note that, all of these *bypass* both the host and the SoC OSes.

4.1.4 The Data Path (Address Translation Support)

Now we describe the most challenging goal: *efficient data path*. The problem is that in existing SmartNIC or SmartSSD SoC designs, ARM cores are hidden behind either the NIC controller or storage interface, thus ARM-x86 communication must be routed through NIC/storage control block and not efficient. Furthermore, many software/hardware components are involved in the data path, hence we must *minimize data copying*, which we achieve by building an *address mapping/translation support* using the aforementioned HW properties (4.1.1). Figure 4.4 walks through this most complicated LeapIO functionality (write path only) in the context of a VM accessing a remote SSD over RDMA. Before jumping into the details, we provide high-level descriptions of the figure and the legend.

Components: The figure shows different hardware components and software layers in data and command transfers such as user application, guest VM, host DRAM ("hRAM"), SoC-level device DRAM buffer ("sRAM[1]"), client/server runtime, rNICs, and the back-end SSD.

Command flow: Through these layers, NVMe commands (represented as blue ▶) flow through the NVMe queue-pair abstraction as described before. The end-to-end command flow is shown in non-bold blue line. An example of an NVMe IO command is `write(blkAddr, memAddr)` where `blkAddr` is a block address within the virtual drive exposed to the user and `memAddr` is the address of the data content in the guest VM.

Data location and path: We attempt to minimize data copying (reduced 🟥 count) and allow various software and hardware layers access the data via memory mapping (□). The data is transferred (bold red arrow) between the client and the server, in this context via RDMA-capable NICs.

Address spaces: While there is only one copy of the original data (🟥), different hardware components and software layers need to access the data in their own address spaces, hence the *need for an address translation support*. Specifically, there are four address spaces involved (see the figure legend): **(1)** `guestAddr` *gA* representing the guest VM address, **(2)** `hostAddr` *hA* denoting the host DRAM physical address, **(3)** `logicalAddr` *lA* implying the logical address (SoC user space) used by the LeapIO runtime and services, **(4)** `socAddr` *sA* representing the SoC's DRAM physical address. In our SoC deployment, the SoC and rNIC are co-located ($\textbf{HW}_4$ in §4.1.1), hence `logicalAddr` mode is the most convenient one for using RDMA between the client/server SoCs.

Client-Side Translation: For the client side, we will refer to Figure 4.4ⓐ–ⓓ.

1. **sRAM** denotes on-SoC DRAM, not static RAM.

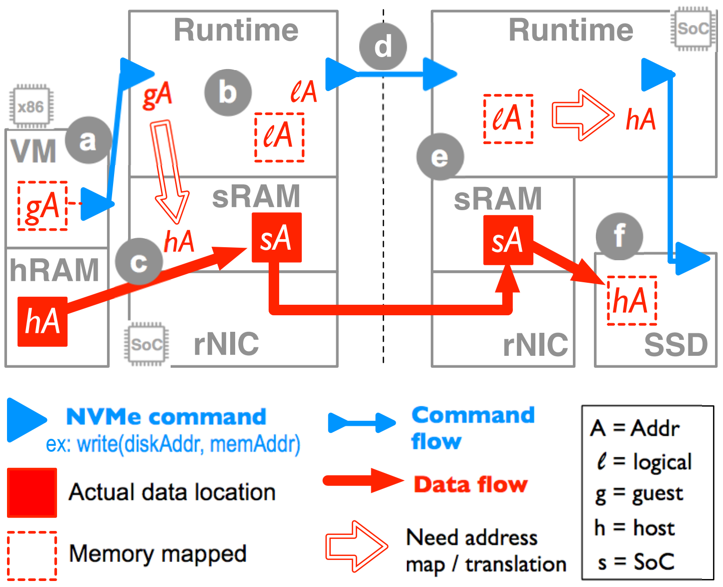


Figure 4.4: **Datapath and address translation.** *The figure shows how we achieve an efficient data path (minimized copy) with our address translation support, as elaborated in §4.1.4. The figure only shows write path via RDMA (read path is similar).*

In step ⓐ, on-x86 guest VM allocates a data block, gets `guestAddr` *gA* and puts a `write` NVMe command ▶ with the `memAddr` pointing to the `guestAddr` *gA*, *i.e.*, `write(blkAddr, gA)`. The data is physically located in the host DRAM (🟥 at `hostAddr` *hA*).

In ⓑ, the LeapIO user-space runtime sees the newly submitted command ▶ and prepares a data block via user-space `malloc()`, hence later it can touch the data ☐ via `logicalAddr` *lA* in the runtime's address space. Because the runtime runs in the SoC, this *lA* is physically mapped to the SoC's DRAM (🟥 at `socAddr` *sA*). Remember that at this point the data at `socAddr` *sA* is still empty.

In step ⓒ, we need to make a host-to-SoC PCIe data transfer (see notes below on efficiency) and here ***the first address translation is needed*** (the first double-edged arrow). That is, to copy the data from the host to SoC's DRAM, we need to translate `guestAddr` *gA* to `hostAddr` *hA* because the runtime only sees "*gA*" in the NVMe command. This `guestAddr-hostAddr`translation is *only available* in the host/hypervisor-managed page tables of the VM that submitted the request. Thus, our trusted runtime must be given access to the host IOMMU (property $\mathbf{HW}_1$ in §4.1.1).

Next, after obtaining the `hostAddr` *`hA`*, our runtime must read the data and copy it to `socAddr` *`sA`* (the first bold red arrow). Thus, the runtime must also have access to the SoC's DMA engine that will DMA the data from the host to SoC's DRAM (hence property $\mathbf{HW}_2$ in §4.1.1).

In ⓓ, at this point, the data is ready to be transferred to the server via RDMA. The client runtime creates a new NVMe command ▶ and supplies the client side's `logicalAddr` *`lA`* as the new `memAddr`, *i.e.*, `write(blkAddr,`*`lA`*`)`. The runtime must also register its `logicalAddr` via the `ibverbs` calls so that the SoC OS (not shown) can tell the rNIC to fetch the data from `socAddr` *`sA`* (the SoC OS has the *`lA`*`-`*`sA`* mapping). This is a standard protocol to RDMA data.

We make several notes before proceeding. First, the host-to-SoC data transfer should *not* be considered as an overhead, but rather a *necessary* copy as the data must traverse the PCIe boundary at least once. This transfer is not done in software, it is performed by enqueueing a single operation to the PCIe controller that does a hardware DMA operation between the two memory regions. Second, LeapIO must be fully trusted to be given host-side page table access, which is acceptable as LeapIO is managed by the cloud provider. A malicious VM's attack surface is restricted to the NVMe queue pairs. Whenever LeapIO detects illeagal NVMe commands, it fails the IOs directly. Overall, LeapIO doesn't expose extra attack surface compared to existing on-x86 hypervisor IO interface.

Server-Side Translation: For the server side, we refer to Figure 4.4ⓔ–ⓖ. LeapIO server keeps monitoring data coming from the network and migrates data to SSD efficiently with direct NVMe access and DMA data transfer between ARM and SSD.

In ⓔ, LeapIO server runtime sees the new command ▶ and prepares a data buffer □ at its `logicalAddr` *`lA`* (a similar process as in step ⓑ). The runtime then makes an RDMA command to fetch the data from the client runtime's `logicalAddr` *`lA`* provided by the incoming NVMe command. The server rNIC then puts the data directly in the SoC's DRAM (■ at `socAddr` *`sA`*). Now LeapIO services can read the data via the `logicalAddr` *`lA`* and run any storage functions $f()$ desired. When it is time to persist the data, the runtime submits a new NVMe command ▶ to the

SSD.

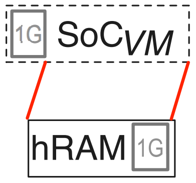
In ⓕ, being outside the SoC, the backend SSD can only DMA data using `hostAddr`(server side) , hence does not recognize `socAddr` *sA* . Thus, the server runtime must submit a new NVMe command that carries "`hostAddr` *hA* " as the `memAddr` of the next write command, *i.e.* `write(blkAddr,`*hA*`)`. This is ***the need for another address translation*** *lA* → *sA* → *hA* (the second double-edged arrow).

For *sA* → *hA* , we need to map the SoC's DRAM space to the aggregate host address space, which can be done with p2p-mem technology (property $\mathbf{HW}_3$ in §4.1.1). With this, the aggregate host address space is the sum of the host and SoC DRAM. As a simplified example, the "`hostAddr` *hA* " that represents the `socAddr` *sA* can be translated from `hA=hostDramSize+sA` (details can vary).

For *lA* → *sA* , the runtime can obtain the `logicalAddr` to `socAddr` translation from the standard `/proc` page map interface in the SoC OS. We use huge page tables and pin the runtime's buffer area so the translation can be set up in the beginning and not slow down the data path.

4.1.5 $\mathbf{SoC}_{VM}$

For fungibility, we design LeapIO to portably run on SoC or x86, such that LeapIO runtime and services are *one code base* that does not fragment the fleet. To support LeapIO to run on x86, we design "SoC_{VM}" (a SoC-like VM) such that our overall design remains the same. Specifically, in Figure 4.3, the "SoC's DRAM" simply becomes the SoC_{VM}'s guest address space. In Figure 4.4, the `socAddr` essentially becomes the SoC_{VM}'s `guestAddr`.



To enable SoC_{VM}'s capability to access the host DRAM, our host hypervisor trusts the SoC_{VM} and performs the memory mapping shown on the right figure. Imagine for simplicity that the SoC_{VM} boots asking for 1GB. The hypervisor allocates a $\boxed{\text{1G}}$ space in the host DRAM, but before finishing, our modified hypervisor extends the SoC_{VM}'s address space by *virtually* adding the entire DRAM size. Thus, any `hostAddr` *hA* can be accessed via SoC_{VM}'s address `1GB+`*hA* (details can vary). To perform the user `guestAddr` *gA* to *hA* translation, we write a host kernel driver that supplies this to the SoC_{VM} via an NVMe-

#lines	Core	SoC_{VM}	Emu
Runtime	8865	+850	+680
QEMU	1388	+385	
Host OS	2340	+560	+360

Table 4.1: **LeapIO complexity (LOC).** *(As described in §3.4)*

like interface to avoid context switches. Finally, to share the guest VM's NVMe queue pairs with SoC_{VM}, we map them into SoC_{VM} as a Base Address Register (BAR) of a virtual PCIe device.

SoC_{VM} also supports legacy storage devices with no NVMe interface. Older generation servers and cheaper server SKUs that rely on SATA based SSDs or HDDs can also be leveraged in LeapIO via the SoC_{VM} implementation (via `libaio`), furthering our fungibility goal. Moreover, SoC_{VM} can *coexist* with the actual SoC such that LeapIO can schedule services on spare x86 cores when the SoC is full.

4.2 Implementation

Table 4.1 breaks down LeapIO 14,388 LOC implementation. The rows represent the software layers we add/modify, including LeapIO runtime, QEMU (v2.9.0), and the host OS/hypervisor (Linux 4.15). In the columns, "Core" represents the required code to run LeapIO on SoC, "SoC_{VM}" represents the support to portably run on x86, and "Emu" means the small emulated part of an ideal SoC (more below).

We develop LeapIO on a custom-designed development board based on the Broadcom StingRay V1 SoC that co-locates an 100Gb Ethernet NIC with 8 Cortex-A72 ARM cores at 3 GHz. Our development board appears to x86 as a smart RDMA Ethernet controller with one physical function dedicated to the on-board SoC (and another for host/VM data), hence the ARM cores can communicate with x86 via RDMA over PCIe (*e.g.*, for setting up the queue pairs).

Of the four HW requirements (§4.1.1), our current SoC, after a 2-year joint hardware development process with Broadcom, can fulfill $\mathbf{HW}_1$, $\mathbf{HW}_3$ (SSD direct DMA from/to SoC DRAM) and

$\mathbf{HW}_4$ (in-SoC NIC shareable to x86) fully and $\mathbf{HW}_2$ with a small caveat. For $\mathbf{HW}_2$ (IOMMU access), we currently satisfy this via huge page translations (fewer address mappings to cache in SoC) facilitated by the hypervisor, which bodes well with the use of huge pages in our cloud configuration. Our software is also conducive to using hardware based virutal NVMe emulators [65, 74] that can directly interact with the IOMMU.

For data-oriented services (*e.g.*, caching and transaction) in LeapIO local virtualization and remote server mode, peer-to-peer DMA (p2p-mem) [54] is used for direct SSD-SoC data transfer to efficiently stage data in SoC DRAM (no x86 involvement). Computation intensive tasks such as compression, encryption can be further offloaded to in-SoC hardware accelerators. Otherwise, we bypass SoC's DRAM (default SSD-host DMA mode) if data path services are not needed.

Despite lack of full $\mathbf{HW}_2$ support, we note that the LeapIO software design and implementation are complete and ready to leverage newer hardware acceleration features such as hardware NVMe emulation features when they are available. Therefore the system performance will improve as hardware evolves while a full software-only (SoC_{VM}) as well as SoC-only implementation allow us to reduce resource/code fragmentation and hardware dependency. To the best of our knowledge, LeapIO is the first comprehensive storage function virtualization stack that uses acceleration opportunistically. It enables cloud providers to expose identical storage services to VMs regardless of server configurations.

4.2.1 NVMe over TCP and REST

Not all machines in the deployment have RDMA NICs and we must envision cases where some deployments only want to add ARM SoC without NIC (*e.g.*, for monetary reasons). Thus, we also support passing NVMe commands (control and data) via TCP. Essentially this is similar to NVMe-over-TCP [85]. This is also for our fungibility principle where NIC can be seen as an option but not a necessity. Similar to TCP, we also provide support for REST APIs [63, 78] as some user VMs interact with external cloud block services.

In our NVMe over TCP implementation, we associate two sockets with each NVMe queue pair over TCP transport, one for command transfer and the other one for data transfer. This design greatly simplifies the overall implementation. Standard NVMeoF TCP implementations, *e.g.*, as in SPDK or Linux kernel, multiplex one socket for both command and data processing, they have to maintain a complex state machine of current command/data transfer status along with extra metadata information due to stateless nature of TCP connections. In LeapIO case, such complex logic is not needed, we use the fact that NVMe commands and completions are of fixed size (64 and 16 bytes, respectively), so we always wait until we `recv()` or `send()` a complete completion/command entry before we go fetching data from the other socket. To improve performance, we utilize non-blocking sockets and polling for immediate command/data processing. NVMe over REST works in a similar manner.

Among the three implemented different communication mechanisms to interact with remote storage devices and services: RDMA, TCP and REST, the performance is as expected – RDMA is the most performant while REST is the slowest (not shown for space). We highlight that any cloud provider can adopt LeapIO with our service that connects to cloud storage (converged or remote) such as Azure Storage or Amazon EBS. This is important for seamless transition to large-scale deployments of LeapIO; as we envision more IO services built in LeapIO, we still want users to reap the benefits of existing full-fledged cloud services.

4.2.2 Polling

LeapIO works in polling mode to avoid hypervisor level guest/host context switches as well as optimizing interrupt-induced software overheads (from physical SSDs). Polling brings the VM NVMe queue pairs closer to SSD physical queues pairs, thus achieving optimal performance. Compared to x86 polling which is power hungry and overkilling performance wise, ARM polling is cheap and can handle requests in timely manner.

To quickly process IO submissions from guest VM, LeapIO polls on the vNVMe submission queue (`SQ`). It hands submission requests to LeapIO runtime, which goes through various layers of services and eventually passed to physical SSD through RDMA/TCP/REST/PCIe.

To rapidly check request completions, LeapIO runtime continuously polls the SSD's completion queue (`CQ` in SoC's DRAM) burning the *same* ARM core above. When the runtime sees a completed request, it puts the notification in the VM's completion queue (`CQ` in VM) and sends an interrupt to the VM (more in §4.2.3). Thus, the NVMe commands *bypass* both the host and the SoC OSes, without involving extra software overhead.

4.2.3 Virtual Interrupt

While the internals of LeapIO are based on polling for speed, LeapIO runtime needs to send virtual interrupts to the user VM to notify guest OS about IO completions (when guest OS uses kernel interrupt-driven storage stack for the virtual NVMe drive).

We can utilize what SR-IOV does and deliver the interrupts from the SoC directly [74]. For this to work, the SoC needs to advertise virtual Message Signaled Interrupt (MSI) capability to user VMs. The host hypervisor enumerates the MSI vectors supported by the SoC and then associates them with the virtual NVMe controller. When LeapIO runtime completes an IO, it issues MSI write to send an interrupt to the VM directly, bypassing host OS/hypervisor, thus introducing minimal performance overhead (no host/guest context switches for injecting interrupts).

This also can be done from the SoC itself without SR-IOV like hardware interrupt support, it requires LeapIO to use software emulated interrupts and inject them to the core-id that the guest is on. If the guest core to physical core mapping changes at runtime then we need to use Inter-processor Interrupt (IPI). A simple driver running at the host level that listens to interrupts from the SoC can do the job. Accounting wise these are basically cycles spent on behalf of the VM so we charge them to the customer, so its not really a first-party overhead. We utilize this method in LeapIO since our SoC doesn't support SR-IOV hardware interrupts to VMs. Optimizing virtual interrupt caused software overhead is out of the scope of this paper.

4.3 Evaluation

We thoroughly evaluate LeapIO with the following questions: §4.3.1: How much overhead does LeapIO runtime impose compared to other IO pass-through/virtualization technologies? §4.3.2: Does LeapIO running on our current ARM SoC deliver a similar performance compared to running on x86? §4.3.3: Can developers easily write and compose storage services/functions in LeapIO?

To mimic a datacenter setup, we use a high-end machine with an 18-core (36 hyperthreads) Intel i9-7980XE CPU running at 2.6GHz with 128G DDR4 DRAM. The SSD is a 2TB data-center Intel P4600 SSD. The user/guest VM is given 8 cores and 8 GB of memory and LeapIO runtime uses 1 core with two loops, one each for polling incoming submission queues, and SSD completion queues.

As we mentioned earlier, modern storage stack is deep and complex. To guide readers in understanding the IO stack setup, we will use the following format: A/B/C/... where A/B implies A using/running on top of B. For clarity, we compare one layer at a time, *e.g.*, A/$\mathbf{B_1}$**-or-**$\mathbf{B_2}$/... when comparing two approaches at layer B. Finally, to easily find our main observations, we label them with obs.

4.3.1 Software Overhead

This section dissects the software overhead of LeapIO runtime. To not mix performance effects from our SoC hardware, we first run LeapIO inside SoC_{VM} (§4.1.5) on x86.

(1) LeapIO vs. PT on Local SSD with FIO/SPDK. We first compare LeapIO with "pass-through" technology (PT) which arguably provides the *most bare-metal* performance a guest VM can reap. With pass-through, guest VM ("gVM") owns the entire local SSD and directly polls the NVMe queue pairs without host OS interference (but PT does not virtualize the SSD like we do). We name this lower stack "gVM/**PT**/SSD" and compare it with our "gVM/**LeapIO**/SSD" stack. Now, we vary what we run on the guest VM.

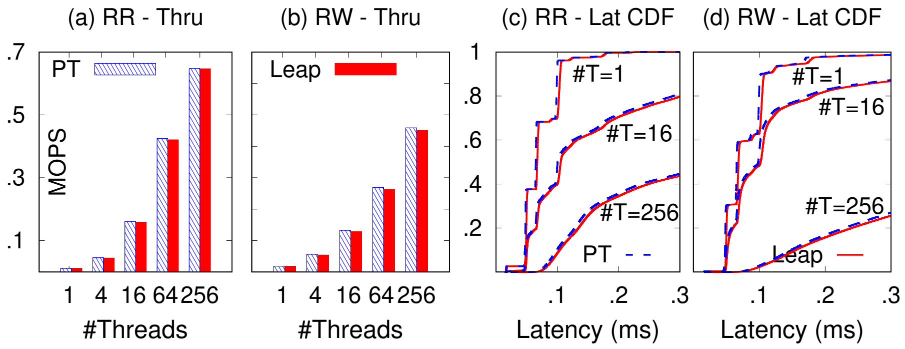


Figure 4.5: **LeapIO vs. Pass-through (PT) with FIO.** *The figure compares LeapIO and* PT *performance as described in §4.3.1(1). From left to right, the figures show read-only (RR) and read-write (RW) throughputs followed with latency CDFs.*

First, we run the FIO benchmark [69] on top of SPDK in the guest VM to not wake up the guest OS (**FIO/SPDK**). This setup gives the highest bare-metal performance as *neither* the guest/host OS is in the data path. We run FIO in two modes (read-only or 50%/50% read-write mix) of 4KB blocks with 1 to 256 threads. To sum up, we are comparing these two stacks: FIO/SPDK/gVM/**PT-or-LeapIO**/SSD.

obs Figure 4.5 shows that we are not far from the bare-metal performance. More specifically, Figure 4.5a-b shows that LeapIO runtime throughput drops only by 2% and 5% for the read-only and read-write throughputs respectively. The write overhead is higher because our datacenter SSD employs a large battery-backed RAM that can buffer write operations in $<5\mu$s. In Figure 4.5c-d, below p99 (the 99^{th} percentile), LeapIO runtime shows only a small overhead (3% on average). At p99.9, our overhead ranges between 6 to 12%. LeapIO runtime is fast because of the direct NVMe queue-pair mapping across different layers. For each 64-byte submission entry, LeapIO runtime only needs to translate 2-3 fields with simple calculations and memory fetches (for translations).

In another experiment (not shown), we convert the 256 threads from 1 guest VM in Figure 4.5a into 8 guest VMs each with 32 threads and obtain the same results. This demonstrates that LeapIO scales well with the number of guest NVMe queue pairs managed.

(2) LeapIO vs. PT on Local SSD with YCSB/RocksDB. Next, we run a real application: RocksDB (v6.0) [88] on ext4 serving YCSB workloads [139] (YCSB/RocksDB/gOS). YCSB is set

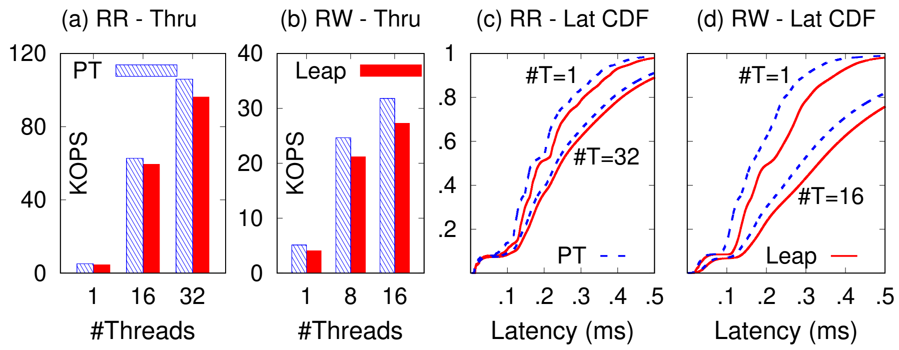


Figure 4.6: **Leap vs. PT (YCSB/RocksDB).** *The figure compares LeapIO and pass-through (*PT*) as described in §4.3.1(2).*

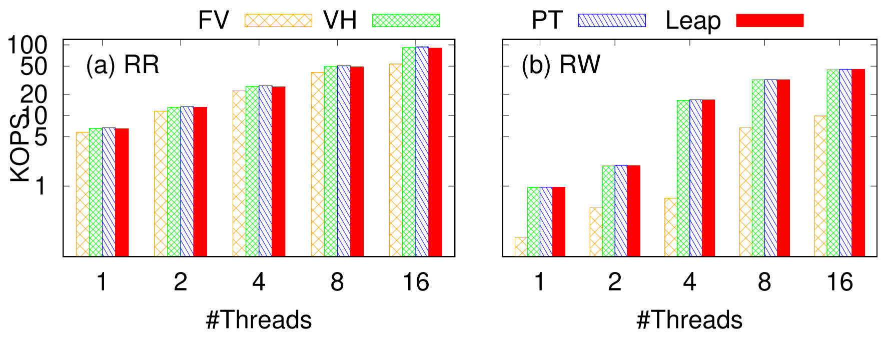


Figure 4.7: **Leap vs. other virt. technologies.** *The figure compares LeapIO with full virtualization (*FV*) and virtual host (*VH*) as described in §4.3.1(3).*

to make uniform request distributions (to measure the worst-case performance) across 100 million key-value entries. We perform read-only or 50-50 read/write workloads. Figure 4.6 confirms the low software overhead of LeapIO by comparing these two stacks: YCSB/RocksDB/gOS/gVM/**PT-or-LeapIO**/SSD. Compared to Figure 4.5c-d, LeapIO latencies are worse than PT mainly due to the software virtual interrupt overhead (VM-exits).

(3) LeapIO vs. Other Technologies on Local SSD. Now we repeat the above experiments but cover other virtualization technologies. To make a faster RocksDB setup that bypasses the guest OS, we run RocksDB on SPDK and run `db_bench` benchmark (db_bench/RocksDB/SPDK/gVM). We switch to `db_bench` as YCSB workloads require the full POSIX API that is currently not supported by SPDK.

We now vary the technologies under the guest VM (gVM/**FV-or-VH-or-PT-or-LeapIO**/SSD). Full virtualization ("**FV**") [70] provides SSD virtualization but is the slowest among all as it must

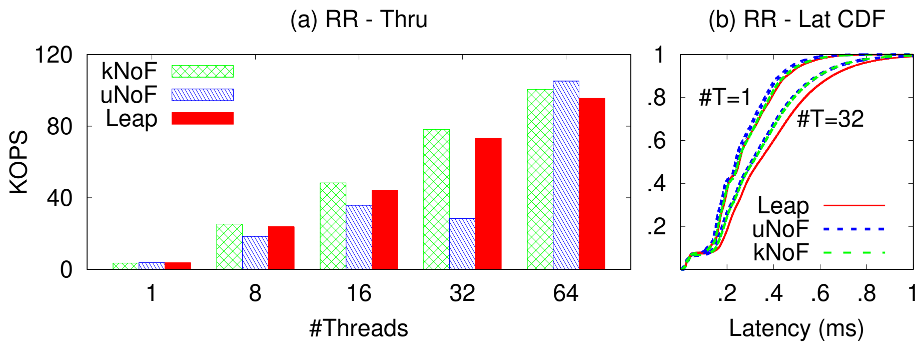


Figure 4.8: **LeapIO vs. kernel/user NVMeoF.** *We compare LeapIO remote NVMe feature with kernel and user NVMeoF (*KNOF *and* UNOF*).* UNOF *is unstable; with 32 threads, at p99.9* UNOF *reaches 14ms while LeapIO can deliver 1.4ms, and at p99.99* UNOF *reaches almost 2000ms while LeapIO is still around 7ms.*

wake up the host OS (via interrupts) to reroute all the virtualized IOs. Virtual host (“**VH**”) [59] is a popular approach [48] that combines virtualization and polling but requires guest OS changes (*e.g.*, using the `virtio` interface and SPDK-like polling to get rid of interrupts.)

obs Figure 4.7 shows the results. While LeapIO loses by 3% to PT, when compared to popular IO virtualization technologies such as virtual-host, LeapIO throughput degradation is only 1.6%. At p99.99 latency LeapIO is only slower by 26μs (1%). This is an acceptable overhead considering that now we can easily move IO services to ARM co-processors.

(4) LeapIO vs. NVMeoF for Remote SSD access. We compare LeapIO server-side runtime with a popular remote IO virtualization technology, NVMeoF, which is a standard way for disaggregating NVMe storage access over RDMA/TCP [38]. Once connecting the NVMeoF client, the server continuously minotors and routes incoming NVMe commands to the backend SSDs. There are two server-side NVMeoF options we evaluate: *kernel-based* one that works in an interrupt-driven mode (“**KNOF**”) and *user-space* one that utilizes SPDK for polling (“**UNOF**”). We use the YCSB/RocksDB client setup as before, but now with remote SSD. Thus, we compare YCSB/RocksDB/gOS/gVM/*client*/–RDMA–/**KNOF-or-UNOF-or-LeapIOServer**/SSD, where *“client”* implies the client-side runtime of either KNOF, UNOF, or LeapIO (TCP setup omitted due to space limit).

obs Based on Figure 4.8, we make two observations here. First kernel-based NVMeoF (KNOF)

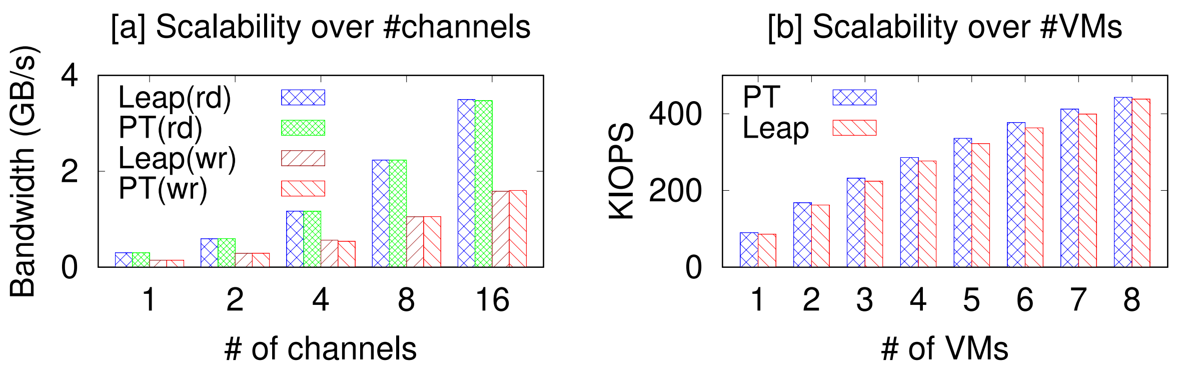


Figure 4.9: **Scalability experiment results.** *The figure is explained in Section §3.5.*

is most stable and performant, but is not easily extensible as services must be built in the kernel. However, our more extensible LeapIO only imposes a small overhead (6% throughout loss and 8% latency overhead). Second, interestingly we found that user-space NVMeoF (UNOF) is *unstable*. In majority of the cases, it is worse than LeapIO but in one case (64 threads) UNOF is better (after repeated experiments). UNOF combined with RDMA is a relatively new and some performance and reliability issues have been recently reported [93, 94, 95]. We also tried running UNOF over TCP to no avail (not shown for space). With this, we can claim that LeapIO is *the first* user-space NVMeoF platform that delivers stable performance for the VMs.

(5) LeapIO Scalability. Figure 4.9 shows LeapIO scalability under multiple SSD channels and multiple VMs. In Figure 4.9(a), we use one user VM and assign different number of channels from an OpenChannel-SSD to it, it shows that the VM is able to achieve bare-metal performance under different channel setups. In Figure 4.9(b), we run multiple VMs on top of LeapIO, in each VM, we run a single thread IO benchmark, by increasing the number of VMs, LeapIO is able to maintain a linear growth in IOPS with near bare-metal performance.

4.3.2 SoC Performance

We now dissect separately the performance of LeapIO client- and server-side runtimes on an ARM SoC vs. on x86.

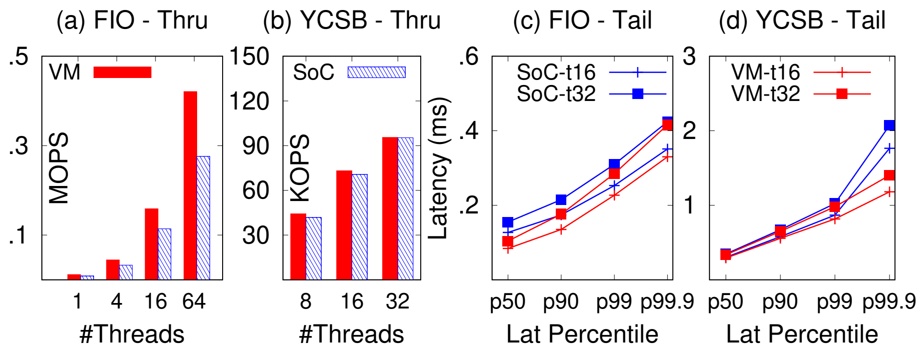


Figure 4.10: **SoC vs. SoC$_{VM}$ Benchmarks.** *The figure compares the performance of LeapIO SSDs (local and remote) running on a SoC vs. in a SoC$_{VM}$ as described in §4.3.2.*

(1) Local SSD (realSoC vs. SoC$_{VM}$). We reuse the FIO-on-local-SSD stack in §4.3.1.1 for this experiment (specifically FIO/SPDK/gVM/**SoC$_{VM}$-or-realSoC**/SSD). Figures 4.10a&c show realSoC runtime is up to 30% slower than SoC$_{VM}$. This is because in our current implementation, we access the guest VMs' and SoC-side queue pairs via SoC-to-host one-sided RDMA (§3.4), which adds an expensive 5μs per operation. We are working with Broadcom to revamp the interface between SoC and the host memory to get closer to native PCIe latencies. Another reason is that the ARM cores run at a 25% lower frequency compared to the x86 cores.

(2) Remote SSD (realSoC vs. SoC$_{VM}$). Next, to measure remote SSD performance, we repeat the setup in §4.3.1.4 (YCSB/RocksDB/gOS/gVM/**SoC$_{VM}$**/–RDMA–/**SoC$_{VM}$-or-realSoC**/ SSD). Figures 4.10b&d show that realSoC on remote side (and SoC$_{VM}$ on client side) has a *minimal overhead* compared to the previous setting (only 5% throughput reduction and 10% latency overhead at p99) because the remote runtime does not need to communicate with the remote host, hence does not suffer from any overheads. However, we note that the overheads would be similar to the previous experiment when both sides use realSoC.

$\boxed{\text{obs}}$ Overall, although current realSoC-LocalSSD is up to 30% slower (will be improved in our future SoC), our cost benefit analysis shows that using even 4× more cores in realSoC compared to the number of cores in SoC$_{VM}$ to achieve performance parity still pays off. From the second experiment, we show that x86 is an overkill for polling and ARM co-processors can easily take over the burden when serving SSDs over the network.

4.3.3 Composability

LeapIO runtime enables composing services in easy ways. Like filtering operations in networking, LeapIO services get a command, process it, and then either send a completion back to the upstream queue or forward sub-commands to many downstream queues (*e.g.*, striping). Composability is achieved by chaining and striping filters. For instance, one simple example we demonstrate later is combining priority and snapshot services.

We build various storage services that compose local/remote devices as well as remote services in just 70 to 4400 LOC each, all in *user space* on LeapIO runtime. In all the experiments below, we use a "search-engine" workload trace containing read-only, user-facing index lookups. We take 1 million IOs, containing various IO sizes from 4K to 7M bytes with average and median size of 36K and 32K bytes respectively. We also co-locate the search workload with a background ("**BG**") workload that performs intensive read/write IOs such as rebuilding the index. The purpose of using a real search-engine trace is as a case study of migrating latency-sensitive services from dedicated servers to a shared cloud, thereby making latency-sensitive services more elastic with resources in proportion to the load.

(a) Prioritization service. A crucial enabler for high cloud utilization is the ability to prioritize time-sensitive, user-facing queries over non-interactive background workloads such as index rebuilding. For this, we build a new service that prioritizes interactive workloads while keeping the batch processing workload make meaningful forward progress when hosts are under-utilized.

The "`Base`" line in Figure 4.11a shows the latency CDF of the search-engine VM without contention. However, when co-located with batch workloads (BG), the search VM suffers long latencies ("`+BG`" line). With our prioritization service, the search VM observes the same performance as if there were no contention ("`+BG+Prio`" line). At the same time, the batch workload obtains 10% of the average resources to make meaningful progress (not shown).

(b) Snapshot/version service. Another important requirement of search engines is to keep the index fresh. The index-refresher job must help foreground jobs serve queries with the freshest

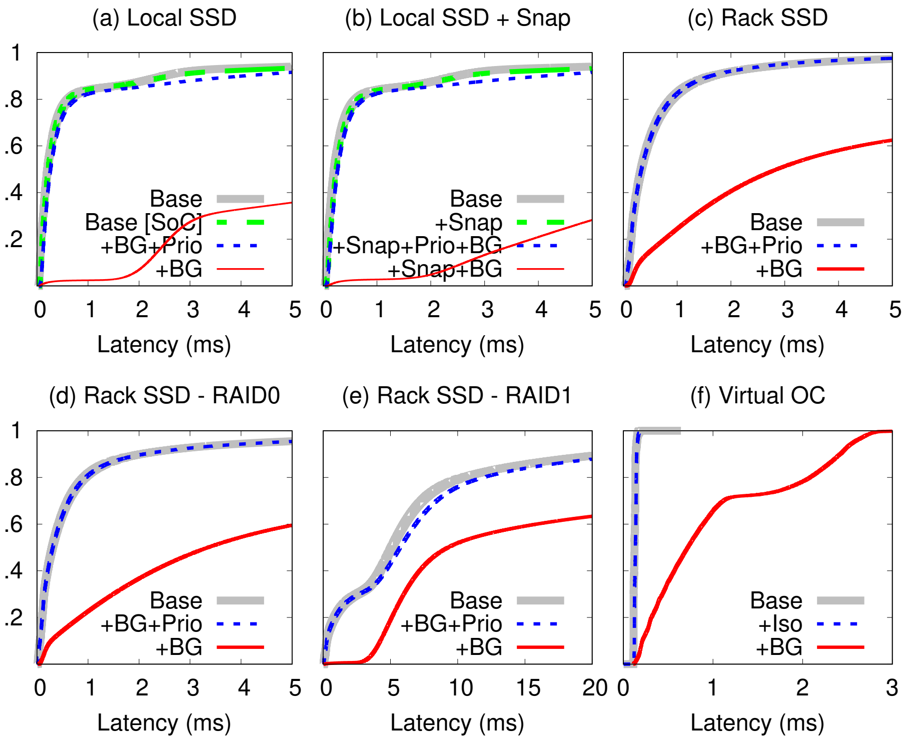


Figure 4.11: **Service features.** *The figures are described in §4.3.3(a)-(f). The experiments are done on* SoC_{VM} *for faster evaluation. In (a), we run one experiment on SoC "[SoC]" to show a similar pattern.*

index. It is undesirable to refresh the index in an offline manner where the old index is entirely rejected and a new one is loaded (causing invalidated caches and long tail latencies). A more favorable way is to update the index gradually.

For this, we build two new services. The first service implements a snapshot feature (the index-refresher job). Here, all writes are first committed to a log using our NVMe multi-block atomic write command while a background thread gradually checkpoints them (while the foreground thread serves consistent versions of the index).

We use a persistent linked list and hashtable to implement an ordered set of writes (ordered by version number). Writers transactionally add versions to this ordered set. Once the publisher is ready to advance, a background thread locks and writes the oldest version to the SSD and transactionally deletes it from the log (this is a idempotent step to provide crash consistency).

The second service is a search VM that looks up the log and obtains blocks of the version they need if they are present in the log and reads the remaining data from the SSD.

Figure 4.11b shows that this snapshot-consistent read feature adds a slightly longer latency to the base non-versioned reads (“`+Snap`” vs “`Base`” lines). When combined with the background writer, the snapshot-consistent reads exhibit long latencies (“`+Snap+BG`” line). Here we can easily compose our snapshot-consistent and prioritization features in LeapIO (the “`+Snap+Prio+BG`” line).

(c) Remote rack-local SSD. Decoupling compute and storage is a long standing feature of many storage services. We want to decouple the search service also from its storage. Figure 4.11c shows the results the same workload used in Figure 4.11b (prioritization) but now the storage is a remote SSD (in the local rack shared by multiple search VMs). The experiment shows that the prioritization mechanism works end-to-end even with the network now in the data path.

(d) Agile rack-local RAID. Our servers that power search engines require disproportionately larger and more powerful SSDs compared to traditional VM workloads. Currently, this means we must overprovision SSD space and bandwidth to keep the fleet uniform and fungible. With LeapIO, we propose *not* to overprovision the dedicated SSDs but rather build larger composable virtual rack-local SSDs.

More specifically, in every rack, each server publishes the free SSD IOPS, space and available network bandwidth that it can spare to a central known billboard every few minutes. Any server that needs to create a virtual drive beyond the capacity of its free space consults the billboard. It then sequentially contacts each server directly to find out if it still has the necessary free capacity until one of them responds affirmatively. In such a case, they execute a peer-to-peer transaction with a producer/consumer relationship and establish a direct data path between the two. The consumer uses the additional space to augment its local SSDs to support the search VMs in the rack which are interested in this partition of the index.

Figure 4.11d shows the same experiments in Figure 4.11c but now the backend drive is a RAID-0 of two virtual SSDs (more is possible) in two machines, delivering a higher performance and capacity for the workload.

(e) Rack-local RAID 1. To protect against storage failures, one can easily extend RAID 0 to RAID 1 (or other RAID protection levels). Figure 4.11e shows the results with RAID-1 of two remote SSDs as the backend. Note that we lose the performance of RAID-0 but now get reliability.

(f) Virtualized Open Channel service for isolation. Another related approach to prioritization is isolation – different VMs use isolated virtual drives within the same SSD. For this, we compose a *unique stack* enabled by LeapIO: gVM/LeapIOClient/OC (where "**OC**" denotes OpenChannel SSD [126, 228]). OC can be configured to isolate flash channels for different tenants [170]. Unfortunately, OC *cannot* be virtualized across multiple VMs, because LightNVM must run in the host OS and directly talks to OC.

With LeapIO, we can *virtualize OC*. Guest VM/OSes can run LightNVM not knowing that underneath it LeapIO remaps the channels. As an example, a guest VM can ask for 4 channels and our new OC service can map the requested channels to the local (or even remote) OC drives. Hence, our new OC service is capable of *exposing virtual channels* and allow guest VMs to reap OC performance isolation benefits. In Figure 4.11f, when a VM (running basic FIO) competes with another write-heavy VM on a shared SSD, the FIO latencies are heavily affected ("`Base`" vs. "`+BG`" lines). However, after we dedicate different channels for these two VMs, the search engine performance is now isolated ("`+Iso`" line).

4.4 Summary

LeapIO is our next-generation cloud storage stack that leverages ARM SoC to alleviate taxing x86 CPUs. Our experience and experiments with LeapIO show that the engineering and performance overhead of moving from x86 to ARM is minimal. In the shorter term, we will continue to move existing host storage services to LeapIO, while our longer term goal is to develop new capabilities that allow even the guest software stack to be offloaded to ARM.

Chapter 5

FEMU: Accurate, Scalable and Extensible NAND-Flash based NVMe SSD Emulator

In this chapter, we introduce FEMU, a storage research platform to foster future full-stack storage research. First, we review the state-of-the-art research platforms that are available today for researchers (§5.1, §5.2). Then, we describe FEMU designs in achiving scalability (§5.3) and accuracy (§5.4). We present FEMU evaluation results in §5.5 and describe its usability and extensibility in §5.6 and §5.7, respectively. Lastly, we conclude in §5.8. The chapter is based on our paper "The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator" (FAST'18) [228].

5.1 SSD Research Platform Introduction

Cheap and extensible research platforms are a key ingredient in fostering wide-spread Solid-State Drives (SSDs) research. Existing SSD research platforms can be categorized into three types: simulator, emulator and hardware based platforms, as shown in table 5.1. Since SSDs' first debut as server storage more than a decade ago, we have seen numerous SSD architecture designs and FTL algorithms innovations. For example, software defined flash (SDF) which offloads NAND management tasks to the host, FTL algorithm innovations like wear leveling mechanisms for increasing SSD lifetime, better IO scheduling / Garbage Collection (GC) algorithms to improve overall I/O performance, etc. While simulation is a quick and easy way to evaluation new designs, it experiences several drawbacks we discuss below.

SSD simulators such as DiskSim's SSD model [101], FlashSim [162] and SSDSim [169], despite their popularity, only support internal-SSD research but not kernel-level extensions. That means, users can only run workload traces to verify new designs. Even today, SSD researchers are still depending on IO workload traces which were published a decade ago retrieved from disk-

Platform	Pros	Cons
Simulator	Cheap; Easy; Time-saving	Trace-driven; Internal research only
Emulator	Cheap; Full-stack research support	Poor scalability; Poor accuracy
Hardware	Full-stack research support; Accurate	Expensive; Complex; Wear-out

Table 5.1: **SSD Research Platforms Comparison.** *SSD Research Platform Pros & Cons: Simulator v.s. Emulator v.s. Hardware platforms*

based systems. This imposes an embarrasing situation here as no new workload traces are open sourced and researchers have to rely on those decade-old disk traces. On the other hand, hardware research platforms such as FPGA boards [257, 279, 327], OpenSSD [46], or OpenChannel SSD [126], support full-stack software/hardware research but their high costs (thousands of dollars per device) impair large-scale SSD research. Also, hardware based platforms are complex to use since the development environment is quite different from application development in the host. It requires very low level knowledge about the System-on-Chip (SoC) to make viable modifications. This would greatly lengthen the project development cycle. Wear-out issues would also jump in when the device is not programmed carefully. As a return, we get the most accurate results since everything is "real".

This leaves software-based emulator such as QEMU-based VSSIM [325], FlashEm [330], and LightNVM's QEMU [45], as the cheap alternative platform. Emulators have the potential to achieve benefits of both the hardware platforms in full system stack support and simulators in easiness to use. Essentially, emulators are software, they simulate the hardware logic and expose a "fake" device (emulated) to guest OS, thus enabling research at different levels. This also guarantees that it's flexible to use as users are free to propose device level innovations and experiment it using application level benchmarks (real workloads).

Unfortunately, the state of existing emulators is bleak; they are either outdated, non-scalable, or not open-sourced. Of the popular SSD Emulators we are aware of, VSSIM design is based on IDE interface, which exposes performance constraints over utilizing high parallelism exposed by today's SSDs. Its simple whole-GC design where GC will lock down the whole device cannot represent today's fine-granular GC algorithms where user and GC request can co-exist. FlashEmu is no longer maintained and LightNVM's QEMU is only designed for OS level FTL development. It's not a performance platform and only supports single channel configuration. The poor design of LightNVM's QEMU plus the virtualization overhead prevent it from being able to emulate hundreds of microsecond level latencies in state-of-the-art NAND Flash.

We argue that it is a critical time for storage research community to have a new software-based emulator (more in §5.2). To this end, we present FEMU, a QEMU-based flash emulator, with the following four "CASE" benefits.

First, FEMU is **c**heap ($0) as it is an open-sourced software. FEMU has been successfully used in several projects, some of which appeared in top-tier OS and storage conferences [164, 318]. We hope FEMU will be useful to broader communities and accelerate research in broader areas.

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 [126] 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 **e**xtensible. 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.

In the following sections, we first present an extended motivation (§5.2).

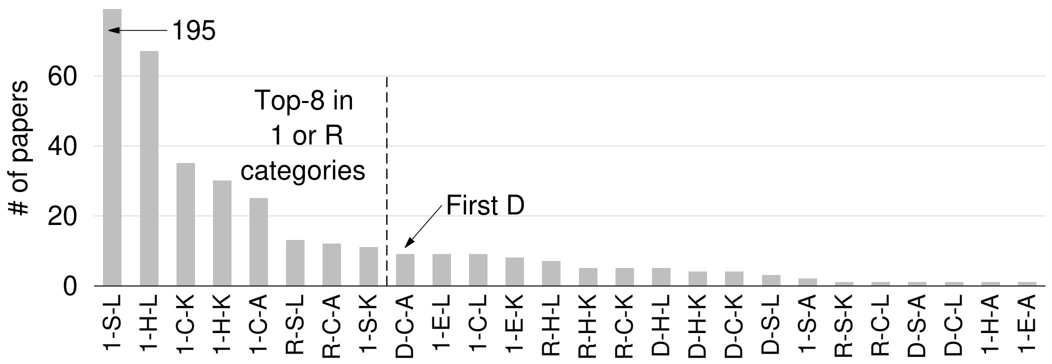


Figure 5.1: **Categorization of SSD research.** *The figure is explained in Section §5.2. The first bar reaches 195 papers.*

5.2 Extended Motivation

5.2.1 The State of SSD Research Platforms:

We reviewed 391 papers in more than 30 major systems and storage conferences and journals[1] published in the last 10 years, and categorized them as follows:

1. What was the **scale** of the research? **[1]:** single SSD; **[R]:** RAID of SSDs (flash array); or **[D]:** distributed/multi-node SSDs.
2. What was the **platform** being used? **[C]:** commodity SSDs; **[E]:** software SSD emulators (VSSIM [325] or FlashEm [330]); **[H]:** hardware platforms (FPGA boards, OpenSSD [46], or OpenChannel SSD [45]); or **[S]:** trace-based simulators (DiskSim+SSD [101] or FlashSim [162] and SSDSim [169]).
3. What **layer** was modified? **[A]:** application layer; **[K]:** OS kernel; **[L]:** low-level SSD controller logic.

Note that some papers can fall into two sub-categories (*e.g.*, modify both the kernel and the SSD logic). Figure 5.1 shows the sorted order of the combined categories. For example, the most popular category is **1-S-L**, where 195 papers target only single SSD (**1**), use simulator (**S**),

1. ASPLOS, EuroSys, FAST, MSST, OSDI, SOSP, SYSTOR, TECS, TPDS, TOC, TOS, etc..

and modify the low-level SSD controller logic (**L**). However, simulators do not support running applications and operating systems.

5.2.2 The Lack of Large-Scale SSD Research

Our first motivation is the lack of papers in the distributed SSDs category (**D-**...), for example, for investigating the impact of SSD-related changes to distributed computing and graph frameworks. One plausible reason is the cost of managing hardware (procurement, installation, maintenance, etc.). is high, especially in large deployments, required for distributed storage experiments. The top-8 categories in Figure 5.1, a total of 324 papers (83%), target single SSD (**1-**...) and flash array (**R-**...). The highest **D** category is **D-C-A** (as highlighted in the figure), where only 9 papers use commodity SSDs (**C**) and modify the application layer (**A**). The next **D** category is **D-H-L**, where hardware platforms (**H**) are used for modifying the SSD controller logic (**L**). Unfortunately, most of the 6 papers in this category are from large companies with large research budget (*e.g.*, FPGA usage in Baidu [257] and Tencent [327]). Other hardware platforms such as OpenSSD [46] and OpenChannel SSD [45] also cost thousands of dollars each, impairing multi-node non-simulation research, especially in academia.

5.2.3 The Rise of Software-Defined Flash

Today, research on host-managed (aka. “software-defined” or “user-programmable”) flash is growing [222, 257, 279, 284, 305, 327]. The idea is to have, not only the inflexible SSD firmware, but also the software (*e.g.*, the host OS or application) manage the flash devices, However, such research is mostly done on top of expensive and hard-to-program FPGA platforms. Recently, a more affordable and simpler platform is available, OpenChannel SSD [45], managed by Linux-based LightNVM [126]. The SSD exposes all the internal physical page addresses (channels, chips, blocks, and pages) to the host. Before its inception (2015), there were only 24 papers that performed kernel-only changes, since then, 11 papers have been published, showing the success of

OpenChannel SSD as a viable hardware platform for software-defined flash research.

However, there remains several issues. First, not all academic communities have budget to purchase such devices. Even if they do, while prototyping the kernel/application, it is preferable not to write too much to and wear out the device. Thus, replacing OpenChannel SSD (during kernel prototyping) with a software-based emulator is desirable.

5.2.4 The Rise of Split-Level Architecture

While most existing research modify a single layer (application/kernel/SSD), some recent works show the benefits of "split-level" architecture [47, 182, 211, 293, 307], wherein some functionalities move up to the OS kernel (**K**) and some other move down to the SSD firmware (**L**) [175, 274, 285]. For example, page-level synchronization for preventing read-write data race should move up because testing data-race free firmware is long and expensive[2]; garbage-collection (GC) management should move up as it must be tied to user SLAs [199]; and some features such as atomic writes, persistent trim, and deduplication should move down [285]. While these are just a few examples, there is a vast research space to explore. So far, we found only 40 papers in split-level **K**+**L** category (*i.e.*, modify *both* the kernel and SSD logic layers), mostly done by companies with access to SSD controllers [182] or academic researchers with Linux+OpenSSD [189, 275] or with block-level emulators (*e.g.*, Linux+FlashEm) [267, 330]. OpenSSD with its single-threaded, single-CPU, whole-blocking GC architecture also has many known major limitations [318]. FlashEm also has limitations as we elaborate more below. Note that the kernel-level LightNVM is not a suitable platform for split-level research (*i.e.*, support **K**, but not **L**). This is because its SSD layer (*i.e.*, OpenChannel SSD Controller & its firmware) is not modifiable; the white-box part of OpenChannel SSD is the exposure of its internal channels and chips to be managed by software (Linux LightNVM), but the OpenChannel firmware logic itself is a black-box part.

2. Per our conversations with SSD engineers

5.2.5 The State of Existing Emulators

We are only aware of three "popular" software-based emulators: FlashEm, LightNVM's QEMU and VSSIM.

FlashEm [330] is an emulator built in the Linux block level layer, hence less portable; it is rigidly tied to its Linux version; to make changes, one must modify Linux kernel. FlashEm is not open-sourced and its development stopped two years ago (confirmed by the creators).

LightNVM's QEMU platform [43] is still in its early stage. Currently, it cannot emulate multiple channels (as in OpenChannel SSD) and is only used for basic testing of 1 target (1 chip behind 1 channel). Worse, due to the virtualization overhead and excessive use of heavy syscalls in virtual I/O path, LightNVM's QEMU performance is not scalable to emulate NAND latencies as it depends on vanilla QEMU NVMe interface (as shown in the NVMe line in Figure 5.2a). Later, we show that FEMU can be used as a drop-in replacement of OpenChannel SSD with relative accuracy, for prototyping purposes.

VSSIM [325] is a QEMU/KVM-based platform that emulates NAND flash latencies on a RAM disk, and has been used in several papers. The major drawback of VSSIM is that it is built within QEMU's IDE interface implementation, which is not scalable. The upper-left red line (IDE line) in Figure 5.2a shows the user-perceived IO read latency through VSSIM without any NAND-delay emulation added. More concurrent IO threads (x-axis) easily multiply the average IO latency (y-axis). For example from 1 to 4 IO threads, the average latency spikes up from 152 to 583μs. The root cause is that IDE is not supported with virtualization optimizations.

When we add just a 50μs delay emulation in VSSIM (*i.e.*, as if a page read takes 50μs), the resulting average latency multiplies further, up to 5ms with 16 IO threads (although the IOs are directed to different channels/chips). This is because VSSIM delay emulation is built within the single-threaded IDE entry path. Thus busy loop creates queueing delays. In our setup, VSSIM's maximum user throughput is only 10 KIOPS and drops to 1 KIOPS with an ongoing GC.

With this drawback, emulating internal SSD parallelism is a challenge. VSSIM worked around the problem by only emulating NAND delays in another background thread in QEMU, disconnected from the main IO path. Thus, for multi-threaded applications, to collect accurate results, users solely depend on VSSIM's monitoring tool [325, Figure 3], which monitors the IO latencies emulated in the background thread. In other words, users cannot simply time the multi-threaded applications (due to IDE poor scalability) at the user level. When multiple applications run concurrently, VSSIM monitor only reports the overall IO latency but cannot distinguish the timing of each application.

Despite these limitations, we (and the community) are *greatly indebted* to VSSIM authors as VSSIM provides a base design for future QEMU-based SSD emulators. As five years have passed, it is time to build a new emulator to keep up with the technology trends.

5.3 FEMU Scalability

By scalability, we mean the ablility of the target system to handle concurrent I/Os within a certain latency threshold without causing unintended queueing dealys. Scalability is an important property of a flash emulator, especially with high internal parallelism of modern SSDs.

Modern SSDs are equipped with hundreds of indepedent NAND flash chips across dozens of channels. Each NAND flash chip is an independent unit which can execute a NAND command (e.g. Page Read, Page Program or Block Erase, etc.) at a time. This implies that modern SSDs can process hundreds of inflight I/Os simutaneously without causing queueing delays. Exploiting this massive amount of hardware levle parallelism has been a key research area for storage researchers. However, as we will show later, stock QEMU/KVM is far from being able to provide similar parallelisms with concurrent I/Os even under the ideal case where there is no delay emulation at all. Thus, to architect an accurate emulator platform which can deliver similar parallelisms and latency timings as real hardware platforms do, we need to solve the scalability bottleneck of current QEMU/KVM implementation. In this section, we first show experimental results demonstrating QEMU/KVM is not scalable to deliver stable performance for low latency and parallelism emulation, then we focus on explaining where such scalabilty bottleneck comes from and how FEMU takles this challenge.

5.3.1 QEMU I/O Scalabilty

QEMU supports various types of storage interface emulations, which can be categorized into three types. The first one (type I) is fully virtualized interfaces, such as IDE/SATA and NVMe, where QEMU emulates the device IO according to corresponding device specifications. In this mode, guest OS can run out-of-box with vanilla device driver without modification, thus providing flexible support to different operating systems. However, this flexibility comes at cost of low performance as those interfaces are not optimized for virtualization. Existing popular SSD emulators, such as VSSIM and LightNVM's QEMU, are based on this type of I/O virtualization, thus they cannot provide scalability we need to emulate high scalability in today's SSDs. The second type (type II) is para-virtualized storage interfaces, e.g. virtio. This is a special type of storage interface designed to work with virtual machines only, which brings better perforamance than full virtualization. However, even with virtio, we will show later that it's still not scalable enough to support our scalability needs. The third type is hardware assisted virtual I/Os, such as VT-d and SR-IOV. They depend on hardware support to assign (partial) device to the VM and achieve close to bare metal performance. Even so, due to QEMU software overhead in interrupt handling, it cannot deliver the scalability we need neither.

This leaves us to use ramdisk (e.g. tmpfs or block ram device) as the backend storage device and stack an emulated storage interface (type I or II) on top. Ramdisks are backed by DRAM, whose access latency is at 100ns level and thus is negligle compared to the NAND latency we are trying to emulate. In this case, we measure I/O latencies under different number of I/O threads to see how scalable QEMU/KVM is. The results are shown in Figure 5.2a. Since the backend is DRAM, the perceived latencies represent capability of QEMU/KVM in handling concurrent I/Os.

Unfortunately, stock QEMU exhibits a scalability limitation. For example, as shown in Figure

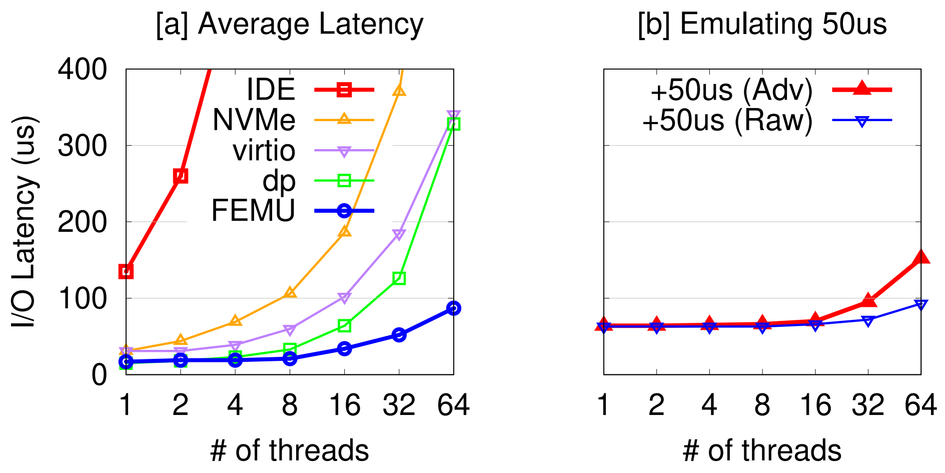


Figure 5.2: **QEMU Scalability.** *The figure shows the scalability of QEMU's IDE, NVMe, virtio, and dataplane (dp) interface implementations, as well as FEMU. The x-axis represents the number of concurrent IO threads running at the user level. Each thread performs random 4KB read IOs. The y-axis shows the user-perceived average IO latency. The storage is memory-backed, thus the IO latency represents the software overhead. The experiments were run on a dual-thread (2x) 24-core machine, hence no CPU contention. For Figure (a), the IDE and NVMe lines representing VSSIM and LightNVM's QEMU respectively are discussed in §5.2; virtio, dp, and FEMU lines in §5.3. For Figure (b), the "+50μs (Raw)" line is discussed in §5.4.1; the "+50μs (Adv)" line in "Result 3" part of §5.4.2.*

5.2a, the red line shows QEMU IDE cannot scale with number of increasing I/O threads at all since it's a two decade old interface designed to process one I/O at a time without parallelism. When number of I/O threads doubles, we can directly observe that I/O latency doubles. Thus, it's not a good fit. With QEMU NVMe (although it is more scalable than IDE), more IO threads still increases the average IO latency (*e.g.*, with 8 IO threads, the average IO latency already reaches 106μs). This is highly undesirable because typical read latency of modern SSDs can be below 100μs, let alone we need to emulated tens to hundreds of parallel I/Os.

More scalable alternatives to NVMe are para-virtual interfaces, as shown by the virtio and virtio-blk dataplane (dp) lines [32, 273]. (`virtio/dp` vs. `NVMe` lines in Figure 5.2a). virtio-blk dataplane (dp) extends the basic virtio-blk interface with a dedicated thread working in polling mode, thus it can achieve better scalalibty compared to virtio. However, these interfaces are not as extensible as NVMe since they only support simple read, write and flush I/O commands. NVMe, as a new standard storage interface designed for today's fast NVM devices, is lightweight and

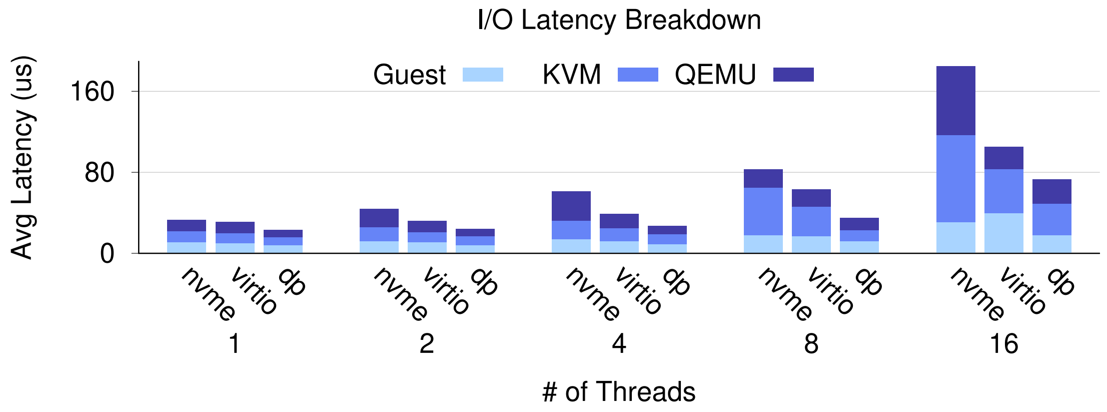


Figure 5.3: **QEMU latency breakdown under NVMe, virtio-blk (virtio) and virtio-dataplane (dp).** *This figures show the latencies brought by each parts in I/O life cycle. It helps us analyze the scalability of each part. "Guest OS" accounts for the time since the IO enters guest kernel until driver I/O submission, "KVM" represents the time taken for I/O virtualization handling in the host kernel plus QEMU post processing before entering corresponding device emulation layer, and "QEMU" shows the time of I/O processing in QEMU I/O emulation.*

more extensive, thus more popular now. Nevertheless, virtio and dp are also not scalable enough to emulate low flash latencies. For example, at 32 IO threads, their IO latencies already reach 185μs and 126μs, respectively.

5.3.2 QEMU I/O Scalabilty Problem Root Causes

As shown in §5.3.1, QEMU NVMe can only scale to 4 I/O threads, imposing severe scalability challenge in achieving high parallelism emulation. In this section, we first go through how NVMe protocol works and then introduce QEMU NVMe emulation. Finally, we identify the root causes of QEMU NVMe scalability bottleneck.

QEMU/KVM is a full system emulator/hypervisor, which can emulate various types of hardware. It supports running guest OS on top at near bare-metal performance for CPU/Memory intensive workloads with the help of hardware assisted virtualization techniques (e.g. Intel VT and EPT). However, I/O virtualization suffers significant overhead due to the necessacity to trap and emulate I/Os in the QEMU process, instead of running it inside the guest OS (a different context).

To the Operating System, an I/O device can be abstracted as a set of registers either mapped to OS memory space (i.e., MMIO[3]) or can be accessed via PIO (Port I/O) operations. Accesses to these registers are sensitive operations and needs to be babysitted by QEMU using a trap-and-emulate method. Put it simple, QEMU monitors access to these areas and takes control of the VM execution, in the meanwhile, vCPU is stalled until QEMU signals completion of above memory access. Those register accesses are triggered during I/O submissions/completions and will frequently interrupt VM execution by jumping back and forth. Once QEMU is done with emulating I/O by serving it from correponding backend image file in the host, it will pause the VM execution again to inject an interrupt to the guest OS (simulating the way how real interrupt works).

QEMU's NVMe implementation uses traditional trap-and-emulation method to emulate I/Os, thus the performance suffers due to frequent VM-exits. Each trap will cause an expensive VM-exit to be executed, which usually takes several microseconds to save VM context and restore host context. Under concurrent I/O workloads VM-exit delay will be queued up for inflight I/Os thus cascadingly affect overall I/O latency.

Moreover, as shown in Figure 5.3, each guest I/O needs go through guest kernel stack, KVM processing (host kernel) and QEMU (host user space). For QEMU/KVM, guest OS codes runs in dedicated vCPU threads, which are introspected by KVM module, QEMU I/O emulation mainly runs in an event loop thread. These are three major parts where I/O latency comes from. Compared to para-virtualized interfaces (`virtio` and `dp`), QEMU NVMe shows a significant overhead in "KVM" and "QEMU" due to its full-virtualization nature. For example, under 16 I/O threads, the time taken in "KVM+QEMU" for `NVMe` is 150μs while it's only 70μs and 45μs for `virtio` and `dp` respectively. This shows QEMU NVMe virtualization is the main reason of its scalability bottleneck. Detailed root causes is below.

Root Causes: Collectively, all of the scalability bottlenecks above are due to two reasons: **(1) Software overheads caused by frequent VM-exits.** For each NVMe I/O, the Guest OS' NVMe driver first "rings the doorbell [44]" to the device (QEMU in our case) that some IOs are in the

3. Memory Mapped I/O, which enables device register accesses via volatile memory reads and writes

device queue. This "doorbell" write is an MMIO operation that will cause an expensive VM-exit ("world switch" [294]) from the Guest OS to QEMU. Such VM-exits have been well-known as I/O virtualization performance bottlenecks for years. Worse, a similar doorbell operation must also be done upon IO completion to update completion queue head position, this doubles the number of VM-exits for each I/Os (corresponding to step ❷ and ❼ in Figure 2.2). Under concurrent I/Os, the VM will be frequently interrupted so as to leaving limited vCPU resource for inflight I/O handling. This is one reason why QEMU NVMe scale to even 4 I/O threads in Figure 2.2. **(2) QEMU AIO processing overheads.** QEMU uses asynchronous IOs (AIO) to perform the actual read/write (byte transfer) to the backing image file. This AIO component is needed to avoid QEMU being blocked by slow IOs (*e.g.*, on a disk image). However, the AIO overhead becomes significant when the storage backend is a RAM-backed image. According to our evaluations, QEMU AIO may take more than 20μs to finish while accessing data directly from the ramdisk backed image file only takes less than 1μs. Although QEMU uses a thread pool to distribute I/Os evenly, they don't help here since the overhead comes from QEMU's single threaded block I/O layers where I/O submissions are sequentialized. To be specific, upon receiving MMIO signal for I/O arrival notification, QEMU needs to go through NVMe device emulation layer, block driver layer, image format driver and raw device driver before I/Os can be put into QEMU's global AIO queue. This is already a long I/O path. Further, worker threads need to process these AIOs by submitting them to the host OS by traversing the whole host I/O stack. While QEMU/KVM depends on these different layers for features implemented in QEMU block layer, such as I/O throttling, VM migration, etc. These features are not needed in our case.

5.3.3 Scalability Solutions

Our solutions: To address these problems, we leverage the fact that FEMU purpose is for research prototyping, thus we perform the following modifications:

• **Polling based QEMU NVMe Design:** In order to overcome the excessive VM-exit overhead, we transform QEMU from an interrupt-driven (trap) to a polling-based design and disable the doorbell writes in the Guest OS (just 1 LOC commented out in the Linux NVMe driver). We create a dedicated thread in QEMU to continuously poll the status of the device queue (a shared memory mapped between the Guest OS and QEMU). This way, the Guest OS still "passes" control to QEMU but without the expensive VM exits. We emphasize that FEMU can still work and get better performance without the changes in the Guest OS as we report later. This optimization can be treated as an optional feature, but the 1 LOC modification is extremely simple to make in many different kernels.

In details, our polling design is enabled by the Shadow Doorbell Buffer Support proposed in NVMe Specification version 1.3[44, §7.10]. It introduces a set of shadow doorbell buffers which are shared memory buffers between guest OS and QEMU. Upon I/O submission and completion, corresponding shadow doorbell buffer will be updated for SQ/CQ tail/head updates and only when necessary will the doorbell register be written. Essentially shadow doorbell buffer adds para-virtualization capability to virtual NVMe controllers, like QEMU NVMe. Thus, it can used to enhance QEMU NVMe I/O performance. Potentially it will reduce the number of VM-exits, thus provide better scalability. Although Shadow Doorbell Buffer Support has been implemented in Linux kernel, QEMU NVMe lacks support to this feature.

To reap the benefits brought by this feature, we first enhance QEMU with shadow doorbell buffer capability. With this, we can observe obvious performance boost (Figure 5.4, `+dbbuf` line). However, the performance is still not scalable enough to support 32 parallel I/Os (not shown). The reason is that there still exits VM-exits caused by doorbell writes. While shadow doorbell buffer mechanism greatly reduce number of doorbell writes needed, they are not eliminated. Thus, I/O performance is affected under intensive workloads and show worse tail latencies. Moreover, QEMU still depends on doorbell writes for new I/O arrivals. As guest OS will send multiple I/Os before it rings the doorbell (thanks to shadow doorbell buffer support), QEMU is passively passed the control at a longer intervall, which may hurt overall I/O latencies.

Thus, to make QEMU NVMe more scalable, we use polling techniques. Polling works by proactively querying new I/O arrivals instead of passively waiting for control transfer. This way, polling can handle I/Os in a more timely manner. Our polling design leverage the fact that SQ/CQ tail/head updates are updated to shadow doorbell buffer as well as doorbell registers, we can check the shadow doorbell buffer peiriodically for latest I/O submissions or completions. If no updates are made to the shadow buffers (i.e., no new I/O submission or completion), we can simply skip QEMU I/O emulation logic and poll the status change again next time. In our current polling design, we utilize QEMU's timer APIs to setup a periodic event for this purpose. With polling, we can see from Figure 5.4, `+poll` that can sustain 400K IOPS under 64 I/O threads. However, from 32 to 64 threads, the aggregate IOPS only increase 19%, which implies we are hitting another bottleneck, which we solve in next section.

• **Customized QEMU AIO Path:** As pointed out earlier, the long I/O path (many layers) in QEMU's AIO module bring much overhead and straggle I/O performance. An initial thought is to figure out a way to shorten I/O path without hurting correctness. Considering FEMU purpose is only for research prototyping and doesn't care about rich features provided by QEMU, we compose our own memory backend and skip QEMU AIO component completely.

To be specific, we do not use virtual image file (in order to skip the AIO subcomponent). Rather, we create our own RAM-backed storage in QEMU's heap space (with configurable size `malloc()`). This way, we totally skip the QEMU block I/O layer and I/O path on the host stack, thus I/Os can be handled immediately after it enter the NVMe device emulation layer. Another problem arises when doing this is DMA correctness. Traditionally, QEMU emulate DMA operations are tightly coupled with the block I/O layer to trasfer data between guest OS and backend image file on the host. With our new memory backend, QEMU's DMA engine doesn't work. To tackle this problem, we then modify QEMU's DMA emulation logic to transfer data from/to our heap-backed storage, transparent to the Guest OS (*i.e.*, the Guest OS is not aware of this change). With these changes, we achieve ultimate performance approaching 1 million IOPS as shown in Figure 5.4

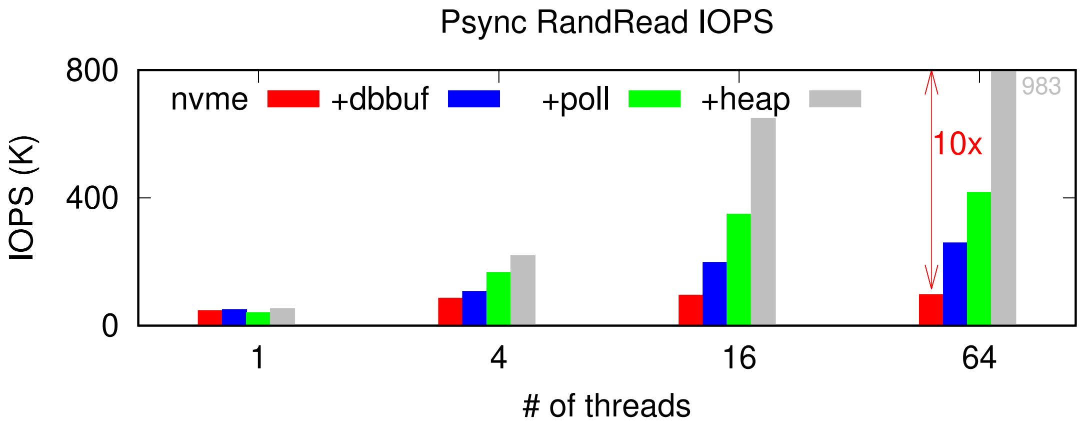


Figure 5.4: **QEMU NVMe IOPS w/ FEMU optimizations.** *The figure shows the scalability of QEMU's NVMe implementation under our optimizations.* `nvme` *line represents stock QEMU NVMe,* `+dbbuf` *represents QEMU NVMe with shadow doorbell buffer support,* `+poll` *adds polling based on* `+dbbuf`*, and finally* `+heap` *applies our own heap storage backend on top of all previous optimizations. The x-axis represents the number of concurrent IO threads running at the user level. Each thread performs random 4KB read IOs. The y-axis shows aggregate IOPS achieved.*

`+heap` line.

5.3.4 FEMU Optimization Results

Latency: The bold `FEMU` blue line in Figure 5.2a shows the scalability achieved. In between 1-32 IO threads, FEMU can keep IO latency stable in less than 52μs, and even below 90μs at 64 IO threads. If the single-line Guest-OS optimization is not applied (the removal of VM-exit), the average latency is 189μs and 264μs for 32 and 64 threads, respectively (not shown in the graph). Thus, we recommend applying the single-line change in the Guest OS to remove expensive VM exits.

IOPS: Figure 5.4 shows the IOPS we can achieve after applying shadow doorbell buffer support, polling design and our customized heap backend storage. It clearly demonstrates that our optimizations can greatly improve overall IOPS by 10$\times$ under 64 I/O threads. This enables FEMU to emulate 32 parallel channels/chips.

The remaining scalability bottleneck now only comes from QEMU's *single-thread* "event loop" [33, 166], which performs the main IO routine such as dequeueing the device queue, triggering

DMA emulations, and sending end-IO completions to the Guest OS. Worse, this thread must synchronize with all the running vCPU threads, incurring additional performance loss. Recent works addressed these limitations (with major changes) [121, 206], but have not been streamlined into QEMU's main distribution. We are exploring the possibility of integrating other solutions in future development of FEMU.

5.4 FEMU Accuracy

We now discuss the accuracy challenges. To accurately emulate an SSD in QEMU, two problems need to be solved: (1) NAND Flash Access Latency Emulation: since NAND reads are at $100\mu s$ level, FEMU needs to be able to accurately delay an I/O for certain amount of time to emulate NAND access and inter-firmware I/O queueing delays, without affecting other inflight I/Os coming from guest OS. (2) SSD Performance Model: we need detailed knowledge about the controller architecture and firmware logic to emulate I/O processings inside the SSD. In this section, we first describe our delay mechanism (§5.4.1), and then dive into our basic and advanced performance models (§5.4.2). Finally, we present FEMU accuracy results towards emulating an enterprise level OpenChannel-SSD.

5.4.1 Delay Emulation

NAND Flash supports page read, page program and block erase operations, at the latency of $100\mu s$, 1.5ms, and 6ms respectively[4]. That said, when a read is sent to the SSD, user will get the data after around $100\mu s$. Traditional SSD simulators don't simulate such delays using wall clock time. Instead, they usually maintain an internal state machine and advance it accordingly when handling an I/O event. For example, with an incoming read, it would simply do `+100`μ`s` to its state structure and return the I/O immediately. Later, it reports the I/O takes $(100+\Delta)\mu s$, where Δ is the extra

4. Latency numbers profiled from CNEX 2TB SSD using Micron L95B eMLC NAND

latency due to queueing, GC, etc. As an emulator, FEMU needs to delay I/O completion using wall clock time so as to emulate hardware latency behavior, giving guest OS/application an illusion that I/O processing in the emulated device takes that amount of time to finish, just as hardware SSD platforms do.

• **"Unsuccessful" Delay Emulation Efforts:** An intuitive way to do delay emulation is to use busy-loops or `sleep()`. `sleep()` would stall the thread thus it's not a good choice, especially under the case that current simulators and QEMU use single thread for I/O handling. Busy-loops can provide most precise delay emulation at the cost of CPU resource by constantly checking if end timestamp has expired. Doing delay emulation inside QEMU is tricky as it needs to be well incorporated with existing QEMU I/O processing framework, without affecting overall I/O performance. Below, we first describe our prior unsuccessful delay emulation efforts and then introduce our current endio-based delay emulation mechanism.

- **"putback"** method: This method utilizs the parallelism exposed by QEMU thread pool. We let each QEMU request carry an ending timestamp, and when worker threads fetch an I/O, they will check if request timestamp has expired, if not, they will put it back to the AIO queue and wait for next time processing.
- **"busyloop"** method: While busyloop in QEMU's single threaded submission path is not feasible, we perform busy-loop in the worker thread (we can busy-loop 64 inflight I/O at the same time which is limited by thread pool size of 64).
- **"timer"** method: For each I/O, we setup a timer event to delay it for certain amount of time before submitting it into QEMU AIO queue. Timer event works in a similar manner to `sleep()`, but it won't block other I/Os (asynchronous manner).

Figure 5.5a shows the results achieved by using different delay emulation methods. For a guest application I/O, the guest kernel stack overhead is 10-20μs in our platform. When we try to emulate a device latency of 50μs for each I/O, the expected I/O latency perceived by guest aplication

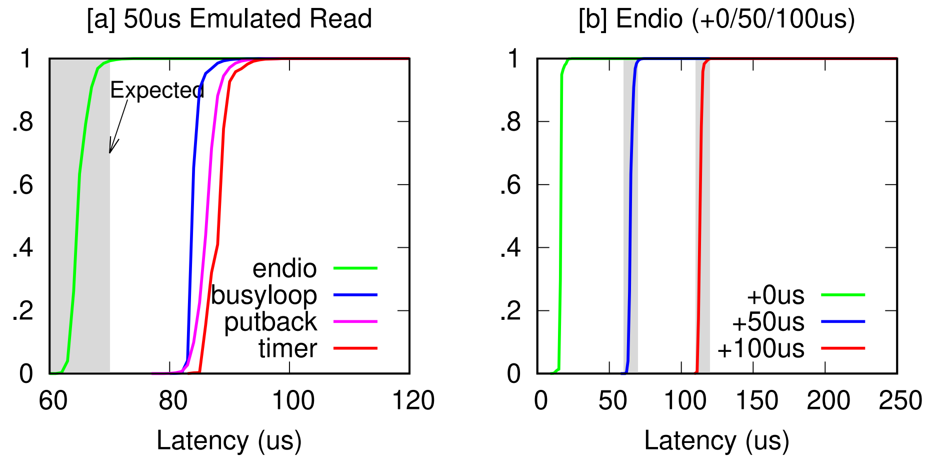


Figure 5.5: **Delay Emulation: 50μs and 100μs.** *Emulating 50μs and 100μs device access latency, FEMU delay emulation doesn't introduce tail latencies.*

I/Os should be in the range of 60-70μs, as shown in the gray area. The `putback`, `busyloop` and `timer` lines correspond to above mentioned delay emulation methods. However, all of them show a 20-30μs offset in the emulated latencies. The reasons are (1). These methods run deep in QEMU's block I/O layer and don't consider QEMU overhead during I/O emulation. (2). These methods lower QEMU AIO processing efficiency by hogging more CPU resources, thus preventing stable I/O latencies. With these experiences, we design `endio` based delay emulation technique to overcome drawbacks in previous designs.

• **Endio Delay Emulation:** When an IO arrives, FEMU will issue the DMA read/write command, then label the IO with an emulated completion time (T_{endio}) and add the IO to our "end-io queue," sorted based on IO completion time. FEMU dedicates an "end-io thread" that continuously takes an IO from the head of the queue and sends an end-io interrupt to the Guest OS, once the IO's emulated completion time has passed current time ($T_{endio}{>}T_{now}$).

• **Endio Delay Emulation Results:** The "+50us (Raw)" line in Figure 5.2b shows a simple (and stable) result where we add a delay of 50μs to *every* IO ($T_{endio}{=}T_{entry}{+}50\mu$s). Note that the end-to-end IO time is more than 50μs because of the Guest OS overhead (roughly 20μs). Important to say that FEMU also does not introduce severe latency tail. In the experiment above, 99% of all the

IOs are stable at 70μs. Only 0.01% (99.99th percentile) of the IOs exhibit latency tail of more than 105μs, which already exists in stock QEMU. For example, in VSSIM, the 99th-percentile latency is already over 150μs. This shows that endio method is salable and efficient in emulate device latencies.

Let's take a closer look at the I/O latency distribution of this endio method. As shown in Figure 5.5a, `endio` line sits entirely in the expected gray area, proving its advantage over other methods (to the right). Further, in Figure 5.5b, we also get similar results when trying to emulate 100μs latency for each I/O.

5.4.2 SSD Performance Models

FEMU scalability and endio delay emulation have paved the way for FEMU to be able to emulate an SSD. What's missing here is the performance model of the SSD, which defines the internal controller architecture and firmware (FTL) used to manage I/Os working around NAND limitations and guaranteeing high performance.

Overview of an SSD controller architecture: Due to NAND Flash material level limitations, such as asymmetric access granularity for read/program and erase, non-in-place updates, limited P/E (program/erase) cycles and data retention time, today's SSDs are usually made as a SoC (System on Chip) with its own processor, SRAM/DRAM, and firmware/FTL (Flash Translation Layer). FTL is the soul of an SSD and it's responsible for mapping table management, caching, I/O scheduling, GC (Garbage Collection), background scrubbing, etc. NAND Flash chips are organized into channels (e.g. 16), with several (e.g. 8) chips mounted on one channel. Controller communicates with NAND Flash chips by sending/receiving information through the channel, which includes NAND commands, address and data transfer.

NAND Operations: Take NAND read process as an example. The controller first sends `NAND read` command to the command register, put page address to the addr register, and then NAND chip will go busy reading data from NAND cell array into its internal page buffer. At this time, the controller is free to do anything else as NAND chip is an independent operation unit. Controller can query the NAND read progress by sending `read status` command, and this is usually offloaded to a specific hardware module by proactive status polling without blocking the controller. When NAND read operation is done (data in page buffer), the controller will send `read data to controller` command, and transfer data to controller DRAM through the channel. The data is first decoded by ECC engine to check if any error happens before returning it back to user. Otherwise, `read retry` will kick in to try hard to read the data out without errors. NAND program operation is similar, but needs to send command to transfer data to NAND page buffer first and then issue `NAND program` command which will write store data into NAND cell arrays. Erase is simple, the controller sends `block erase` and the NAND will become busy with erasing data in corresponding block. Once operation is finished, NAND Flash chip will be in `ready` status again, meaning it can accept new commands for processing.

Channel: All the NAND Flash chips on the same channel share one bus, which is multiplexed for command, address and data transfer. When channel is busy with data transfer from/to a NAND chip, it's in `busy` status, and at this time, other pending operations which need to use channel must wait for prior operation completion. Thus, the channel may be contentious and limit overall parallelism among NAND chips mounted on the same channel.

Since command and address transfer only take several nanoseconds, In our performance models, FEMU only emulates the data transfer latency. Below we present two delay model aiming for accurate emulation toward a commercial OpenChannel-SSD. With these delay models, FEMU is also able to run a FTL inside QEMU (as prior project VSSIM does) taking I/O latencies from various sources into account.

Basic Delay Model: The challenge now is to compute the end-io time (T_{endio}) for every IO accurately. We begin with a basic delay model by marking every plane and channel with their next free time (T_{free}). For example, if a page write arrives to currently-free channel #1 and plane #2, then we will advance the channel's next free time ($T_{freeOfChannel1}$=T_{now}+$T_{channel}$,

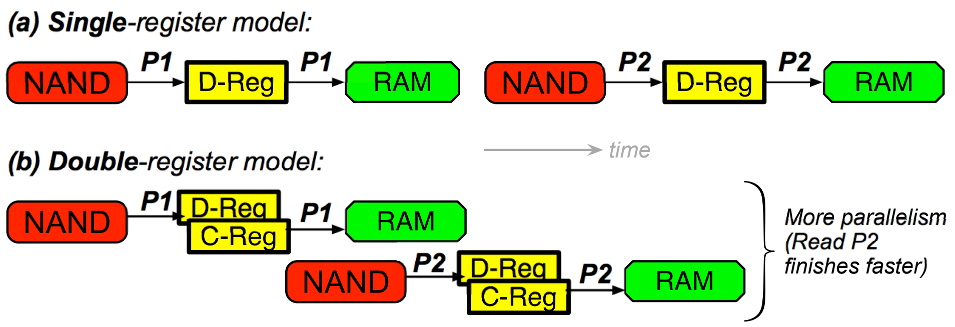

(a) Single-register model:
NAND
P1
D-Reg
P1
RAM
NAND
P2
D-Reg
P2
RAM
time
(b) Double-register model:
NAND
P1
D-Reg
C-Reg
P1
RAM
NAND
P2
D-Reg
C-Reg
P2
RAM
More parallelism (Read P2 finishes faster)

Figure 5.6: **Single- vs. double-register model.** *(a) In a single-register model, a plane only has one data register (D-Reg). Read of page P2 cannot start until P1 finishes using the register (i.e., the transfer to the controller's RAM completes). (b) In a double-register model, after P1 is read to the data register, it is copied quickly to the cache register (D-Reg to C-Reg). As the data register is free, read of P2 can begin (in parallel with P1's transfer to the RAM), hence finishes faster.*

where $T_{channel}$ is a configurable page transfer time over a channel) and the plane's next free time ($T_{freeOfPlane2}$+=T_{usrWrt}, where T_{usrWrt} is a configurable write/programming time of a NAND page). Thus, the end-io time of this write operation will be T_{endio}=$T_{freeOfPlane2}$.

Now, let us say a page read to the same plane arrives while the write is ongoing. Here, we will advance $T_{freeOfPlane2}$ by T_{read}, where T_{read} is a configurable read time of a NAND page, and $T_{freeOfChannel1}$ by $T_{channel}$. This read's end-io time will be T_{endio}=$T_{freeOfChannel1}$ (as this is a read operation, not a write IO).

In summary, this basic *queueing* model represents a *single-register* and *uniform page latency* model. That is, every plane only has a single page register, hence cannot serve multiple IOs in parallel (*i.e.*, a plane's T_{free} represents IO serialization in that plane) and the NAND page read, write, and transfer times (T_{read}, T_{usrWrt} and $T_{channel}$) are all *single* values. We also note that GC logic can be easily added to this basic model; a GC is essentially a series of reads/writes (and erases, T_{erase}) that will also advance plane's and channel's T_{free}accordingly. Similarly, queueing delay can also be well emulated. If an incoming I/O finds target channel/chip with a free time-stamp in the future, then it means corresponding channel/chip is busy and we will mark the I/O latency as `waiting time` plus the time needed to serve this I/O when it's scheduled for execution.

Advanced "OC" Delay Model: While the model above is sufficient for basic comparative research (*e.g.*, comparing different FTL/GC schemes, some researchers might want to emulate the detailed intricacies of modern hardware. Below, we show how we extend our model and achieve a more accurate delay emulation of OpenChannel SSD ("**OC**" for short). OC is a new kind of SSDs which expose its internal architecture to the host, thus allowing moving FTL management up to the OS. Our OC is a CNEX Westlake PCIe SSD with 16 channels and in total 128 NAND chips. Nowadays, Linux supports running OC with a kernel-level FTL named LightNVM. LightNVM implements basic sector-based mapping table, data placement and GC functions.

The OC's NAND hardware has the following intricacies. First, OC uses *double-register* planes; every plane is built with two registers (data+cache registers), hence a NAND page read/write in a plane can overlap with a data transfer via the channel to the plane (*i.e.*, more parallelism). Figure 5.6 contrasts the single- vs. double-register models where the completion time of the second IO to page P2 is faster in the double-register model.

Second, OC uses a *non-uniform* page latency model; that is, pages that are mapped to upper bits of MLC cells ("upper" pages) incur higher latencies than those mapped to lower bits ("Lower" pages); for example 48/64μs for lower/upper-page read and 900/2400μs for lower/upper-page write. Making it more complex, the 512 pages in each NAND block are not mapped in a uniformly interleaving manner as in "`LuLuLuLu...`", but rather in a specific way, "`LLLLLLuLLuLLuu...`", where pages #0-6 and #8-9 are mapped to Lower pages, pages #7 and #10 to upper pages, and the rest ("...") have a repeating pattern of "`LLuu`".

In addition, we built an efficient OC-extension to FEMU NVMe (based on LightNVM's QEMU base OC extension structures). With these, we are able to run LightNVM on top of FEMU.

Sizes		Latencies	
SSD Capacity	128 GB	Page Read	48μs; 64μs (MLC)
#Channels	2	(flash-to-register)	
#Planes/channel	8	Page Write	900μs; 2400μs (MLC)
Plane size	8 GB	(register-to-flash)	
#Planes/chip	2	Page data transfer	60μs
#Blocks/plane	1024	(via channel)	
#Pages/block	512	Block erase	6ms
Page size	16 KB		

Table 5.2: **FEMU emulated OpenChannel-SSD Parameters.** *We use the above parameters for the emulated SSD. Latency numbers are profiled from a real enterprise OpenChannel-SSD. The latencies are based on average values; actual latencies can vary due to read retry, different voltages, etc. Flash reads/writes must use the plane register. We use 128 GB out of 256 GB physical memory to serve as the emulated SSD backend storage. For the microbenchmark experiments, we change #Channels & #Planes/channel combinations to verify the latency accuracy under different settings.*

5.5 FEMU Accurary Results

5.5.1 Workloads & Experiment Setups

We use fio to stress test FEMU and OC under different configurations with `direct=1` on raw devices. We use filebench to compare FEMU and OC under real world application workloads, including File Server, Network FS, OLTP, Varmail, Video Server and Web Proxy.

We assign our OC to a VM and access it from inside the guest OS for fair comparison with FEMU emulated OC. Need to mention that assigned OC in VM doesn't suffer performance drop compared to OC running on physical machines. Our host machine consists of 2x Intel(R) Xeon(R) CPU E5-2670 v3 running at a base frequency of 2.30GHz with 256GB DRAM. FEMU emulated SSD takes up 128GB memory. HyperThreading, C/P-states and Intel Turbo Boost are turned off for consistent and max performance. We also pin FEMU threads to physical cores to avoid latency jitters caused by process migrations. Hugepage is used for minimum GPA-HVA (guest physical addr to host virtual addr) address translations.

By incorporating this detailed model, FEMU can act as an accurate drop-in replacement of OC, which we demonstrate with the following results.

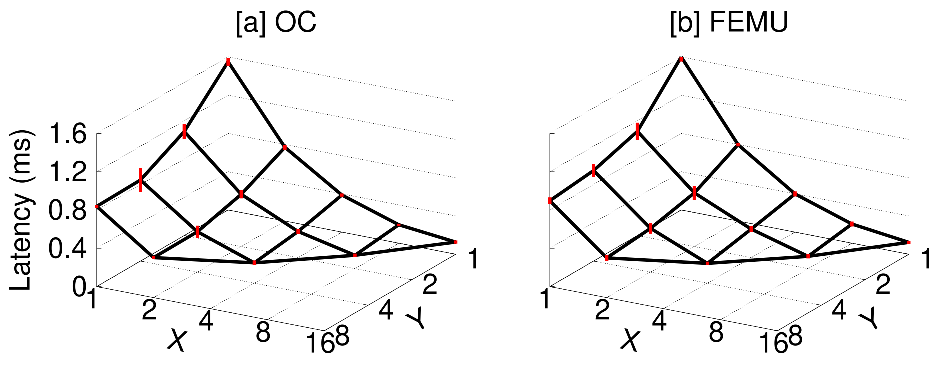


Figure 5.7: **OpenChannel SSD (OC) vs. FEMU.** *X: # of channels, Y: # of planes per channel. The figures are described in §5.5.2. The red line represents latency standard errors. For each configuration set (X, Y), we run the same fio workloads on both FEMU and OC and collect latency numbers. This experiment is to show FEMU can accurately emulate the queueing delay when multiple I/Os are sent to one NAND chip and parallelism when multiple I/Os are served evenly by multiple NAND chips.*

5.5.2 MicroBench Results

Result 1: Figure 5.7 compares the IO latencies on OC vs. FEMU. The workload is 16 IO threads performing random reads uniformly spread throughout the storage space. We map the storage space to different configurations. For example, x=1 and $\surd$=1 implies that OC and FEMU are configured with only 1 channel and 1 plane/channel, thus as a result, the average latency is high (z>1550μs) as all the 16 concurrent reads are contending for the same plane and channel. The result for x=16 and $\surd$=1 implies that we use 16 channels with 1 plane/channel (a total of 16 planes). Here, the concurrent reads are absorbed in parallel by all the planes and channels, hence a faster average read latency (z<130μs). Overall, Figures 5.7a and 5.7b exhibit a highly similar pattern, showing the success of our queuing delay emulation. The latency difference (error) is only between 0.8-11.6%; $Error{=}(Lat_{femu}{-}Lat_{oc})/Lat_{oc}$.

5.5.3 MacroBench Results

Result 2: Figure 5.8a shows the results from running several macrobenchmarks with six filebench personalities, with 16 IO threads of concurrent reads/writes on 16 planes across 4 channels. The

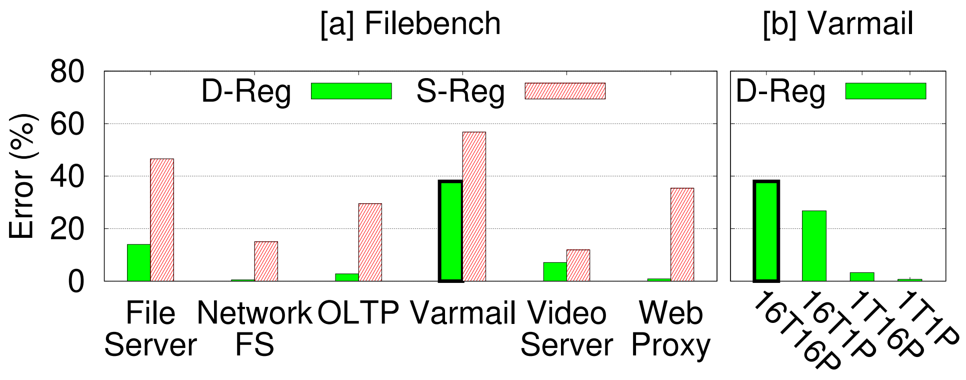


Figure 5.8: **Filebench on OpenChannel SSD (OC) vs. FEMU.** *The figures are described in the "Result 2" segment of §5.5.3. The y-axis shows the latency difference (error) of the benchmark results on OC vs. FEMU (Error=$(Lat_{femu}-Lat_{oc})/Lat_{oc}$). D-Reg and S-Reg represent the advanced and basic model respectively. The two bars with bold edge in Figures (a) and (b) are the same experiment and configuration (varmail with 16 threads on 16 planes).*

figure only shows the latency difference ($Error$) which contrasts the accuracy of our basic and advanced delay models. With the basic model, the resulting latencies are highly inaccurate (12-57%), but with the advanced model, the error drops to only 0.5-38%, which are 1.5-40× more accurate across the six benchmarks.

We believe that these errors are reasonable as we deal with delay emulation of tens of μs granularity. We leave further optimization for future work; we might have missed other OC intricacies that should be incorporated into our advanced model (as explained at the end of §5.2, OC only exposes channels and chips, but other details are not exposed by the vendor). Nevertheless, we investigate further the residual errors, as shown in Figure 5.8b. Here, we use the `varmail` personality but we vary the #IO threads [T] and #planes [P]. For example, in the 16 threads on 16 planes configuration (x="16T16P" in Figure 5.8b, which is the same configuration used in experiments in Figure 5.8a), the error is 38%. However, the error decreases in less complex configurations (*e.g.*, 0.7% error with single thread on single plane). Thus, higher errors come from more complex configurations (*e.g.*, more IO threads and more planes), which we explain next.

Result 3: We find that using an advanced model requires more CPU computation, and this compute overhead will backlog with higher thread count. To show this, Figure 5.2b compares the

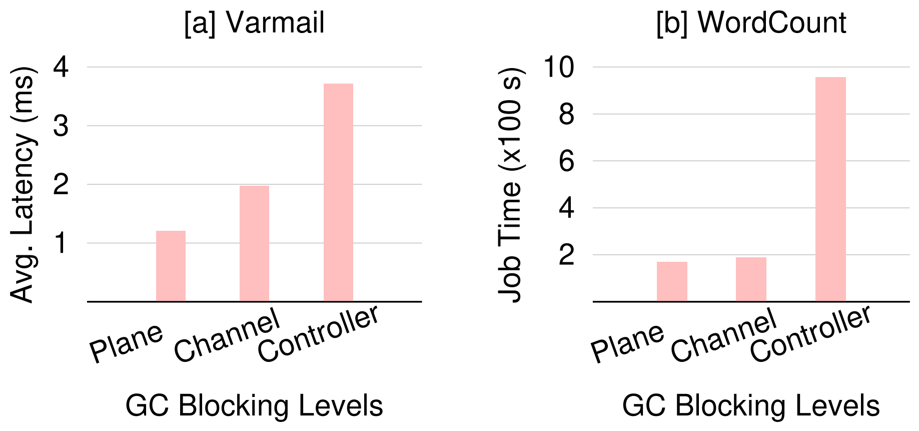


Figure 5.9: **Use examples.** *Figure 5.9a is described in the "FTL and GC schemes" segment of Section 5.6. Figure 5.9b is discussed in the "Distributed SSDs" segment of Section 5.6.*

simple +50μs delay emulation in our raw implementation (§5.4.1) vs. advanced model. Here, both cases simply add +50μs, but the advanced model must traverse many `if-else` statements (to check register, plane, and channel next free time), hence the compute overhead. Further scalability optimizations, as discussed at the end of §5.3 can help.

5.6 FEMU Usability

Being a *software*-based emulation platform, FEMU can be extended in many different ways. We now describe existing features/usabilities of FEMU, briefly showcase successful extensions used in our recent work [164, 318] as well as possible future work that FEMU features enable.

• **FTL and GC schemes:** In default mode, our FTL employs a dynamic mapping and a channel-blocking GC as used in other simulators [101, 169]. One of our projects uses FEMU to compare different GC schemes: controller, channel, and plane blocking [318]. In controller-blocking GC, a GC operation "locks down" the controller, preventing any foreground IOs to be served (as in OpenSSD [46]). In channel-blocking GC, only channels involved in GC page movement are blocked (as in SSDSim [169]). In plane-blocking GC, the most efficient one, page movement only flows within a plane without using any channel (*i.e.*, "copyback" [26]). Sample results are shown in Figure 5.9a. Beyond our work, recent works also show the benefits of SSD partitioning for performance isolation [126, 170, 199, 251, 292], which are done on either a simulator or a hardware platform. More partitioning schemes can also be explored with FEMU.

Possible future research: Decades of FTL/GC research mainly uses simulators, but future FTL/GC research can also be done with FEMU.

• **White-box vs. black-box mode:** FEMU can be used as **(1)** a white-box device such as Open-Channel SSD where the device exposes physical page addresses and the FTL is managed by the OS such as in Linux LightNVM or **(2)** a black-box device such as commodity SSDs where the FTL resides inside FEMU and only logical addresses are exposed to the OS.

• **Multi-device support for flash-array research:** FEMU is configurable to appear as multiple devices to the Guest OS. For example, if FEMU exposes 4 SSDs, inside FEMU there will be 4 separate NVMe instances and FTL structures (with no overlapping channels) managed in a single QEMU instance. Previous emulators (VSSIM and LightNVM's QEMU) do not support this. We are not aware of any software-based emulator that can emulate a flash array. To setup an emulated SSD array, one must assemble network block device connections to multiple machines running QEMU. A nested virtualization is another alternative, but nevertheless, these methods add a significant overhead to an already stringent latency requirement (§5.3).

• **Disaggregated Flash:** Disaggregation makes centralized resource management easier and helps improve resource utlilization. Historically, iSCSI has been the dominating networking storage protocol for remote block storage access. To keep up with fast speeds of today's SSDs, a new protocol named NVMe over Fabrics (NVMe-oF) has been standardized to support NVMe Flash Disaggregation. To overcome the software overhead brought by context swithes, interrupts due to operating systems management overhead, user space based I/O framework, such as SPDK is a popular choice in conjunction with NVMe-oF. As a first step, we have identified FEMU's full support to run SPDK and FEMU emulated NVMe device supports NVMe-oF. Since NVMe-oF requires RDMA to work, we use a software-based solution (SoftROCE) without using an expensive

RDMA capable NIC. In our setup with two VMs, we can successfully access the NVMe SSD exposed by the other VM through NVMe-oF.

5.7 FEMU Extensibility

• **Extensible OS-SSD NVMe commands:** **(1)** As FEMU supports NVMe, new OS-to-SSD commands can be added (*e.g.*, for host-aware SSD management or split-level architecture [274]). Currently in LightNVM, a GC operation reads valid pages from OC to the host DRAM and then writes them back to OC. This wastes host-SSD PCIe bandwidth; LightNVM foreground throughput drops by 50% under a GC. Our conversation with LightNVM developers suggests that one can add a new "`pageMove fromAddr toAddr`" NVMe command from the OS to FEMU/OC such that the data movement does not cross the PCIe interface. As mentioned earlier, split-level architecture is trending [134, 188, 267, 300, 320] and our NVMe-powered FEMU can be extended to support more commands such as transactions, deduplication, and multi-stream. **(2)** As mentioned earlier, split-level architecture is trending; our NVMe-powered FEMU can be extended to support other commands such as transactions, deduplication, and multi-stream [134, 188, 202, 246, 267, 300, 320]. **(3)** Techniques that manage GCs across an array of SSDs, many were only done with simulators [209], can also be supported by FEMU multi-device and NVMe supports.

Successful project: Combining FEMU's NVMe and multi-device supports, we have built an optimized Linux Software RAID-5 on an array of transparent SSDs. Specifically, if an SSD cannot serve a read request because of a conflicting GC, the transparent SSD will return an `EBUSY` error to the OS, which then will trigger our RAID-5 to reconstruct the non-available data from other SSDs. Moreover, if our RAID layer receives multiple `EBUSY`s from more than one SSD (within a read stripe), our RAID layer will resubmit the read with a new preemption flag turned on (via NVMe), which will slightly postpone the GC.

• **Page-level latency variability:** As discussed before (§5.4), FEMU supports page-level latency variability. Among SSD engineers, it is known that "not all chips are equal." High quality chips are mixed with lesser quality chips as long as the overall quality passes the standard. Bad chips can induce more error rates that require longer, repeated reads with different voltages. FEMU can also be extended to emulate such delays.

• **Distributed SSDs:** Multiple instances of FEMU can be easily deployed across multiple machines (as simple as running Linux hypervisor KVMs), which promotes more large-scale SSD research. For example, we are also able to evaluate the performance of Hadoop's wordcount workload on a cluster of machines running FEMU, but with different GC schemes as shown in Figure 5.9b. Since HDFS uses large IOs, which will eventually be striped across many channels/planes, there is a smaller performance gap between channel and plane blocking across the three GC mechanisms. We hope FEMU can spur more work that modifies the SSD layer to speed up distributed computing frameworks (*e.g.*, distributed graph processing frameworks).

Possible future research: Many out-of-core distributed graph processing frameworks were recently proposed [234, 332]. This type of research modifies the frameworks to work well on disks/SSDs. On the contrary, there is little work that modifies the SSD layer to speed up the frameworks (§5.2). We hope FEMU can spur more work in this important area.

• **Page-level fault injection:** Beyond performance-related research, flash reliability research [245, 278] can leverage FEMU as well (*e.g.*, by injecting page-level corruptions and faults and observing how the high-level software stack reacts).

• **Interface:** FEMU supports all modern interfaces, NVMe and virtio. The GuestOS will read/write through either of the interface. Inside FEMU, we have all the basic SSD internal functionalities such as the FTL module, GC and wear-leveling algorithms, delay emulation, performance monitor.

5.8 Conclusion

As modern SSD internals are becoming more complex, their implications to the entire storage stack should be investigated. In this context, we believe FEMU is a fitting research platform. We hope that our cheap and extensible FEMU can speed up future SSD research.

CHAPTER 6

DISCUSSION

6.1 MITTOS Limitations and Discussions

What are the limitations of MITTOS*?* First, tail latencies can be caused by software bugs; for example, an IO path can hit a non-deterministic bug that triggers a long lock contention or inefficient loops. Such slow and buggy paths are hard to foresee.

Second, while rare, hardware performance can degrade over time due to many factors [165, §2], or the other way around, performance can improve as device wears out (*e.g.*, faster SLC programming time as gate oxide weakens [152, §3.3]). This suggests that latency profiles must be recollected over time; a sampling runtime method can be used to catch a significant deviation.

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 including CPU and network, as we discuss below. If MITTOS is only applied to a subset of the resources (*e.g.*, storage stack only), then other approaches are still needed. We want to emphasize that other techniques such as hedged/tied requests *can co-exist* with MITTOS. In such a setting, MITTOS still delivers its benefits; applications can failover fast if the contention is within the storage stack. Timeouts are used as the last resort.

How do users/applications set deadline values? So far, we use the p95 expected value, but automating the setup of deadline/SLO values in general is an open research problem [187]. We believe EBUSY signals can help applications set a more optimum deadline. For example, too many EBUSYs imply that the deadline is too strict, but rare EBUSYs and longer tail latencies imply that the deadline is too relax. The open challenge is to find a "sweet spot" in between, which we leave for future work.

Can MITTOS *principles be applied to other resources?* We strongly believe MITTOS 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 [317]. 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 [240, 253], but we find that the stall cannot be completely eliminated; in the last 3 months, we study the implementations of many Java GC algorithms (Yak [253], G1GC [144], CMS and PS [12]) and find that, in the current architecture, `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 [99], which can easily induce tail latencies to applications such as SMR-backed key-value stores [242, 264]. MITTOS can be applied naturally in this context, also empowered by the development of SMR-aware OS/file systems [18, 100].

6.2 TEAFA Discussions

• **Fine-grained time window:** We can make our time-window implementation more fine grained. Right now the time window is coarse grained, *e.g.*, an SSD is not allowed to do GC for 1 second, while the rest are allowed to.

Important to note is that concurrent GCs that delay pages in different stripes are tolerable. For example, consider two full-stripe I/Os A and B that each will create seven parallel pages to seven SSDs ($A_1..A_7$ and $B_1..B_7$). It is possible that a GC in SSD1 blocks A_1 and another concurrent GC

	1	2	3	4	5	6	7	8
0N–1N	×	.	.	.	.	.	.	.
1N–2N	.	×	.	.	.	.	.	.
2N–3N	.	.	×	.	.	.	.	.
3N–4N	.	.	.	×	.	.	.	.
4N–5N	.	.	.	.	×	.	.	.
5N–6N	.	.	.	.	.	×	.	.
6N–7N	.	.	.	.	.	.	×	.
7N–8N	.	.	.	.	.	.	.	×

Table 6.1: **Fine-grained time window.** *This table shows an example fine-granular time window mechanism based on SSD LBA ranges. Here, 1–8 on top row represents the 8 SSDs in the same RAID group, and each $[iN..(i+1)N]$ represents the LBA range from iN to $(i+1)N$. "×" represent BG operations are allowed to happen in the SSD while "." means BG operations are disallowed.*

in SSD2 blocks B_2. Let's assume one parity per stripe (r=1). As long as parities A_8 and B_8 are not blocked, TEAFA can tolerate the two GCs as they delay pages in different stripes. This is the reason why TEAFA can tolerate r delayed pages per I/O stripe. So what we can do is to provide a **2 dimensional** time window.

For example, SSD1 is allowed to do GC from LBA [0 to N), but other SSDs are not allowed to GC on LBA [0 to N). At the same time window, SSD2 is allowed to do GC on from LBA [N to 2N), but other SSDs are not allowed to do it. In other words in every time window we have a 2-dimensional configuration where the x-axis is the SSD numbers and the y-axis is the logical partitioning of the LBAs (which we can configure). For example, lets' say we break the per-SSD LBA into 4 logical partition, each with N bytes: $0-N$, $N-2N$, $2N-3N$, $3N-4N$.

Thus the configuration for the first time window is like this for 8 SSDs , from 1 to 8 are like:

From Table 6.1, we can see that SSD1 is only allowed to do GC within 0N-1N LBAs, and so on. In the next few time windows, we slide the configuration accordingly. The advantage of this approach is that the SSDs are all still doing GC at the same time.

However, this more fine-grained approach is more challenging to implemenet. Let's say to utilization full parallelism, we say L1 are mapped to row #1, L2, mapped to row#2. well when we GC data in L2, we can use a copyback mechanism where data in L1 and L2 does not leave, hence won't have any contention. But these days GC copyback is not enabled, because during GC the SSD piggyback ECC checking to check that data is valid. In this case, the controller must read the data via the channel to the DRAM, hence the channel will be busy and contend with the user IOs. We leave further exploration as future work.

• **Why not handling writes:** We do not address write latency because writes are buffered. In addition, applications tend to do buffering by themselves, and with the prevalence of persistent memory, foreground write are logged and are mapped to NVM to have deterministic performance. On the other hand, NVM might not big enough to handle all reads. And internally in SSD, writes can be flushed to free chips, while reads cannot be easily redirected.

• **Scalability:** While our evaluation is mainly done on RAID-5 with 4 FEMU drives, TEAFA can be easily extended to more drives beyond RAID-5 (*e.g.*, RAID-6, erasure coding, or RAID-1, etc.). We have given the max RAID width (N_d) supported by TEAFA, and to do so, the SSD vendors only need to adust the time window value accordingly based on our constraints. Moreover, TEAFA does not harm the reliability of the flash array as TEAFA busy read are already distinguished from normal failed IOs. Under device failure, TEAFA reconstruction busy reconstruction logics can be dynamically disabled and later resumed when the RAID is back (*e.g.*, by swapping out the old drive and swap in new ones).

6.3 TTFLASH Limitations and Discussions

We now summarize the limitations of TTFLASH, TEAFA, and MITTOS and discuss their designs.

6.3.1 TTFLASH Limitations

First, TTFLASH depends on RAIN, hence the loss of one channel per N channels. Increasing N will reduce channel loss but cut less tail latencies under write bursts. Under heavy write bursts, TTFLASH cannot cut all tails Finally, TTFLASH requires intra-plane copybacks, skipping ECC checks, which requires future work as we address below.

• **ECC checking** *(with scrubbing)*: ECC-check is performed when data pass through the ECC engine (part of the controller). On foreground reads, before data is returned to the host, ECC is *always* checked (TTFLASH does *not* modify this property). Due to increasing bit errors, it is suggested that ECC checking runs more frequently, for example, by forcing all background GC copyback-ed pages read out from the plane and through the controller, albeit reduced performance.

TTFLASH, however, depends on *intra-plane* copybacks, which implies *no* ECC checking on copyback-ed pages, potentially compromising data integrity. A simple possible solution to compensate this problem is periodic idle-time scrubbing within the SSD, which will force flash pages (user and parity) flow through the ECC engine. This is a reasonable solution for several reasons. First, SSD scrubbing (unlike disk) is fast given the massive read bandwidth. For example, a 2 GB/s 512-GB client SSD can be scrubbed under 5 minutes. Second, scrubbing can be easily optimized, for example, by only selecting blocks that are recently GC-ed or have higher P/E counts and history of bit flips, which by implication can also reduce read disturbs. Third, periodic background operations can be scheduled without affecting foreground performance (a rich literature in this space [104]). However, more future work is required to evaluate the ramifications of background ECC checks.

• **Wear leveling** *(via horizontal shifting and vertical migration)*: Our static RAIN layout (§3.3.2) in general does not lead to wear-out imbalance in common cases. However, rare cases such as random-write transactions (*e.g.*, MSNFS) cause imbalanced wear-outs (at chip/plane level).

Imbalanced wear-outs can happen due to the two following cases: (1) There is write imbalance *within* a stripe (MSNFS exhibits this pattern). In Figure 3.3 for example, if in stripe S_0 {$012P_{012}$}, LPN 1 is more frequently updated than the rest, the planes of LPN 1 and P_{012} will wear out faster than the other planes in the same group. (2) There is write imbalance *across* the stripes. For example, if stripes in group G_0 (*e.g.*, stripe {$012P_{012}$}) are more frequently updated than stripes in other groups, then the planes in G_0 will wear out faster.

The two wear-out problems above can be fixed by *dynamic* horizontal shifting and vertical migration, respectively. With horizontal shifting, we can shift the parity locations of stripes with imbalanced hot pages. For example, S_0 can be mapped as $\{12\mathrm{P}_{012}0\}$ across the 4 planes in the same group; LPN 1 and P will now be directed to colder planes. With vertical migration, hot stripes can be migrated from one plane group to another ("vertically"), balancing the wear-out across plane groups.

As a combined result, an LPN is still and always statically mapped to a stripe number. A stripe, by default, is statically mapped to a plane group and has a static parity location (*e.g.*, S_0 is in group G_0 with P_{012} behind channel C_3). However, to mark dynamic modification, we can add a "mapping-modified" bit in the standard FTL table (LPN-PPN mapping). If the bit is zero, the LPN-PPN translation performs as usual, as the stripe mapping stays static (the common case). If the bit is set (the rare case in rare workloads), the LPN-PPN translation must consult a new stripe-information table that stores the mapping between a stripe (S_k) to a group number (G_i) and parity channel position (C_j).

6.4 LeapIO Discussion

6.4.1 Threat Model

In LeapIO threat model, both ARM SoC and SoC_{VM} are trusted and are allowed to access necessary host resources to fulfill cloud storage stack offloading purposes for efficient data path. Our ARM-SoC and SoC_{VM} are first-party entities which are only accessed by us. They interact with applications only via queues in shared memory and therefore, do not expose extra attack surfaces.

In LeapIO framework, SoC is owned and solely used the cloud provider, thus it is trusted. It needs to prevent potential exploits from malicious users. Potentially malicious user VMs can only interact with LeapIO runtime via the shared queue pair interface, which is under the guidance of host hypervisor during creation phase, thereby revealing no security risks. All the NVMe com-

Cloud Providers	Alibaba	Azure	EC2	GCP
1 vCPU yearly price	$400	$320	$364	$500

Table 6.2: **TCO Analysis.** *Single vCPU yearly pricing calculated using the entry-level VM types provided by major cloud providers [61, 62, 64, 71]. The price calculation is based on the 3-year commitment plan which provides best cost savings for users. It also represents the minimum revenue achieved by offloading cloud storage stack to ARM SoC and renting those released x86 cores to user VMs.*

mands from user VMs carry guest information and LeapIO runtime will perform sanity check on Logical Block Address (LBA) and data buffer addresses, thus VMs won't be able to jump outside of the controlled queue pair view. LeapIO directly fails the IO if illegal command opcode, LBA address or data buffer address are found. Overall, we don't expose extra attacking surface compared to existing on-x86 hypervisor IO interface.

On the other hand, users want to keep their private data confidential, even from cloud providers. In LeapIO, SoC is only granted limited set of host resource access capability but not allowed to access resources beyond user VM IO scope. For instance, our runtime can only refer to host IOMMU for registered guest IO buffer addresses, but not allowed to access other non-IO guest memory regions. This makes sure that user data won't be "stolen" by the cloud provider. In addition, hardware security features such as Intel SGX [119] is already being used by cloud providers to guarantee user VM data safety, which can be deployed in conjunction with LeapIO.

6.4.2 Cost Analysis

We mainly focus on the TCO analysis related to CPUs since that's the major resource we release by offloading cloud storage stack to ARM SoCs. Moreover, CPU is the dominant resource for VM pricing. More CPUs correspond to more service density, so revenue increase proportional to CPU increase, *e.g.*, increasing the number of CPUs by 10% will bring a 10% revenue increase. LeapIO deployment is simply plugging the ARM SoCs into the server. Thus, the cost saving by LeapIO design comes from the difference between revenues brought by selling those x86 cores originally dedicated for cloud storage stack and the TCO of ARM SoCs, as shown in Figure 6.1.

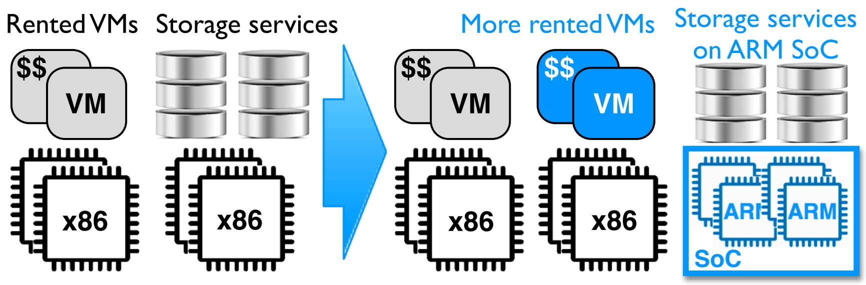


Figure 6.1: **LeapIO Goal.** *By offloading storage services to ARM SoCs, we release more host CPU resources for rented VMs.*

Revenue Analysis: Overall, as shown in Table 6.2, the yearly price of 1 vCPU is \$400 in average. Considering a modern data center server with 2 Skylake 24-core CPUs [86], LeapIO releases $2 \times 24 \times$ %10–%20 = 5–10 cores, which will convert to \$2000–\$4000 increased revenues at slightly extra TCO (from ARM SoC).

The bill-of-material cost of an ARM SoC is cheap and the power consumed is 10W, making an annual TCO of less than \$100. Considering large volume purchase will significantly lower the SoC price, more revenue gains can be achieved.

Cost Analysis: Costs include buying ARM SoC hardware and we refer to it as an inclusive measure for TCO. The revenue comes from selling services, *e.g.*, VMs, by renting cores to users. And the profit comes from the difference between revenue and cost. According to public information, Microsoft Azure has an operating margins of 28% [79]. That means for every dollar of revenue, 72% goes to costs such as TCO, and 28% is the profit. So to calculate the overall cost increase, we use the following analysis:

The estimated revenue that could be garnered from the x86 cores on one server is \$20000. And the estimated TCO for that revenue on another x86 machine will be \$20000 $\times$ 72% = \$14400. Recall that the TCO of additional ARM SoCs installed to each server is roughly \$100, thus, the estimated TCO increase is (\$14400 + \$100) / \$14400 = 0.7%. As cloud operators both oversubscribe and can't always fill their data centers with customers, the sketch of the above analysis is complete enough. We are not accounting for oversubscription and empty datacenters because that's too complicated.

Overall, the cloud provider can potentially increase \$2000 at only 0.7% extra TCO.

6.4.3 ARM Offloading Capability

As already demonstrated by a recent work [174], "Arm processor is powerful enough as the NVMeoF target," where they use a similar ARM co-processor as in our case. In our profiling experiments (not shown), we compare ARM SoC vs. x86 performance when running the traditional Linux block IO stack with different workloads (not in VMs), including microbenchmarks using FIO and macrobenchmarks using YCSB+RocksDB. our results show that overall, the performance on ARM SoC is within $2\times$ that of x86 platform.

Our storage stack tasks (as listed in Table 1 in the main submission) are not data-intensive tasks, thus ARM cores are capable to handle them quite well. Data intensive related features such as compression, encryption can be easily implemented using accelerators in the ARM SoC or the host. Finally, we would like to summarize ARM vs. x86 for clarity: For throughput parity at acceptable tail-latency increase, roughly 1.8 ARM cores can replace a single x86 core. For tail-latency parity, roughly 2.3 ARM cores are needed for replacing a single x86 core. Overall, our 8-core ARM SoC is enough to hit network line-rate as well as saturate local SSD.

Although current realSoC-LocalSSD is up to 30% slower (will be improved in our future SoC), our cost benefit analysis shows that using even $4\times$ more cores in realSoC compared to the number of cores in SoC$_{VM}$ to achieve performance parity still pays off. x86 is an overkill for polling and ARM co-processors can easily take over the burden when serving SSDs over the different transports.

6.4.4 LeapIO Technical Novelties

We quickly summarize the design properties required in deployment (P1-P6) and recap our technical contributions (T1-T4) as stated in our main submission, so we can later articulate how LeapIO differs from related works.

(P1) Fungibility: Run the same storage-services on a variety of servers (i.e. ARM, RDMA, NVMe, SR-IoV, +IOMMU are considered optional). **(P2) Transparency:** Unmodified guest VMs/applications. **(P3) Virtualizability:** Simultaneously virtualize local/remote SSDs and local/ hyper-converged/remote storage-services. **(P4) Composability:** Multiple SSDs and local/remote SW-services/HW-accelerators can be easily combined for richer virtual-drives. **(P5) Performance:** Minimize data copies between different components along with polled IO-completions. **(P6) Extensibility:** Services run in user space regardless of hardware (ARM/x86, RDMA/TCP/REST, or NVMe/SATA) with the encapsulation of all local/remote SSDs/services behind NVMe.

(T1): A new, complete set of hardware properties (see HW1-HW4 in Sec 2.2) to make ARMSoC-to-peripheral communications as efficient as x86-to-peripherals. **(T2):** A uniform address space across x86, ARM-SoC and other PCIe-devices to enable line-rate translations/data-movement. **(T3):** A portable-runtime exploiting T1 and T2 to make offloading seamless. **(T4):** Several complex services composed of local/remote SSDs/devices/services.

Existing commercial NVMeoF target adapters, such as Mellanox Bluefield [49] and Broadcom Glass Creek [65] don't satisfy P1, P5 and (partially) P3. Storage services cannot work without Bluefield or Glass Creek hardware (no P1). They also don't have T1 – the software won't saturate host-level SSDs and accelerators, and remote SSDs simultaneously at line rate because it lacks the ability to help ARM (where portable services can run) bypass the NIC when talking to other PCIe devices without taking PCIe lanes away from the NIC (compromising on P5 and P3). Moreover, our T1-T4 can improve when better hardware is present, for instance, T2 can swap its translation logic with an NVMe-SR-IOV when it is available. Further, Bluefield and Glass Creek adapters can only be configured either as an initiator or target, but not both at the same time, making it more complicated to provide rack-local storage where both local and remote drives must be virtualized simultaneously. Unlike them, the LeapIO platform enables local and remote NVMe virtualization simultaneously.

Existing in-network applications exploit SmartNICs to co-design networking applications for performance and offload application logics, e.g. FlexNIC [192], Floem [262], NICA [150]. They require application changes and software requires new hardware to work (no P1-P4, partial P6) whereas LeapIO is transparent to guest-VMs/applications/hardware.

Existing user-space IO (SPDK/DPDK) [324] solutions cannot achieve extensibility (no P6) easily in offloaded environment; they are not designed for managing peer PCIe devices from ARM (lack of T1).

Regarding the use of P2P-DMA, existing peer-to-peer DMA techniques (SPIN [123], GPUnet [204], GPUrdma [140], Morpheus [301], GPUDirect [72]) try to optimize the datapath between SSD and GPU/FPGA/NIC, where application data is DMA'ed between PCIe device-buffers. However, P5 at line-rate cannot be fully realized without T1 and T2. Moreover, the aforementioned works mainly depend on existing system/library support for P2P-DMA for performance, LeapIO presents an ARM SoC centric design for general P2P DMA among multiple devices.

Existing OS/storage/network offloading systems (Solros [247], Caribou [172], GPUfs [291], NICA [150]) will fragment code bases – one code base for legacy servers without accelerators and one code base for servers with accelerators (no P1-P2). LeapIO is a principled approach to tackling this problem where software components can be swapped with hardware components while the storage-logic can freely move between x86 and ARM.

Regarding address translation support, our goal was to provide an offload-ready storage stack that can opportunistically leverage hardware features when available rather than depend heavily on hardware. The reason for this was to ensure fungibility of servers in our data center – a significant portion of which do not have cutting edge hardware that is just about taking off. Developing services that take a hard dependency on hardware then automatically fragments the servers where we cannot offer all virtual drive types on all servers. For this purpose, we leverage hardware when available. And one such hardware is that of virtual NVMe and address translation. The part of the paper concerning address translation is necessary for all of our server fleet to be able to run all of our storage service innovations.

6.4.5 Deployment

Cloud infrastructure management simplicity is an important aspect for data center scale solutions. LeapIO is a principled design to satisfy deployment requirements with a transparent software model and plug-in hardware platform.

As pointed out in recent work, Harmony [122], existing offloading solutions "introduce a semantic gap in existing, otherwise silo-ed, management techniques and complicates infrastructure managment." This aligns well with our observations. While Harmony [122] focuses on in-network compute domain, LeapIO presents an end-to-end solution simplying the deployment and new service development efforts.

On the other hand, cloud storage user requirements are evolving at the speed of months [146], thus, how fast the offloading framework can react to new needs is of great importance. LeapIO enables fast features/services development in a user-space x86-like environment, by abstracting out the low-level hardware capabilities and differences between x86 and ARM. For instance, our RAID-0/1 services are done in 2 developer days and block caching services in 7 developer days. Compared to FPGA-based platforms, LeapIO's choice of utilizing ARM for storage service offloading brings us great flexibility.

Chapter 7

Related Work

7.1 Storage Performance

• **Storage tails:** A growing number of work has investigated many root causes of storage latency tail, including multi tenancy [170, 199, 244, 297], maintenance jobs [104, 127, 239], inefficient policies [168, 238, 322], device cleaning/garbage collection [99, 141, 170, 199, 318], hardware variability [165], and bursty workload behaviors [333]. MittOS does not eliminate these root causes but rather expose the implied busyness to applications.

• **Storage tail tolerance:** Throughout the paper, we already discussed existing solutions such as snitching/adaptivity [8, 295], cloning at various different levels [105, 304, 313], hedged and tied requests [141]. A growing number of works recently investigated sources of storage-level tail latencies, including background jobs [104], file system allocation policies [168], block-level I/O schedulers [322], and disk/SSD hardware-level defects [147, 159, 165]. An earlier work addresses load-induced tail latencies with RAID parity [124]. Our work specifically addresses GC-induced tail latencies.

• **Performance isolation (QoS):** A key to reduce performance variability is performance isolation, such as isolation of CPU [329], IO throughput [157, 289, 297], buffer cache [244], and end-to-end resources [108, 136]. 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 (§3.1.3).

• **OS transparency:** MittOS in spirit is similar to other works that advocate more information exposure to applications [110, 225] and first-class supports for interactive applications [323]. MittOS provides busyness transparency by slightly modifying the user-kernel interfaces (mainly for passing deadlines and returning EBUSY).

• **GC-impact reduction:** Our work is about eliminating GC impacts, while many other existing works are about reducing GC impacts. There are two main reduction approaches: *isolation* and *optimization*, both with drawbacks. First, isolation (*e.g.*, OPS isolation [199]) only isolates a tenant (*e.g.*, sequential) from another one (*e.g.*, random-write). It does not help a tenant with both random-write and sequential workloads on the same dataset. OPS isolation must differentiate users while TTFLASH is user-agnostic. Second, GC optimization, which can be achieved by better page layout management (*e.g.*, value locality [163], log-structured [138, 218, 248]) only helps in reducing GC period but does not eliminate blocked I/Os.

• **GC-impact elimination:** We are only aware of a handful of works that attempt to eliminate GC impact, which fall into two categories: without or with redundancy. Without redundancy, one can eliminate GC impact by *preemption* [133, 220, 309]. With redundancy, one must depend on RAIN. To the best of our knowledge, our work is the first one that leverages SSD internal redundancy to eliminate GC tail latencies. There are other works that leverage redundancy in flash array (described later below).

• **RAIN:** SSD's internal parity-based redundancy (RAIN) has become a reliability standard. Some companies reveal such usage but unfortunately without topology details [13, 21]. In literature, we are aware of only four major ones: eSAP [200], PPC[171], FRA [223] and LARS [221]. These efforts, however, mainly concern about write optimization and wear leveling in RAIN but do not leverage RAIN to eliminate GC tail latencies.

• **Flash array:** TTFLASH works within a single SSD. In the context of SSD array, we are aware of two published techniques on GC tolerance: Flash on Rails [292] and Harmonia [209]. Flash on Rails [292] eliminates read blocking (read-write contention) with a ring of multiple drives where 1–2 drives are used for write logging and the other drives are used for reads. The major drawback is that read/write I/Os cannot utilize the aggregate bandwidth of the array. In Harmonia [209], the host OS controls all the SSDs to perform GC at the same time (*i.e.*, it is better that all SSDs are "unavailable" at the same time, but then provide stable performance afterwards), which requires more complex host-SSD communication.

7.2 Storge Efficiency

We reviewed the growing literature in IO accelerator and virtual storage and did not find a single technology that meets our needs. Below we summarize our findings as laid out in Table 7.1, specifically in the context of the six dimensions of support (represented by the last six columns).

First, many works highlight the need for IO accelerators (the "**Acc**" column), *e.g.*, with ASIC, FPGA, GPU, Smart SSDs, and custom NICs. In our case, the accelerator is a custom ARM-based SoC (§4.1.1) for reducing the storage tax.

		Acc	**Uni**	**Port**	**vNVMe**	**Usr**	**sVirt**
A	ActiveDisks [98]	√					
	ActiveFlash [299]	√					
	iSSD [135]	√					
	ActStorage [268]	√					
	Biscuit [156]	√			√	√	
	BlueDBM [183]	√					
	Caribou [172]	√					
	FAWN [107]					√	
	Ibex [308]	√				√	
	IDISKS [194]	√					
	FlashShare [328]	√			√		
	GraFBoost [184]	√					
	HyperLoop [197]	√				√	
	INSIDER [271]	√					
	LightStore [137]	√				√	
	QsmartSSD [145]	√				√	
	SmartSSD [191]	√					
	SR-IOV [90]	√	√		√		
	Summarizer [212]	√			√	√	
	Willow [279]					√	
	YourSQL [181]	√				√	
B	AMF [222]						
	Decibel [251]				√	√	
	DiskCryptNet [243]		√		√	√	
	FlashBlox [170]				√	√	
	IOFlow [297]					√	

		Acc	Uni	Port	vNVMe	Usr	sVirt
	KAML [179]	√			√		
	LightNVM [126]				√	√	
	Malacology [280]		√			√	
	Moneta [130]					√	
	NOVA [315]						
	NVMeoF [38]			√	√	√	
	Orion [319]					√	
	SDF [257]					√	
	SPDK [324]				√	√	
	Strata [216]						
C	Arrakis [259]		√			√	
	BlueCache [316]	√				√	
	CIDR [103]				√	√	
	CORFU [113]					√	
	FlashNet [300]		√				
	FlatFlash [97]		√		√		
	Gordon [131]	√					
	Helios [254]	√	√				
	IX [120]		√			√	
	LegoOS [282]		√				
	M3 [111]	√	√			√	
	NBA [203]			√			
	QuickSAN [132]	√				√	
	Reflex [211]				√	√	
	Solros [247]	√	√				
	Transkernel [161]	√					
	VRIO [215]					√	
D	AccelNet [151]	√					
	Bluefield [49]	√	√		√	√	
	Catapult [129]	√					
	E3 [237]	√				√	
	Eris [230]	√					
	FlexNIC [192]	√				√	
	Floem [262]	√				√	
	IncBricks [236]	√					
	iPipe [235]	√				√	
	KV-Direct [226]	√					
	NetCache [177]	√				√	
	NetChain [176]	√				√	
	NICA [150]	√					
	SwitchKV [231]	√				√	
	UNO [217]	√		√			
E	ActivePointers [281]		√			√	
	Dandelion [270]		√				
	DUA [288]	√	√				

	Acc	Uni	Port	vNVMe	Usr	sVirt
ExtraV [219]	√					
GAIA [128]		√			√	
GENESYS [303]	√	√				
GPUfs [291]		√			√	
GPUnet [204]					√	
GPUrdma [140]					√	
HeteroISA [116]			√		√	
HEXO [255]	√		√			
Morpheus [301]		√		√		
NDPos [115]	√	√			√	
OmniX [290]	√	√				
Popcorn [117]	√		√			
PortPerf [261]	√		√			
pTask [269]	√					
LeapIO	√	√	√	√	√	√

Table 7.1: **Related Work (§7.2):** *The columns (dimensions of support) are as follow.* **Acc***: Hardware acceleration;* **Uni***: Unified address space;* **Port***: Portability/fungibility;* **vNVMe***: Virtual NVMe abstraction;* **Usr***: User-space/software-defined storage functions;* **sVirt***: Simultaneous local+remote NVMe virtualization. The row (related work) categories are:* **A***. Storage accelartion/offloading;* **B***. Software-defined storage;* **C***. Disaggregated/split systems;* **D***. Programmable NICs; and* **E***. Heterogeneous system designs. We revived in detail a total of 85 related papers.*

When using accelerators, it is desirable to support a unified address space (**Uni**) to reduce data movement. While most work in this space focus on unifying two address spaces (*e.g.*, host-GPU, host-coprocessor, or host-SSD spaces), we have to unify three address spaces (guest-VM/host/SoC) with 2-level address translations (§4.1.4).

One uniqueness of our work is addressing portability/fungibility (**Port**) where LeapIO runtime and arbitrary storage functions can run on either the host x86 or ARM SoC, hence our SoC deployment can be treated as an acceleration opportunity rather than a necessity. In most of other works, only specifically provided functions (*e.g.*, caching, replication, or consensus protocol) are offloadable.

We chose virtual NVMe (**vNVMe**) as the core abstraction such that we can establish an end-to-end storage communication from guest VMs to the remote backend SSDs through many IO layers/functions that speak the same NVMe language (§4.1.3). With this for example, LeapIO is the first platform that enables virtual SSD channels (backed by OpenChannel SSDs) to be composable for guest VMs (§4.3.3).

All of the above allow us to support user-space/software-defined storage functions (**Usr**) just like many other works. In LeapIO, user-level storage functions can be agnostic to the underlying hardware (x86/SoC, local/remote storage). With this for example, LeapIO is the first to support user-space NVMeoF with stable performance (§4.3.1).

Finally, to support "spillover drive" (Table 1.1), LeapIO is the first system that supports *simultaneous* local and remote NVMe virtualization (the **sVirt** column). Related work like Bluefield with hardware NVMe emulation [49, 74] cannot achieve this because it can only support running in either local or remote virtualization mode (*e.g.*, initiator or target), but not both simultaneously in composable ways.

Overall, our unique contribution is combining these six dimensions of support to address our deployment goals. We also would like to emphasize that our work is orthogonal to other works. For example, in the context of GPU/FPGA offloading, application computations can be offloaded there, but when IOs are made, the storage functions are offloaded to our ARM SoC. In terms of virtual NIC, its network QoS capability can benefit remote storage QoS. In terms of framework/language level support for in-NIC/Storage offloading, it can be co-located with LeapIO to accelerate server applications. In terms of accelerator level support for OS services or OS support for accelerators, LeapIO can benefit from such designs for more general purpose application offloading.

Below we compare LeapIO with related works in detail.

• **Storage Acceleration/offloading:** The closest to LeapIO is the work that proposes offloading storage computations and applications to programmable SSDs. LightStore [137], KAML [179], LSM-SSD [306] and KV-SSD [89] offload key-value management in the storage controller, however, such solutions are tightly coupled with key-value interface design and limit their flexibility and extensibility to support various types of storage services.

IDISKS [194], ActiveDisks [98], ActiveFlash [299], iSSD [135], Willow [279], Summarizer [212], SmartSSD [145, 191] and Biscuit [156] provide programming framework support to exploit free cycles in storage controller for query processing, streaming processing, replication, and general storage applications offloading. These works require non-trivial application changes and performance benefits can be bounded by resource limitation due to contention with FTL tasks. In contrast, LeapIO provides VM storage transparently with close-to bare-metal performance. LeapIO also differs from these works fundamentally in design principles to provide offload-ready storage stack and simultaneous local and remote NVMe virtualization support.

While technologies such as SR-IOV (for local SSD virtualization [90]) and NVMeoF (for remote SSD access [38]) can be combined to provide local/remote virtual SSDs, they are *not* composable to provide richer services. SR-IOV bypasses the host/hypervisor and allows VM direct control over storage devices, thus disabling storage services commonly implemented at the hypervisor level. LeapIO not only reaps the benefits from today's SR-IOV enabled SSD designs and NVMeoF, but also combining them without taking away x86 resources along with seamless service integrations.

INSIDER [271], GraFBoost [184] and Ibex [308] explore storage accelerations using FPGAs. While FPGAs can provide better performance than ARM-based solutions, FPGA programming is difficult and time-consuming, making it hard to satisfy the cloud storage development and deployment speed. LeapIO is designed to tolerant hardware heterogeneity and enables a uniform user-space software development environment.

BlueDBM [183], Caribou [172], Corfu [113], and FAWN [107] present a clustered design to accelerate storage applications. While the hardware architecture and the design of LeapIO make it conducive for such offloading, our main aim is to free the x86 cores so that they run customer VMs while first party complex data paths run on ARM. Likewise, compared to approaches such as Solros [247] where additional x86 cores are used for processing IO, LeapIO focuses on bringing a degree of coherence and address space uniformity across host x86 cores, cheaper ARM processors, and RDMA NICs to help storage services achieve complex tasks.

Moreover, LeapIO is different from other works in terms of providing services with a uniform interface of queues even when the hardware lacks those features, thus ensuring portability of services (that always assume future-looking interfaces such as queue pairs) across servers that lack RDMA NICs or NVMe SSDs or virtual-NVMe IP. Moreover, we also help data centers reduce cost by implementing virtual NVMe SSDs in host or ARM memory without requiring complex SSD firmware/logic that can support SR-IOV.

• **Software-defined storage:** Storage virtualization is a major component in software-defined storage. The focus in this area has mainly been to reduce guest-host communications, avoid context switches as much as possible, and reduce software overhead [42, 102, 125, 166, 258, 321]. LeapIO minimizes such performance overhead via direct queue pair mappings across hardware boundaries and cheap polling utilizing ARM co-processors.

SPDK [324] is a user-space polling based storage platform, burning precious CPU resources for performance. While it aims to provide better storage performance than traditional kernel-level block IO stack, it breaks application compatibility and only limited applications can run on top of it whereas LeapIO works in a transparent manner. SPDK cannot achieve extensibility easily in offloaded environment as they are not designed for managing peer PCIe devices from ARM. IOFlow [297] and Malacology [280] present complex software defined cloud storage stack design on traditional x86 centric server systems, and LeapIO aims to achieve similar features by offloading them on cost-efficient ARM SoCs. LeapIO builds on these ideas and extends them to not only access remote SSDs and complex storage services *without any host intervention*, but also compose them in rich ways with minimum software overhead (such as interupts, context switches, multiple data copies, etc.).

Other systems focus on isolation guarantees [170, 195, 199, 251, 287]. AMF [222], FlashBlox [170], and LightNVM [126] demonstrate the performance benefits of direct user management of NAND Flash by moving FTL from SSD controller to the host. LeapIO extends similar ideas by running such management on ARM SoCs and exposing virtualized channels directly to VMs, thus freeing x86 for more important VM tasks as well as guaranteeing performance QoS.

Recent software proposals and emerging hardware [51, 54] allow efficient access to remote SSDs [38, 210, 211, 224, 251, 300]. These are point technologies purpose built for accessing SSDs over a particular networking stack (e.g. RDMA). LeapIO strives to free the cloud providers to choose any networking protocol in any stack (kernel/user space) while providing the ability to run complex functions in the data path. The need for flexibility and transparency has motivated us to implement NVMe over TCP, RDMA, REST, etc. with the ability to compose multiple transports into one rich and complex data path without x86 involvement.

• **Disaggregated/split systems:** Several distributed/disaggregated system innovations [107, 113, 114, 132] have been proposed to leverage cutting edge storage and networking hardware. LegoOS [282] introduces a disaggregated data center operating system design based on fast networks. CORFU [113] and Tango [114] manage multiple Flash devices as a single log to simplify distributed application development. FlashNet [300], Gordon [131], and QuickSAN [132] present a software-hardware co-designed system for integrating storage and networking stack for performance. These disaggregated designs are orthogonal to LeapIO designs. LeapIO provides a platform for cloud providers to implement such techniques without taxing the valuable x86 CPUs. LeapIO brings the ability to access a wide variety of accelerators from the ARM SoC, making it a powerful platform for implementing first party distributed storage services.

Split systems like Arrakis [259] and IX [120] advocate to treat OS as control plane while utilizing hardware support and kernel-bypassing for fast dataplanes. HeteroISA [116], Helios [254], and M3 [111] provide software abstractions for heterogeneous systems with types of various co-processors (GPUs, ARMs, FPGAs, etc.) and enable computation migration among them. LeapIO builds on these ideas to provide a fast virtual storage dataplane and unified address space to abstract out x86 and ARM co-processor differences and enable agile service development which can move freely between x86 and ARM SoC.

• **Programmable NICs and Heterogeneous system designs:** Utilizing programmable NICs for application logic offloading has been explored in different dimensions. Existing commercial SmartNIC platforms, like Bluefield [49] services, cannot work without Bluefield-hardware (lack of fungibility). It's also challenging to saturate host-level SSDs and remote SSDs simultaneously at line rate because it lacks the ability to help ARM (where portable services can run) bypass the NIC when talking to other PCIe devices without taking PCIe lanes away from the NIC. LeapIO can opportunistically utilize the hardware acceleration capability and achieve max performance for virtual storage.

AccelNet [151] and Catapult [129] utilize FPGA based SmartNICs to offload Microsoft data center networking functions. Various other types of applications are also explored, for example, FlexNIC [192] allows user to install rules into DMA engine for efficient packet processing, IncBricks [236] and NetCache [177] implement a key-value caching layer in the programmable network dataplane, others target offloading KV stores like SwitchKV [231] and KV-Direct [226], microservices like E3 [237]. Floem [262] and NICA [150] propose language/OS level framework support to simplify writing offloaded applications. These in-network applications exploit SmartNICs to co-design applications for performance and offload logic, *e.g.* FlexNIC [192], Floem [262], NICA [150]. They require application changes and software requires new hardware to work whereas LeapIO is transparent to guest-VMs/applications/hardware.

Overall speaking, these in-NIC computing solutions can co-exist with LeapIO based storage offloading framework. While LeapIO focuses on providing transparent NVMe storage to VMs, the VM-level networking applications can still benefit from in-NIC offloading designs. LeapIO differs from this existing work by designing and implementing a runtime that allows services to be implemented in a hardware-independent manner. That is, high-level services such as rack-local storage, multi-block atomic writes, tiered storage etc (are written once) will work regardless of server configuration – SmartNIC or not, RDMA or not, SR-IOV or not. Every required data path component such as address translation, network protocol used, encryption etc. are modularized such that the system can swap the underlying implementation depending on server configuration. This is very important for us to ensure that all the servers in the data center run a uniform storage service code to minimize development effort. We also want to introduce co-designed hardware such that it will free up the most number of x86 cores depending on our workloads and not be dictated by hardware designed by third parties.

Beside SSD and NIC based accelerators, past work on GPUs focus on easing GPU access to OS services, such as GPUfs [291] for storage, GPUnet [204] and GPUrdma [140] for networking accesses to GPUs, ActivePointers [281] for shared memory, and GAIA [128] for shared page cache. Considering the rapid growth of a diverse set of accelerators, there is an increasing interest in adding OS support to such new accelerators like NDPos [115], OmniX [290], Dandelion [270] and pTask [269]. LeapIO shares similar goals with these systems, and propose a set of hardware properties and software techniques to achieve shared address space and an ARM co-processor centric storage offloading framework.

CHAPTER 8

FUTURE WORK

The increasing adoption of specialized hardware poses tremendous challenges to application development due to largely missing system-level support. An urgent problem is: **How should we structure system software to allow applications to effectively exploit a diverse collection of specialized storage devices?** Compute accelerators (*e.g.*, GPUs, FPGAs) have spawned much effort to enable their use for a wide range of applications, such as ML, video processing, and even some traditional database operations. Similarly, a diversity of specialized storage components has been proposed in recent years. Unfortunately, they are held back by aged and inflexible storage system interfaces and usage models.

Storage continues to rely on the rigid decades-old device interface that only allows `read/write`-block, and even the file system (FS)-level is often limited to `open()/read()/write()/close()` semantics. With the emergence of many new storage component characteristics, such as byte-addressability, compute offload, direct transfer to/from accelerators, it is becoming extremely difficult to efficiently utilize these features through existing interfaces.

In more detail, there are many changes happening in today's computing systems and they bring many opportunities to systems research. Specifically, on the application side, emerging application models/paradigms, such as data analytics, ML, and serverless computing are being mainstream data center workhorse. On the hardware side, we see numerous new hardware springs up due to the desire to mine intelligence from data faster, such as accelerators (GPU, FPGAs) for fast computation on data, blazingly fast IO devices (NVMe, NVM, RDMA NIC) for fast data movement and smart devices (SmartSSD, SmartNIC) for near-data acceleration. Thus, designing proper systems level support for emerging hardware is very important to bring hardware-promised performance benefits to applications. To tackle this problem, we believe a data-centric design to revisit the systems-software/hardware boundaries and responsibility divisions is needed.

System Support for New Storage Technologies: Storage controllers (SSDs, Intel Optane NVM) are the frontline to deliver storage performance. These technologies are quickly evolving with many new features built on top, such as Zoned Namespace (ZNS) for better host control to storage device management, Key-Value (KV-SSD) interface for higher data throughput, IO Determinism (IOD) for performance predictability. However, all these innovations cannot be easily reaped by applications without proper systems level support or abstraction. The question we should look at is how to design OS-level indirection layers and file system support to hide the complexity of these device-level features. This will enable storage applications to take advantage of them transparently. Another direction is to revisit existing applications designs and re-architect them to exploit hardware benefits directly bypassing the OS.

Storage Specialization for Modern Application Paradigms: While we have seen a lot of advancement in accelerated data processing and frameworks for programming modern data intensive applications, data management is still using the traditional general IO model and oblivious of application level data access patterns. The question we want to ask here is how storage systems should evolve to better support modern applications for performance? Since these applications have become the major CPU cycle consumers in modern data centers, it's time for us to specialize and optimize the storage stack for them. One approach is to study the data access patterns during different phases of the applications (*e.g.*, data shuffling in analytics, ML data processing pipelines, and remote storage access in serverless) and design specific optimizations such as caching, prefetching policies and better programming models to alleviate the IO bottleneck.

Near Data Computing: Emerging smart SSDs and NICs with computation capability (*e.g.*, ARM, FPGA) attached to them are attracting more attention these days. They provide a "look-aside" or "on-path" computing subsystem to process data without requiring host level resources, thus helping reduce the amount of data movement and provide potential high performance. However, with so many different flavors of "smart" devices available, the lack of uniform software platform support makes programming such devices challenging as it requires a lot of application level expertise to understand how to offload certain logics and handle low level communication efficiency problems. Worse, with today's CPU centric design, file accesses from smart devices are inefficient and have to be routed through host CPU. In this direction, it is extremely important to design high-level interfaces for applications to easily push computation and file access semantics to the near-data processors without major system architecture changes. This will potentially ease programmability while achieving efficiency and performance.

Predictable Prediction Serving Systems: With ML models being increasingly used as major workhorses for a wide range of applications, it is extremely important to achieve predictable inference latencies for modern prediction serving systems. Prediction serving systems are typically deployed across a cluster of machines, thus they are prone to slowdowns and failures and similarly the tail latency problem exists. While recent works such as [213] start looking at "parity models" to enhance reliability of the prediction serving systems, we argue that the design principles proposed in this dissertation to cut storage-related tails can also be applied in the computing scenario to guarantee low and stable prediction serving. For instance, we can "fast-fail" slow computations and utilize the model redundancy to quickly "fail-over" to faster computing nodes for predictable prediction latencies. More designs can be explored in this direction.

CHAPTER 9

CONCLUSION

This dissertation has shown how to build cloud storage stack with predictable latencies and cost-efficiency as well as enabling new research platform designs to satisfy rising full-stack research needs. First, to tackle the notorious tail latency problem in modern Flash-based storage stack, we adopt a holistic approach to co-design software/hardware with determinitic IO latencies. We build three systems across different layers of the storage stack for an end-to-end tail-tolerant storage stack. Second, to reduce the heavy data center storage tax, we design LeapIO to offload modern storage services and address a wide range of real data center deployment challenges. Third, to enable full-stack software/hardware research, we design a new NVMe SSD platform on top of QEMU. Collectively, we build three major systems to improve storage performance, cost-efficiency and usability, which we summarize below:

TEAFA is a OS/SSD co-designed AFA system to deliver predictable IO latencies. TEAFA adopts a simple yet powerful storage interface change to communicate device-side background activities, and informs the host to react timely. TEAFA proactively reconstructs read IO data using the redundant data widely avaialbe in modern storage systems and introduces a simple time window based scheduling mechanism to guarantee low tail latencies even for high percentiles (*e.g.*, p99.99). Extensive evaluation results demonstrate TEAFA's effectiveness when compared to state-of-the-art techniques and deliver up to orders of magnitude latency improvement over baseline.

LeapIO is a Hypervisor/ARM-SoC co-designed cloud storage stack for efficient storage service offloading and agile development in user space. LeapIO is designed to satisfy real-deployment requirements such as fungibility, virtualizabilty, performance and compoasability. LeapIO employs a unified address space across x86, ARM, and other IO devices to minimize data copies. We demonstrate that LeapIO is able to achieve close-to bare metal performance and improve resource utilization. LeapIO is being deployed in Microsoft data centers as of the writing which proves the success of its design.

FEMU is QEMU/software based NVMe SSD emulator which brings the “CASE” benefits: cheap (open-sourced and successfully used in several projects which appeared in top-tier OS and storage conferences), accurate (relateively low per-IO latency emulation error rate (*i.e.*, 11% in average) at microsecond level), scalable (being able to support high level channel/chip parallelism), and extensible (drop-in replacement as OpenChannel- or regular Blackbox- SSDs).

References

[1] https://github.com/ucare-uchicago/MittSSD.

[2] https://github.com/ucare-uchicago/tinyTailFlash.

[3] https://github.com/huaicheng/LeapIO.

[4] https://github.com/huaicheng/TeaFA.

[5] https://github.com/ucare-uchicago/femu.

[6] Alibaba Deploys Alibaba Open Channel SSD for Next Generation Data Centers. https://www.alibabacloud.com/blog/alibaba-deploys-alibaba-open-channel-ssd-for-next-generation-data-centers_593802.

[7] All-Flash Array Market worth 17.8 billion by 2023. https://www.marketsandmarkets.com/PressReleases/all-flash-array.asp.

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

[9] DiskSim. http://www.pdl.cmu.edu/DiskSim/.

[10] Filebench. http://filebench.sourceforge.net/wiki/index.php/Main_Page.

[11] Global All-Flash Array (AFA) Market Outlook to 2023 - Increasing Deployment of AFA Storage in AI and ML Applications Presents Lucrative Opportunities. https://www.globenewswire.com/news-release/2019/03/14/1753148/0/en/Global-All-Flash-Array-AFA-Market-Outlook-to-2023-Increasing-Deployment-of-AFA-Storage-in-AI-and-ML-Applications-Presents-Lucrative-Opportunities.html.

[12] Memory Management in the Java HotSpot Virtual Machine. http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf.

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

[14] Microsoft Rolls Out SSD-Backed Azure Premium Cloud Storage. http://www.eweek.com/cloud/microsoft-rolls-out-ssd-backed-azure-premium-cloud-storage.html.

[15] New SSD-Backed Elastic Block Storage. https://aws.amazon.com/blogs/aws/new-ssd-backed-elastic-block-storage/.

[16] Physical Page, Logical Page, and Codeword Correspondence. http://www.google.com/patents/US20130246891.

[17] Report: SSD market doubles, optical drive shipment rapidly down. http://www.myce.com/news/report-ssd-market-doubles-optical-drive-shipment-rapidly-down-70415/.

[18] SMRFS-EXT4. https://github.com/Seagate/SMR_FS-EXT4.

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

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

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

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

[23] VIOLIN Flash Fabric Architecture. https://www.violinsystems.com/wp-content/uploads/resources/Violin-Whitepaper-Flash-Fabric-Architecture.pdf.

[24] 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.

[25] Data Set Management Commands Proposal for ATA8-ACS2. http://www.t13.org/Documents/UploadedDocuments/docs2008/e07154r6-Data_Set_Management_Proposal_for_ATA-ACS2.pdf, 2007.

[26] Using COPYBACK Operations to Maintain Data Integrity in NAND Flash Devices. https://www.micron.com/~/media/documents/products/technical-note/nand-flash/tn2941_idm_copyback.pdf, 2008.

[27] SATA-IO Releases Revision 3.1 Specification. https://sata-io.org/system/files/member-downloads/SATA-IORevision31_PRfinal_0.pdf, 2011.

[28] 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.

[29] NVM Express Base Specification 1.0. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_0e.pdf, 2013.

[30] Raid5: Make Stripe Handling Multi-threading. https://lwn.net/Articles/563142/, 2013.

[31] MZHPV128HDGM (SM951) 128 GB PCIe Gen3 8Gb/s x4 M.2. http://www.samsung.com/semiconductor/products/flash-storage/client-ssd/MZHPV128HDGM, 2014.

[32] Towards Multi-threaded Device Emulation in QEMU. KVM Forum, 2014.

[33] Improving the QEMU Event Loop. KVM Forum, 2015.

[34] Storage Latency in Flash Arrays. https://www.violinsystems.com/wp-content/uploads/Storage-Mojo-WP-storage-latency.pdf, 2015.

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

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

[37] FlashBlade: The Cloud-Scale All-Flash Data Platform. https://people.eecs.berkeley.edu/~xtan/flashblade_berkeley_talk.pdf, 2016.

[38] NVM Express over Fabrics Revision 1.0. https://nvmexpress.org/wp-content/uploads/NVMe_over_Fabrics_1_0_Gold_20160605-1.pdf, 2016.

[39] OpenChannel Solid State Drives NVMe Specification (Revision 1.2). http://lightnvm.io/docs/Open-ChannelSSDInterfaceSpecification12-final.pdf, 2016.

[40] Top NoSQL Database Engines. http://www.kdnuggets.com/2016/06/top-nosql-database-engines.html, 2016.

[41] DFC Open Source Community. https://github.com/DFC-OpenSource, 2017.

[42] I/O Latency Optimization with Polling. https://events.static.linuxfound.org/sites/events/files/slides/lemoal-nvme-polling-vault-2017-final_0.pdf, 2017.

[43] LightNVM's QEMU. https://github.com/OpenChannelSSD/qemu-nvme, 2017.

[44] NVM Express Base Specification 1.3. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3e.pdf, 2017.

[45] Open-Channel Solid State Drives. http://lightnvm.io, 2017.

[46] The OpenSSD Project. http://openssd.io, 2017.

[47] Violin Memory. All Flash Array Architecture, 2017.

[48] Alibaba: Using SPDK in Production. https://ci.spdk.io/download/events/2018-summit/day1_08_ShengMingAlibaba.pdf, 2018.

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

[50] Flashtec NVRAM Drives. https://www.microsemi.com/product-directory/storage-boards/3690-flashtec-nvram-drives, 2018.

[51] HowTo Configure NVMe over Fabrics (NVMe-oF) Target Offload. https://community.mellanox.com/s/article/howto-configure-nvme-over-fabrics--nvme-of--target-offload, 2018.

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

[53] Open-Channel Solid State Drives NVMe Specification (Revision 2.0). http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf, 2018.

[54] p2pmem: Enabling PCIe Peer-2-Peer in Linux. https://www.snia.org/sites/default/files/SDC/2017/presentations/Solid_State_Stor_NVM_PM_NVDIMM/Bates_Stephen_p2pmem_Enabling_PCIe_Peer-2-Peer_in_Linux.pdf, 2018.

[55] Project Denali to Define Flexible SSDs for Cloud-scale Applications. https://azure.microsoft.com/en-us/blog/project-denali-to-define-flexible-ssds-for-cloud-scale-applications/, 2018.

[56] RAIL: Predictable, Low Tail Latency for NVMe Storage. https://platformlab.stanford.edu/Presentations/Litz.pdf, 2018.

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

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

[59] Accelerating NVMe I/Os in Virtual Machines via SPDK vhost. https://www.lfasiallc.com/wp-content/uploads/2017/11/Accelerating-NVMe-I_Os-in-Virtual-Machine-via-SPDK-vhost_Ziye-Yang-_-Changpeng-Liu.pdf, 2019.

[60] Additive Increase/Multiplicative Decrease. https://en.wikipedia.org/wiki/Additive_increase/multiplicative_decrease, 2019.

[61] Alibaba Virtual Machine Pricing. https://www.alibabacloud.com/pricing-calculator#/add/acm-fbbcfda8-d88e-4a19-b917-ee2fa7f14abe/vm_intl/vm_intl, 2019.

[62] AWS EC2 Virtual Machine Pricing. https://awstcocalculator.com/, 2019.

[63] Azure Storage Client Library for C++. https://github.com/Azure/azure-storage-cpp.git, 2019.

[64] Azure Virtual Machine Pricing. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/ubuntu-advantage-essential/, 2019.

[65] Broadcom Announces Availability of Industry's First Universal NVMe Storage Adapter for Bare Metal and Virtualized Servers. https://www.globenewswire.com/news-release/2019/08/06/1897739/0/en/Broadcom-Announces-Availability-of-Industry-s-First-Universal-NVMe-Storage-Adapter-for-Bare-Metal-and-Virtualized-Servers.html, 2019.

[66] Cloud Storage Market worth 88.91 Billion USD by 2022. https://www.marketsandmarkets.com/PressReleases/cloud-storage.asp, 2019.

[67] Dell EMC VMAX 950F All-Flash Storage. https://shop.dellemc.com/en-us/Product-Family/EMC-VMAX-Products/Dell-EMC-VMAX-950F-All-Flash-Storage/p/Dell-EMC-VMAX-950F-All-Flash-Storage?PID=EMC_SRS-VMAX-36FA_SPLSH, 2019.

[68] Emulab D430s. https://gitlab.flux.utah.edu/emulab/emulab-devel/wikis/Utah-Cluster/d430s, 2019.

[69] Flexible I/O Tester. https://github.com/axboe/fio.git, 2019.

[70] Full Virtualization. https://en.wikipedia.org/wiki/Full_virtualization, 2019.

[71] GCP Virtual Machine Pricing. https://cloud.google.com/compute/vm-instance-pricing, 2019.

[72] GPUDirect Storage: A Direct Path Between Storage and GPU Memory. https://devblogs.nvidia.com/gpudirect-storage, 2019.

[73] HPE 3PAR StorageServ Storage. https://www.hpe.com/us/en/storage/3par.html, 2019.

[74] In-Hardware Storage Virtualization - NVMe SNAP Revolutionizes Data Center Storage. http://www.mellanox.com/related-docs/solutions/SB_Mellanox_NVMe_SNAP.pdf, 2019.

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

[76] Linux DMA From User Space. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842418/Linux+DMA+From+User+Space, 2019.

[77] Micron 9100 U.2 and HHHL NVMe PCIe SSDs. https://www.micron.com/-/media/client/global/documents/products/data-sheet/ssd/9100_hhhl_u_2_pcie_ssd.pdf, 2019.

[78] Microsoft Azure Storage C++ REST SDK. https://github.com/Microsoft/cpprestsdk.git, 2019.

[79] Microsoft Earnings Release FY19 Q1. https://www.microsoft.com/en-us/Investor/earnings/FY-2019-Q1/intelligent-cloud-performance, 2019.

[80] NetApp AFF A-Series All Flash Arrays. https://www.netapp.com/us/products/storage-systems/all-flash-array/aff-a-series.aspx?cid=pio&ref_source=pio, 2019.

[81] New NVMe Specification Defines Zoned Namespaces (ZNS) as Go-To Industry Technology.

https://nvmexpress.org/new-nvmetm-specification-defines-zoned-namespaces-zns-as-go-to-industry-technology/, 2019.

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

[83] NVM Express Base Specification 1.4. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf, 2019.

[84] NVMe 1.4 Specification Published: Further Optimizing Performance and Reliability. https://www.anandtech.com/show/14543/nvme-14-specification-published/2, 2019.

[85] NVMe/TCP Transport Binding specification. https://nvmexpress.org/welcome-nvme-tcp-to-the-nvme-of-family-of-transports, 2019.

[86] Open Computer Project - Project Olympus. https://www.opencompute.org/wiki/Server/ProjectOlympus, 2019.

[87] RAID5 cache. https://www.kernel.org/doc/Documentation/md/raid5-cache.txt, 2019.

[88] RocksDB - A Persistent Key-Value Store for Fast Storage Environments. https://rocksdb.org, 2019.

[89] Samsung Key Value SSD Enables High Performance Scaling. https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf, 2019.

[90] Single-Root Input/Output Virtualization. http://www.pcisig.com/specifications/iov/single_root, 2019.

[91] SNIA I/O Trace Data Files. http://iotta.snia.org/traces, 2019.

[92] Solid State Storage (SSS) Performance Test Specification (PTS). https://www.snia.org/tech_activities/standards/curr_standards/pts, 2019.

[93] SPDK Fails to Come Up After Long FIO Run. https://github.com/spdk/spdk/issues/691, 2019.

[94] SPDK NVMf Target Crashed While Running File System. https://github.com/spdk/spdk/issues/763, 2019.

[95] SPDK Performance Very Slow. https://github.com/spdk/spdk/issues/731, 2019.

[96] User space mappable DMA Buffer. https://github.com/ikwzm/udmabuf, 2019.

[97] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen mei Hwu. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[98] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[99] 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.

[100] 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.

[101] 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.

[102] Irfan Ahmad, Ajay Gulati, and Ali Mashtizadeh. vIC: Interrupt Coalescing for Virtual Machine Storage Device IO. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC)*, 2011.

[103] Mohammadamin Ajdari, Pyeongsu Park, Joonsung Kim, Dongup Kwon, and Jangwoo Kim. CIDR: A Cost-Effective In-line Data Reduction System for Terabit-per-Second Scale SSD Arrays. In *Proceedings of the 25th International Symposium on High Performance Computer Architecture (HPCA-25)*, 2019.

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

[105] 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.

[106] 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.

[107] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[108] 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.

[109] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

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

[111] Nils Asmussen, Marcus Volp, Benedikt Nothen, Hermann Hartig, and Gerhard Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[112] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[113] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on*

Networked Systems Design and Implementation (NSDI), 2012.

[114] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[115] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's Time to Think About an Operating System for Near Data Processing Architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)*, 2017.

[116] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[117] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[118] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM (CACM)*, 60(4), April 2017.

[119] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[120] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[121] Muli Ben-Yehuda, Michael Factor, Eran Rom, Avishay Traeger, Eran Borovik, and Ben-Ami Yassour. Adding Advanced Storage Controller Functionality via Low-Overhead Virtualization. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[122] Theophilus A. Benson. In-Network Compute: Considered Armed and Dangerous. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, 2019.

[123] Shai Bergman, Tanya Brokhman, Tzachi Cohen, and Mark Silberstein. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017.

[124] Yitzhak Birk. Random RAIDs with Selective Exploitation of Redundancy for High Performance Video Servers. In *Proceedings of 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 1997.

[125] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th ACM International Conference on Systems and Storage (SYSTOR)*, 2013.

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

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

[128] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. GAIA: An OS Page Cache for Heterogeneous Systems. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[129] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[130] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories. In *43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*, 2010.

[131] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[132] Adrian M. Caulfield and Steven Swanson. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[133] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(4), November 2004.

[134] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. Software Orchestrated Flash Array. In *Proceedings of the 7th ACM International Conference on Systems and Storage (SYSTOR)*, 2014.

[135] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International Conference on Supercomputing (ICS)*, 2013.

[136] 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.

[137] 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.

[138] 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.

[139] 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.

[140] Feras Daoud, Amir Watad, and Mark Silberstein. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2016.

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

[142] 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.

[143] 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.

[144] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-First Garbage Collection. In *Proceedings of the International Symposium on Memory Management (ISMM)*, 2004.

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

[146] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable Solid-State Storage in Future Cloud Datacenters. In *Communications of the ACM (CACM)*, 2019.

[147] 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.

[148] Nima Elyasi, Mohammad Arjomand, Anand Sivasubramaniam, Mahmut T. Kandemir, Chita R. Das, and Myoungsoo Jung. Exploiting Intra-Request Slack to Improve SSD Performance. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[149] Nima Elyasi, Changho Choi, Anand Sivasubramaniam, Jingpei Yang, and Vijay Balakrishnan. Trimming the Tail for Deterministic Read Performance in SSDs. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2019.

[150] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[151] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[152] 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.

[153] 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.

[154] Laura M. Grupp, John D. Davis, and Steven Swanson. The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[155] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[156] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[157] 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.

[158] Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deploying Safe User-Level Network Services with icTCP. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[159] 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.

[160] 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.

[161] Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin. Transkernel: Bridging Monolithic Kernels to Peripheral Cores. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[162] 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.

[163] 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.

[164] 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.

[165] 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.

[166] Nadav Har'El, Nadav, Gordon, Abel, Landau, Alex, Ben-Yehuda, Muli, Traeger, Avishay, Ladelsky, and Razya. Efficient and Scalable Paravirtual I/O System. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[167] 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.

[168] 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.

[169] 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.

[170] 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.

[171] Soojun Im and Dongkun Shin. Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD. *IEEE Transactions on Computers (TC)*, 60(1), October 2010.

[172] Zsolt Istvan, David Sidler, and Gustavo Alonso. Caribou: Intelligent Distributed Storage. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, 2017.

[173] 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.

[174] Yichen Jia, Eric Anger, and Feng Chen. When NVMe over Fabrics Meets Arm: Performance and Implications. In *Proceedings of the 35th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2019.

[175] Xavier Jimenez and David Novo. Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST)*, 2014.

[176] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soule, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[177] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In

Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP), 2017.

[178] 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.

[179] 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.

[180] Young Tack Jin, Sungjoon Ahn, and Sungjin Lee. Performance Analysis of NVMe SSD-based All-flash Array Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018.

[181] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. YourSQL: A High-Performance Database System Leveraging In-Storage Computing. In *Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB)*, 2016.

[182] William K. Josephson, Lars A. Bongo, David Flynn, Fusion-io, and Kai Li. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[183] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[184] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. GraFBoost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[185] 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.

[186] Myoungsoo Jung, Wonil Choi, John Shalf, and Mahmut Kandemir. Triple-A: A Non-SSD Based Autonomic All-Flash Array for Scalable High Performance Computing Storage Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[187] 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.

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

[189] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[190] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang suk Kee, Francisco Londono, Sangyoon

Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards Building a High-Performance, Scale-In Key-Value Storage System. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[191] Yangwook Kang, Yang suk Kee, Ethan L. Miller, and Chanik Park. Enabling Cost-effective Data Processing with Smart SSD. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2013.

[192] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[193] 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.

[194] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A Case for Intelligent Disks (IDISKs). In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1998.

[195] Bryan S. Kim. Utilitarian Performance Isolation in Shared SSDs. In *the 10th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[196] 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.

[197] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. HyperLoop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[198] 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.

[199] 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.

[200] 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.

[201] 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.

[202] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in SSDs: Model and Quantitative Analysis. In *Proceedings of the 28th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2012.

[203] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. NBA

(Network Balancing Act): A High-performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the 2015 EuroSys Conference (EuroSys)*, 2015.

[204] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[205] Shine Kim, Jonghyun Bae, Hakbeom Jang, Wenjing Jin, Jeonghun Gong, Seungyeon Lee, Tae Jun Ham, and Jae W. Lee. Practical Erase Suspension for Modern Low-latency SSDs. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[206] Tae Yong Kim, Dong Hyun Kang, Dongwoo Lee, and Young Ik Eom. Improving Performance by Bridging the Semantic Gap between Multi-queue SSD and I/O Virtualization Framework. In *Proceedings of the 31st IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2015.

[207] 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.

[208] 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.

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

[210] Ana Klimovic, Christos Kozyrakis, Eno Thereksa, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *Proceedings of the 2016 EuroSys Conference (EuroSys)*, 2016.

[211] 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.

[212] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading Communication with Computing Near Storage. In *50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50)*, 2017.

[213] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity Models: Erasure-Coded Resilience for Prediction Serving Systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[214] Kevin Kremer and André Brinkmann. FADaC: A Self-Adapting Data Classifier For FlashMemory. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[215] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual Remote I/O. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[216] 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.

[217] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M. Swift, and T.V. Lakshman. UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*, 2017.

[218] 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.

[219] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H. Peter Hofstee, Gi-Joon Nam, Mark R. Nutter, and Damir Jamsek. ExtraV: Boosting Graph Processing Near Storage with a Coherent Accelerator. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, 2017.

[220] 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.

[221] Sehwan Lee, Bitna Lee, Kern Koh, and Hyokyung Bahn. A lifespan-aware reliability scheme for RAID-based flash storage. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC)*, 2011.

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

[223] Yangsup Lee, Sanghyuk Jung, and Yong Ho Song. FRA: A Flash-aware Redundancy Array of Flash Storage Devices. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System (CODES+ISSS)*, 2009.

[224] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale Disaggregated Storage. In *the 9th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[225] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos Kawazoe Aguilera, and Michael Walfish. Detecting failures in distributed systems with the FALCON spy network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[226] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[227] 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.

[228] 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.

[229] Huaicheng Li, Ronald Shi, Mingzhe Hao, Martin L. Putra, Fadhil I. Kurnia, Achmad I. Kistijantoro, Sujin Park, Jaeyoung Do, Xing Lin, and Haryadi S. Gunawi. TeaFA: A Tail-Evading Flash Array with a Gray-box IO Determinism Interface. In *In submission.*, 2020.

[230] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[231] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G. Andersen, and Michael J. Freedman. Be Fast, Cheap and in Control with SwitchKV. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[232] Chun-Yi Liu, Jagadish Kotra, Myoungsoo Jung, and Mahmut T. Kandemir. PEN: Design and Evaluation of Partial-Erase for 3D NAND-Based High Density SSDs. In *Proceedings of the 16th USENIX Symposium on File and Storage Technologies (FAST)*, 2018.

[233] 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.

[234] Hang Liu and H. Howie Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[235] Ming Liu, Tianyi Cui, Henrik Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. iPipe: A Framework for Building Distributed Applications on Multicore SoC SmartNICs. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2019.

[236] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[237] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[238] 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.

[239] 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.

[240] 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.

[241] Parag Maharana and K R Kishore. Reducing Latency and Improving Performance Consistency in

NVMeOF. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160811_S304D_Maharana.pdf, 2016.

[242] 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.

[243] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. Disk|Crypt|Net: Rethinking the Stack for High-Performance Video Streaming. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

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

[245] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2015.

[246] 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 2016 USENIX Annual Technical Conference (ATC)*, 2016.

[247] Changwoo Min, Woonhak Kang, Mohan Kumar, Sanidhya Kashyap, Steffen Maass, Heeseung Jo, and Taesoo Kim. Solros: A Data-Centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of the 2018 EuroSys Conference (EuroSys)*, 2018.

[248] 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.

[249] Katherine Missimer and Richard West. Partitioned Real-Time NAND Flash Storage. In *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[250] Lars Nagel, Tim SüS, Kevin Kremer, M. Umar Hameed, Lingfang Zeng, and André Brinkmann. Time-efficient Garbage Collection in SSDs. *ArXiv*, abs/1807.09313, 2018.

[251] 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.

[252] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2018.

[253] 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.

[254] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[255] Pierre Olivier, A K M Fazla Mehrab, Stefan Lankes, Mohamed Lamine Karaoui, Robert Lyerly, and Binoy Ravindran. HEXO: Offloading HPC Compute-Intensive Workloads on Low-Cost,

Low-Power Embedded Systems. In *Proceedings of the 28th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2019.

[256] 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.

[257] 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 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[258] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. MDev-NVMe: A NVMe Storage Virtualization Solution with Mediated Pass-Through. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[259] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[260] Chris Petersen, Wei Zhang, and Alexei Naberezhnov. Enabling NVMe I/O Determinism @Scale. `https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180807_INVT-102A-1_Petersen.pdf`, 2018.

[261] Phitchaya Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable Performance on Heterogeneous Architectures. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[262] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[263] 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.

[264] 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.

[265] Roman Pletka, Ioannis Koltsidas, Nikolas Ioannou, Saša Tomić, Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Aaron Fry, and Tim Fisher. Management of Next-Generation NAND Flash to Achieve Enterprise-Level Endurance and Latency Targets. *ACM Transactions on Storage (TOS)*, 14(4), December 2018.

[266] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating System Transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[267] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[268] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB)*, 1998.

[269] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions To Manage GPUs as Compute Devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[270] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[271] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[272] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[273] Rusty Russell. virtio: Towards a De-Facto Standard for Virtual I/O Devices. In *ACM SIGOPS Operating Systems Review (OSR)*, 2008.

[274] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of the 2012 EuroSys Conference (EuroSys)*, 2012.

[275] Mohit Saxena, Yiying Zhang, Michael M. Swift, Andrea C. Arpaci Dusseau, and Remzi H. Arpaci Dusseau. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[276] 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.

[277] 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.

[278] 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.

[279] 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.

[280] Michael A. Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A Programmable Storage System. In *Proceedings of the 2017 EuroSys Conference (EuroSys)*, 2017.

[281] Sagi Shahar, Shai Bergman, and Mark Silberstein. ActivePointers: A Case for Software Address Translation on GPUs. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[282] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed

OS for Hardware Resource Disaggregation. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[283] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: A Deep Integration of Device and Application for Flash based Key-Value Caching. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[284] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: A Deep Integration of Device and Application for Flash Based Key-Value Caching. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[285] Liang Shi, Kaijie Wu, Mengying Zhao, Chun Jason Xue, Duo Liu, and Edwin H.-M. Sha. Retention Trimming for Lifetime Improvement of Flash Memory Storage Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 35(1), January 2016.

[286] Ji Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Gecko: Contention-Oblivious Disk Arrays for Cloud Storage. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[287] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. OS I/O Path Optimizations for Flash Solid-state Drives. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[288] Ran Shu, Peng Cheng, Guo Chen, Zhiyuan Guo, Lei Qu, Yongqiang Xiong, Derek Chiou, and Thomas Moscibroda. Direct Universal Access: Making Data Center Resources Available to FPGA. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[289] 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.

[290] Mark Silberstein. OmniX: an accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS XVI)*, 2017.

[291] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[292] 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.

[293] Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Snapshots in a Flash with ioSnap. In *Proceedings of the 2014 EuroSys Conference (EuroSys)*, 2014.

[294] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2001.

[295] 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.

[296] 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.

[297] 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.

[298] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[299] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*, 2013.

[300] Animesh Trivedi, Nikolas loannou, Bernard Metzler, Patrick Stuedi, Jonas Pfefferle, Ioannis Koltsidas, Kornilios Kourtis, and Thomas R. Gross. FlashNet: Flash/Network Stack Co-design. In *Proceedings of the 10th ACM International Conference on Systems and Storage (SYSTOR)*, 2017.

[301] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[302] 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.

[303] Jan Vesely, Arkaprava Basu, Abhishek Bhattacharjee, Gabriel H. Loh, Mark Oskin, and Steven K. Reinhardt. Generic System Calls for GPUs. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[304] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low Latency via Redundancy. In *The 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2013.

[305] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the 2014 EuroSys Conference (EuroSys)*, 2014.

[306] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the 2014 EuroSys Conference (EuroSys)*, 2014.

[307] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[308] Louis Woods, Zsolt Istvan, and Gustavo Alonso. Ibex—An Intelligent Storage Engine with Support

for Advanced SQL Off-loading. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, 2014.

[309] 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.

[310] 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.

[311] Suzhen Wu, Yanping Lin, Bo Mao, and Hong Jiang. Garbage Collection aware Cache Management with Improved Performance for Flash-based SSDs. In *Proceedings of the 30th International Conference on Supercomputing (ICS)*, 2016.

[312] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and BoMao. GC-aware Request Steering with Improved Performance andReliability for SSD-based RAIDs. In *Proceedings of the 32th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[313] 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.

[314] 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.

[315] 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.

[316] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. BlueCache: A Scalable Distributed Flash-based Key-value Store. In *Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB)*, 2016.

[317] 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.

[318] 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.

[319] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks. In *Proceedings of the 17th USENIX Symposium on File and Storage Technologies (FAST)*, 2019.

[320] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. AutoStream: Automatic Stream Management for Multi-streamed SSDs. In *Proceedings of the 10th ACM International Conference on Systems and Storage (SYSTOR)*, 2017.

[321] Jisoo Yang, Dave B. Minturn, and Frank Hady. When Poll is Better than Interrupt. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST)*, 2012.

[322] 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.

[323] 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.

[324] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *Proceedings of the 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017.

[325] 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.

[326] 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.

[327] 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.

[328] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut 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.

[329] 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.

[330] 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.

[331] Nannan Zhao, Ali Anwar, Yue Cheng, Mohammed Salman, Daping Li, Jiguang Wan, Changsheng Xie, Xubin He, Feiyi Wang, and Ali R. Butt. Chameleon: An Adaptive Wear Balancer for Flash Clusters. In *Proceedings of the 32th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[332] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[333] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*, 2014.

[334] Aviad Zuck, Philipp Gühring, Tao Zhang, Donald E. Porter, and Dan Tsafrir. Why and How to Increase SSD Performance Transparency. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, 2019.