Enhancing Multi-Tenant Disaggregated Storage Systems with H/W Innovations

Jaehong Min

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2025

Reading Committee:

Arvind Krishnamurthy, Chair

Ming Liu

Baris Kasikci

Ratul Mahajan

Program Authorized to Offer Degree:
Paul G. Allen School of Computer Science & Engineering

University of Washington

Abstract

Enhancing Multi-Tenant Disaggregated Storage Systems
with H/W Innovations

Jaehong Min

Chair of the Supervisory Committee:
Arvind Krishnamurthy
Computer Science & Engineering

Disaggregated storage architectures have become a foundational element in modern datacenter design, enabling independent scaling of compute and storage resources. However, supporting multi-tenant workloads on shared flash-based storage devices remains challenging due to interference, limited hardware isolation, and host-centric software bottlenecks. This dissertation explores system and interface designs that leverage hardware innovations—specifically, SmartNICs and NVMe SSDs—to improve performance, fairness, and adaptability in disaggregated storage systems.

The first part introduces **Gimbal**, a software storage switch that enables multi-tenant-aware scheduling and congestion control on SmartNIC-based storage nodes. Gimbal employs write cost estimation, credit-based flow control, and hierarchical I/O scheduling to isolate tenants and maintain throughput fairness under constrained compute budgets.

We then present **eZNS**, an elastic Zoned Namespace abstraction that enables dynamic resource allocation for ZNS SSDs. eZNS decouples rigid namespace boundaries and allows flexible sharing of zones through global and local overdrive policies. Coordinated I/O planning and proactive space management improve utilization and write efficiency while preserving predictable performance.

Lastly, we propose the **Interposable Transport Protocol (ITP)**, a transport-layer abstraction for SmartNIC-based disaggregated storage. ITP enables in-network request redirection, replication, and remote memory access by treating SmartNICs as protocol-aware dataplane processors. The prototype demonstrates that forwarding and RMA operations can complete within single-digit microseconds using ARM cores, achieving near-RDMA performance without dedicated hardware support.

Collectively, these contributions show that co-designing software abstractions with emerging hardware enables disaggregated storage systems to achieve high performance, adaptability, and strong multi-tenant isolation. The proposed designs lay a foundation for scalable, composable storage infrastructure in modern cloud environments.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Storage disaggregation has gained significant interest recently since it allows for independent scaling of compute/storage capacities and achieves high resource utilization [65, 51, 77, 92, 137]. As datacenter network speeds have transitioned to 100/200 Gbps and the NVMe-oF specification [94] that enables flash-based SSDs to communicate over a network is becoming widely adopted, a disaggregated NVMe SSD can provide microsecond-scale access latencies and millions of IOPS.

Simultaneously, SmartNICs are emerging as a key hardware platform for implementing disaggregated storage. These devices offer lower power consumption compared to traditional x86-based systems while still providing sufficient bandwidth for storage systems. SmartNICs are equipped with wimpy cores that, despite their lower performance, are adequate for handling relatively simple storage protocols. Additionally, SmartNICs come with hardware acceleration engines designed for storage tasks, allowing users to maximize their utility. In modern datacenters, SmartNICs are increasingly being deployed in the form of NICs attached to existing server machines, offloading the storage stack on the initiator side [9, 86]. Various manufacturers are also supporting configurations that leverage SmartNICs as target-side controllers to build JBOFs (Just a Bunch of Flash) [19, 126, 40], further enhancing the flexibility and performance of disaggregated storage systems.

However, making disaggregated storage work efficiently and fairly in multi-tenant environments remains a fundamental challenge. Disaggregated storage systems are shared across tenants running diverse applications with widely varying I/O characteristics. This diversity creates contention at multiple layers—from shared queueing at the network interface to interference inside SSDs caused by shared usage of internal NAND channels, dies, and resources. Ensuring fair bandwidth allo-

cation, low tail latencies, and QoS guarantees without overprovisioning the hardware is critical, especially when tenants operate under strict performance SLAs. Traditional approaches to achieving isolation—such as dedicating entire SSDs per tenant—sacrifice efficiency, leaving capacity and bandwidth underutilized.

The limitations stem, in part, from the opaque and complex nature of conventional SSDs. These devices function as self-contained systems that hide internal operations like garbage collection (GC), write buffering, and wear leveling. Such black-box behavior complicates both performance prediction and control, leading to unpredictable behavior under mixed workloads. Worse, when shared among tenants, internal resource contention can exacerbate tail latency problems and degrade overall system responsiveness.

To address these issues, recent proposals have explored exposing internal SSD structures to the host. One such effort is the NVMe Zoned Namespace (ZNS) interface, which organizes the SSD into logical zones and delegates management tasks like GC and write placement to the host. ZNS promises benefits such as reduced write amplification, lower DRAM overhead, and improved endurance. However, its current design targets single-tenant, log-structured workloads and lacks the flexibility to support multiple concurrent users with distinct access patterns. Simply partitioning the zones is not sufficient when internal NAND resources—like dies and planes—remain physically shared. Moreover, optimal zone sizing and write behaviors may differ substantially across tenants, leading to inefficiencies when fixed configurations are used.

On the other hand, while device-level solutions like ZNS address some internal inefficiencies, broader system performance and control require rethinking storage functionality itself. Storage functions increasingly resemble network services, particularly in how they route and transform I/O requests to meet performance, reliability, and consistency goals. Tasks such as request replication, redirection, and mapping are conceptually similar to packet forwarding, NAT, and route aggregation in networking. Yet, traditional host-based storage stacks are limited in their ability to perform these operations efficiently at line rate, especially when multiple tenants and failure domains are involved. SmartNICs present an opportunity to rethink this design by pushing such logic into the network layer, enabling faster, more programmable control paths.

This thesis addresses the above challenges by proposing architectural and interface-level innovations that make disaggregated storage systems more fair, predictable, and performant in multi-tenant

settings. We investigate how the compute capabilities of SmartNICs can be harnessed for fine-grained I/O scheduling, how the SSD interface can be redesigned to support multi-tenant awareness and elasticity, and how in-network processing can be leveraged to offload and accelerate storage-layer functionality. Through a combination of system design, implementation, and evaluation, we demonstrate that these approaches lead to significant improvements in performance, fairness, and scalability.

1.1 Thesis Statement and Contributions

To overcome the limitations of traditional disaggregated storage systems in multi-tenant environments, this thesis explores a set of architectural and interface-level enhancements grounded in the co-design of hardware and software. Our work focuses on three key areas: leveraging SmartNICs for enforcing fair and low-overhead I/O scheduling, redesigning the SSD interface for elasticity and multi-tenant awareness, and enabling in-network acceleration of storage functions using transport-layer innovations. These techniques collectively address the core challenges of interference, unpredictability, and inefficient resource use that arise when storage is decoupled from compute and shared across multiple tenants.

First, we analyze the performance degradation caused by various I/O interference scenarios in a multi-tenant disaggregated storage system and identified the associated challenges. By conceptualizing the internals of conventional SSDs as complex network systems and viewing them as black-boxes, we propose a software storage switch that leverages network techniques to achieve fair I/O scheduling. Furthermore, through additional evaluations, we demonstrate that the limited computing capabilities of SmartNICs are sufficient to realize this software storage switch.

While this approach achieves notable performance improvements, conventional SSDs obscure the information necessary for accurate performance prediction, such as GC cost, forcing us to rely on approximations. To address this, we turn our attention to the emerging ZNS SSD interface for data center applications. However, the current ZNS is inadequate for multi-tenant environments. Thus, we design the Elastic ZNS (eZNS) interface, which adapts to the number of users and their workload profiles to configure optimal zone settings. This new interface redistributes SSD resources according to the current user I/O profiles, maximizing SSD utilization to enhance application performance. At the same time, it minimizes the probability of users physically sharing the same NAND memory

dies, thereby reducing I/O interference and mitigating unpredictable performance degradation.

Lastly, we demonstrate that leveraging SmartNICs to accelerate storage software at the network layer can significantly enhance end-to-end performance. Storage software performs various types of request routing—such as mapping, redirection, and replication—to ensure data reliability and improve QoS. By translating these tasks into network packet routing problems, we can offload and accelerate them at the network layer. Traditional network transport protocols struggle with tasks such as changing packet destinations without host software stack intervention. To address this, we design a new Interposable Transport Protocol. Additionally, we show that SmartNICs, with their inherent structural advantages, excel as NVMf Routers capable of performing on-path request routing.

1.2 Published Works

- Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. **Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs**. In Proceedings of the 2021 ACM SIGCOMM Conference (SIGCOMM '21).

- Jaehong Min and Chenxingyu Zhao and Ming Liu and Arvind Krishnamurthy. 2023. **eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs**. In Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI '23).

Chapter 2

Backgrounds

2.1 NAND-based SSDs

A NAND-based SSD combines an array of flash memory dies and is able to deliver a bandwidth of several GB/s. It comprises four main architectural components (Figure 2.1): (1) a host interface logic (HIL) that implements the protocol used to communicate with the host, such as SCSI [114] and recent NVMe [93]; (2) an SSD controller, enclosing an embedded processor and a flash channel controller, which is responsible for the address translation and scheduling, as well as flash memory management; (3) onboard DRAM, buffering transmitted I/O data and metadata, storing the address translation table, and providing a write cache; (4) a multi-channel subsystem that connects NAND dies via a high-bandwidth interconnect. As shown in Figure 2.1, a NAND *die* consists of hundreds of *erase blocks*, where each *block* contains hundreds to thousands of *pages*. Each channel holds multiple dies to increase I/O parallelism and bandwidth. Each page encloses a fixed-sized data region and a metadata area that stores ECC and other information. Flash memory supports three major operations: *read*, *program*, and *erase*. The access granularity of a read/program is a page, while the erase command is performed in units of blocks. NAND flash memory has three unique characteristics [2, 24, 26, 54, 76]: (1) no in-place update, where the whole block must be erased before updating any page in that block; (2) asymmetric performance between reads and programs; (3) limited lifetime (endurance) – each cell has a finite number of program/erase (P/E) cycles [63].

To effectively use the NAND flash and address its limitations, SSDs employ a special mapping layer called the flash translation layer (FTL). It provides three major functionalities [28, 57, 100, 143]: (1) dynamically mapping logical block addresses (LBA) to physical NAND pages addresses (PPA);

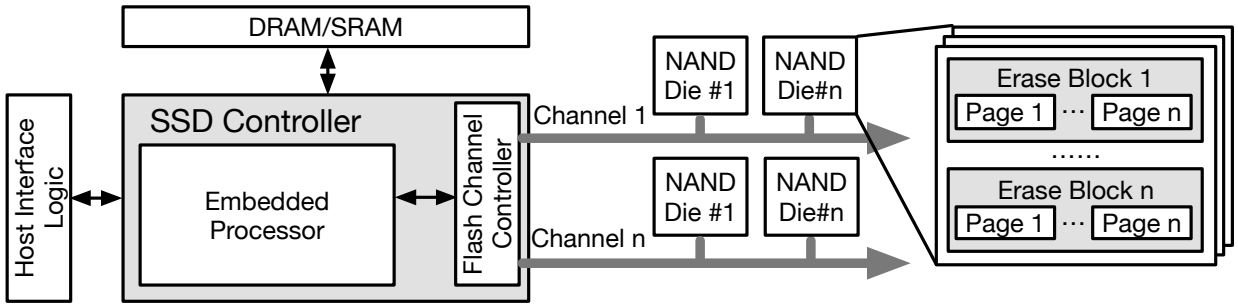


Figure 2.1: The architecture of a NAND-based SSD.

(2) implementing a garbage collection (GC) mechanism to handle the no in-place update issue and asynchronously reclaim invalid pages; (3) applying a wear-leveling technique to evenly balance the usage (or aging property) of all blocks and prolong the SSD lifespan. However, FTL brings in considerable overheads. First, the translation table requires a large amount of DRAM to store the mapping entries, e.g., 1GB for 1TB NAND capacity for 4KB data unit size. Second, when serving a user I/O, the compounding effect of GC and wear-leveling would trigger additional SSD internal writes (i.e., copying valid pages to erase the block) and lead to the WAF (Write Amplification Factor) problem. Third, the FTL does not employ performance isolation mechanisms and incurs significant interference issues under mixed I/O profiles [87, 97].

2.2 NVMe-over-Fabrics Protocol

NVMe-over-Fabrics (NVMe-oF) [94] is an emerging storage protocol to support disaggregation of modern memory devices (e.g., NAND, persistent RAM). It defines a common architecture that supports the NVMe block storage protocol over a range of storage network fabrics (e.g., RDMA, TCP, Fiber Channel). NVMe-oF extends the NVMe base specification [93] and the controller interface. A storage client (i.e., NVMe-oF initiator) first attaches to a storage server (also known as NVMe-oF target) and then issues NVMe commands to the remote controller. The NVMe-oF target comprises two main components: NVMe target core and fabric transport. After setting up a connection with the initiator, it creates a one-to-one mapping between IO submission queues and IO completion queues.

NVMe-over-RDMA relies on memory-mapped IO for all operations. Both the host and the device perform memory read/write of the host memory to modify the related data structures (including submission queue, completion queue, data buffer). NVMe-over-RDMA uses different RDMA verbs

for initiating and fulfilling IO flows. Specifically, *RDMA_SEND* is used to issue a submission capsule to the target and a completion capsule back to the host. All data transfers are performed at the NVMe-oF target using *RDMA_READ* and *RDMA_WRITE* verbs. Thus, the data transfer phase requires no host-side computing cycles. Further, unlike the NVMe specification, NVMe-oF does not introduce an interrupt mechanism for the storage controller. Instead, host interrupts are generated by the host fabric interface (e.g., host bus adapter, RDMA NIC).

Concretely, the request flow of a read/write under NVMe-over-RDMA is as follows: (a) a client host sends an NVMe command capsule (including the NVMe submission queue entry and the scatter-gather address list) to an NVMe-oF target using *RDMA_SEND*; (b) the target process picks up commands from the submission queue. Under a write, it fetches client data via the *RDMA_READ*; (c) the target storage controller then performs a read or write IO execution on the SSDs; (d) in case of reads, the NVMe-oF target issues a *RDMA_WRITE* to transmit data from a local buffer back to the client host memory; (e) the NVMe-oF target process catches the completion signal, builds a response capsule (which contains the completion queue entry), and sends this completion capsule via *RDMA_SEND*. Some NVMe-oF implementations allow for inlining small data blocks (e.g., 4KB) into the capsule, reducing the number of RDMA messages and improving the IO latency.

2.3 Common SmartNIC Architecture

The structural characteristics of modern SmartNICs or DPUs have been studied by numerous researchers [115, 105, 59]. Compared to traditional NICs, SmartNICs incorporate Processing Units (PUs) implemented with ASICs or FPGAs, and they are categorized as either on-path or off-path SmartNICs based on how these PUs are connected to the data path. PUs can be classified into offload engines, which are fully implemented in hardware to handle common functions, and general-purpose processors like ARM or RISC-V cores. Most SmartNICs provide both types of PUs to leverage their combined strengths. Offload engines are designed to accelerate compute-intensive tasks (e.g., encryption, compression, regular expression processing) by processing them at line rate through hardware acceleration. Some SmartNICs even offer logic to offload entire application protocols, such as NVMe-oF, to minimize latency. Additionally, SmartNICs provide separate DRAM that users can utilize as buffer space, enhancing flexible acceleration capabilities. They also include user-programmable DMA engines to efficiently transfer data between host memory and SmartNIC

memory.

Despite the relatively consistent hardware structures across different SmartNICs, the development environments provided to users can vary significantly between manufacturers. Fortunately, many manufacturers have recently adopted DPDK (Data Plane Development Kit) as a foundational software framework, which simplifies development and enhances performance. However, many offload engines and specific features still rely on proprietary SDKs provided by each manufacturer. This dependence on manufacturer-specific SDKs can pose challenges for developers aiming to create portable and flexible applications. Therefore, to maximize the utilization of SmartNICs, it is crucial to understand their structural commonalities and design a simple acceleration middle layer that can effectively connect various SmartNICs and applications. This approach will simplify implementation and enhance the flexibility and portability of the system.

Chapter 3

Gimbal: Enabling Multi-tenant Storage Disaggregation on SmartNIC JBOFs

Recently, SmartNIC-based disaggregated storage solutions, such as Mellanox BlueField and Broadcom Stingray [19, 126], have emerged and become increasingly popular because of their low deployment costs and competitive IO performance compared to traditional server-based approaches. Such a storage node usually comprises a commercial-of-the-shelf high-bandwidth SmartNIC, domain-specific accelerators (like RAID), a PCIe-switch, and a collection of NVMe SSDs, supported by a standalone power supply. Consider the Broadcom Stingray solution as an example. Compared with a conventional disaggregated server node, the Stingray PS1100R storage box is much cheaper and consumes up to 52.5W active power while delivering 1.4 million 4KB random read IOPS at 75.3μs unloaded latency.

Disaggregated storage is shared among multiple tenants for running different kinds of storage applications with diverse IO access patterns. An efficient multi-tenancy mechanism should maximize NVMe SSD usage, ensure fairness among different storage streams, and provide QoS guarantees without overloading the device. However, today's SmartNIC JBOFs[1] lack such essential support, which is non-trivial to build. First, the unpredictable performance characteristics of NVMe SSDs (which vary with IO size, read/write mix, random/sequential access patterns, and SSD conditions) make it extremely hard to estimate the runtime bandwidth capacity and per-IO cost of the storage device. For example, as shown in Section 3.1.2, a fragmented SSD can only achieve 16.9% write bandwidth of a clean SSD; adding 5% writes to a read-only stream could cause a 42.6% total IOPS

[1]JBOF = Just a Bunch of Flash

drop. Second, an SSD has a complex internal architecture, and its controller does not disclose the execution details of individual IO commands. This complicates the IO service time estimation of a tenant as well as the fair scheduler design. Finally, SmartNICs are wimpy computing devices. Besides driving the full networking and storage bandwidth, the amount of available computation per-IO is bounded, i.e., 1μs and 5μs for a 4KB and 128KB read, respectively.

To address these challenges, we design and implement Gimbal, a *software storage switch* that orchestrates NVMe-oF commands among multiple co-located tenants. Gimbal borrows ideas from traditional networking and applies them to the domain of managing storage resources. First, Gimbal views the SSD device as a networked system and applies a *delay-based congestion control* mechanism to estimate its runtime bandwidth headroom. Second, it introduces the virtual slot concept and the write cost estimation logic to enable online characterization of the *per-IO cost*. Finally, Gimbal exposes an SSD virtual view via an end-to-end *credit-based flow control* and *request priority tagging* so that applications are able to design flexible IO prioritization, rate-limiting, and load-balancing mechanisms.

We prototype Gimbal on Broadcom Stingray PS1100R SmartNIC JBOFs and compare with previously proposed multi-tenant storage solutions with isolation mechanisms (i.e., Reflex [66], Parda [42], FlashFQ [117]). Our experiments with synthetic workloads show that Gimbal not only achieves up to x6.6 better utilization and 62.6% less tail latency but also improves the fairness for various complex workloads. We also ported a commercial key-value store (i.e., RocksDB) over a blobstore file system implemented on our disaggregated storage platform. Our implementation employs a hierarchical blob allocator to fully utilize a pool of storage nodes. It manages the storage load and steers read requests based on the runtime loads of the storage devices using an IO rate limiter and a load balancer. Our evaluations show that Gimbal can improve application throughput by x1.7 and reduce tail latency by 35.0% on average.

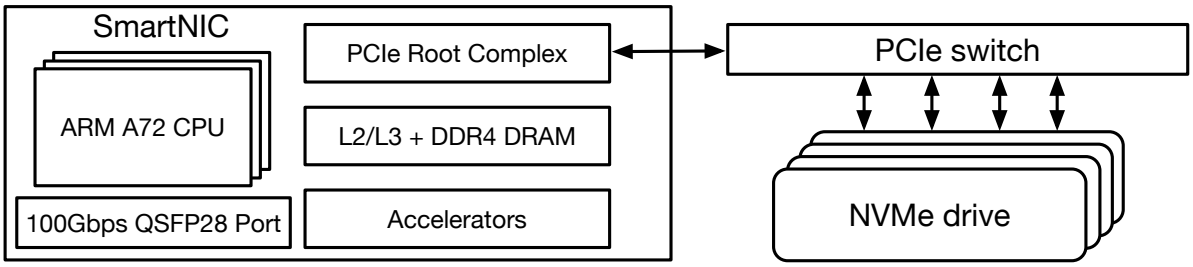


Figure 3.1: Architectural block diagram of the Stingray PS1100R SmartNIC-based disaggregated storage.

3.1 Background and Motivating Experiments

3.1.1 SmartNIC JBOF

SmartNICs [19, 52, 126, 79, 1, 6] have emerged in the datacenter recently, not only for accelerating packet manipulations and virtual switching functionalities [38, 9], but also offloading generic distributed workloads [74, 82, 83, 35]. Typically, a SmartNIC comprises general-purpose computing substrates (e.g., ARM or FPGA), an array of domain-specific accelerators (e.g., crypto engine, reconfigurable match-action table), onboard memory, and a traffic manager for packet steering. Most SmartNICs are low-profile PCIe devices that add incremental cost to the existing datacenter infrastructure and have shown the potential to expand the warehouse computing capacity cheaply.

Lately, hardware vendors have combined a SmartNIC with NVMe drives as *disaggregated storage* to replace traditional server-based solutions for cost efficiency. Figure 3.1 presents the Broadcom Stingray solution [126]. It encloses a PS1100R SmartNIC, a PCIe carrier board, a few NVMe SSDs, and a standalone power supply. The carrier board holds both the SmartNIC and NVMe drives and an on-board PCIe switch connecting the components. The SmartNIC has 8 $\times$ 3.0GHz ARM A72 CPU, 8GB DDR4-2400 DRAM (along with 16MB cache), FlexSPARX [39] acceleration engines, 100Gb NetXtreme Ethernet NIC, and PCIe Gen3 root complex controllers. The PCIe switch offers $\times$16 PCIe 3.0 lanes (15.75GB/s theoretical peak bandwidth), and can support either 2 $\times$ 8 or 4 $\times$ 4 PCIe bifurcation. A Stingray disaggregated storage box with four Samsung DCT983 960GB SSDs, is listed for $3228.0, with likely much lower bulk prices, and is thus much cheaper than a Xeon-based one with similar IO configurations. Unsurprisingly, it also consumes lower power than a Xeon disaggregated server node. The power consumption of a Stingray PS1100R storage node is

Figure 3.2: Read/write latency comparison between SmartNIC and Server JBOFs.

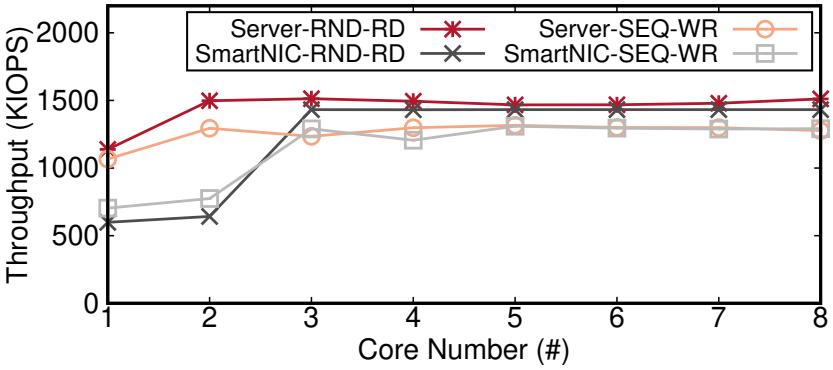


Figure 3.3: Read/write throughput as increasing the number of cores on server and SmartNIC JBOFs.

52.5W at most, nearly one-fourth of the Xeon one (192.0W). Section 3.4.1 describes the hardware configurations. We use a *Watts Up Pro* meter [135] to measure the wall power. The software stack on a SmartNIC JBOF works similar to the Xeon-based server JBOF, except that the NVMe-oF target runs on the SmartNIC wimpy cores instead of the x86 cores. The IO request processing also includes five steps as described above.

SmartNIC JBOFs achieve performance that is competitive to server JBOFs. In terms of unloaded read/write latency, we configure `fio` [37] with one outstanding IO and measure the average latency as we increase the request size (Figure 3.2). When serving random reads, the SmartNIC solution adds 1.0% latencies on average across five cases where the request size is no larger than 64KB. The latency differences rise to 20.3% and 23.3% if the IO block size is 128KB and 256KB, respectively. For sequential writes, SmartNIC execution adds only 2.7μs compared with the server, on average across all cases. We further break down the latency at the NVMe-oF target and compare the

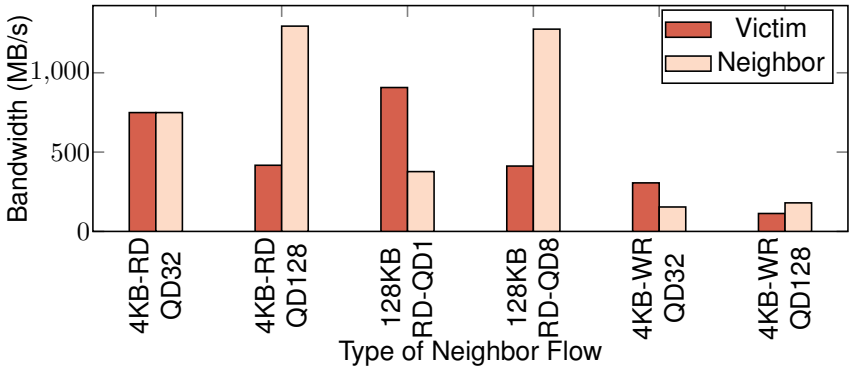


Figure 3.4: Multi-tenant interference in different workloads. (RD:Read, WR:Write, QD:Number of Concurrent I/Os)

server and SmartNIC cases. We find that the most time-consuming part for both reads and writes is the NVMe command execution phase (including writing into the submission queue, processing commands within the SSD, and catching signals from the completion queue). This explains why the latencies on SmartNIC and server JBOFs are similar. For a 4KB/128KB random read, it contributes to 92.4%/86.1% and 88.8%/92.2% for server and SmartNIC, respectively.

Considering bandwidth, SmartNIC JBOF is also able to saturate the storage limit but using more cores. This experiment measures the maximum 4KB random read and sequential write bandwidth as we increase the number of cores. For each FIO configuration, we increase the IO depth to maximize throughput. As depicted in Figure 3.3, the server achieves 1513 KIOPS and 1316 KIOPS using two cores, respectively. In the case of the SmartNIC, it is able to serve similar read and write traffic with 3 ARM cores. One core is enough to achieve the maximum bandwidth under a large request size (i.e., 128KB).

3.1.2 Multi-tenant Disaggregated Storage

Different kinds of storage applications share disaggregated storage nodes, and therefore, we need to provide support for multi-tenancy and isolation between different workloads. Today's NVMe SSDs provide some isolation support. For example, an NVMe namespace is a collection of logical block addresses that provides independent addressing and is accessed via the host software. However, namespaces do not isolate SSD data blocks physically, and requests to access different namespaces can still interfere.

An ideal mechanism should achieve the following goals: (1) provide fairness across tenants; (2) maintain high device utilization; (3) exhibit low computation overheads and predictable delays at the

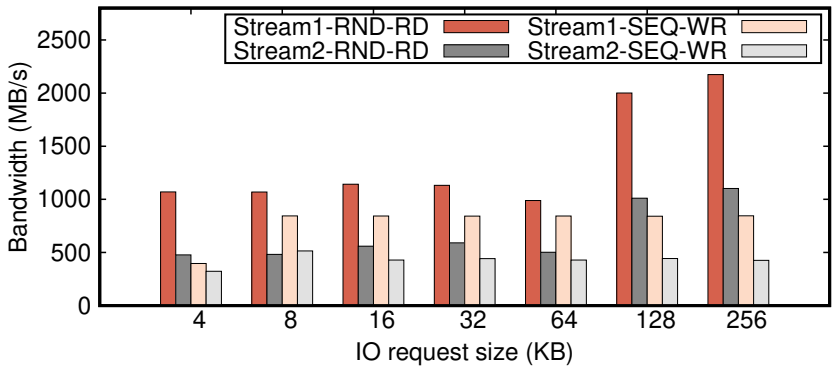


Figure 3.5: Bandwidth of two competing storage streams with different IO depth varying the IO request size.

storage node. Our characterizations show that existing SmartNIC JBOFs present limited support for multi-tenancy, as multiple aspects of IO interference cause unfair resource sharing of storage throughput. In Figure 3.4, the victim flow issues random 4KB reads with 32 concurrent I/Os, and we inject a neighboring flow with various IO sizes, intensity, and patterns. Overall, a flow with high intensity always obtains more bandwidth regardless of the IO size and pattern. For example, the bandwidth of the neighboring flow with random 128KB reads is 377MB/s and 58.4% less than the victim's when it has only one concurrent IO. But, it dramatically rises to 1275MB/s as the concurrent IO is increased to 8 and obtains 3.1x higher bandwidth than the victim. In addition, the bandwidth of the victim decreases significantly when the neighboring flow uses a write pattern. The victim shows 59.1% less bandwidth when the neighbor has the same IO size and intensity but performs writes.

In the disaggregated storage environment, different IO streams interact with each other along the IO path, and will impact their performance. We categorize interference factors into three types, i.e., IO intensity, IO size, and IO pattern (read v.s. write, random v.s. sequential). We then use controlled experiments to demonstrate how each factor affects a storage stream performance and causes unfair resource sharing. Our experiment setup is described in Section 3.4.1 and we use the *fio* [37] utility plus the SPDK fio plugin. Note that (1) fio NVMe-oF clients run on different client servers, and read/write to the same NVMe SSD; (2) we use dedicated NVMe-oF target cores to handle different fio streams.

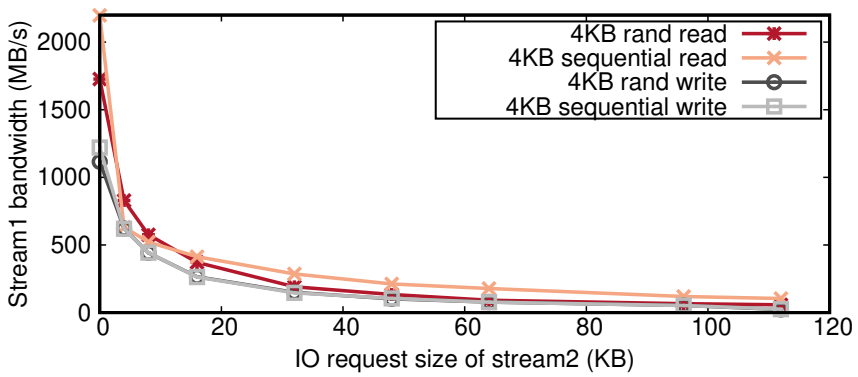


Figure 3.6: Stream1 bandwidth (4KB random/sequential read/write) varying the IO request size for stream2.

IO intensity: indicates how frequent a storage stream issues IO requests to the remote storage. We configure two competing fio storage streams with the same IO request size and read/write type. Both streams can maximize the SSD bandwidth alone, but they differ in the iodepth (i.e., the number of outstanding IO requests) where stream1 issues twice the number of requests as stream2. Figure 3.5 reports the results for 4KB random read and 16KB sequential write, respectively. On average across different IO sizes, stream1 achieves 2.0× and 1.8× more bandwidth compared with stream2 for two cases, respectively. This is because the NVMe-oF target receives more requests from stream1, and submits more NVMe commands to SSD device queues. As a result, stream1 obtains more bandwidth share than stream2.

IO size: presents the read/write block size of an NVMe-oF request within a stream. Again, we configure two competing fio storage streams with the same read/write type and iodepth (which is large enough to saturate the remote bandwidth). The IO size of stream1 is 4KB, and stream2 gradually increase its size. We report the achieved bandwidth of stream1 under four cases in Figure 3.6. Even though two storage streams would submit the similar amount of requests into the NVMe drive, apparently, large IO occupies more bandwidth for any of the random/sequential read/write scenarios. For example, in the case of random read, when stream1 and stream2 are both 4KB, each stream takes around 850.0 MB/s. When a 4KB stream1 mixes with 64KB stream2, stream1 only uses 91.0MB/s, while stream2 achieves 1473.0MB/s.

IO pattern: refers to the type of IO request (e.g., read or write, random or sequential) of a storage stream. Since the NVMe SSD presents distinct execution costs for different typed IOs, when they

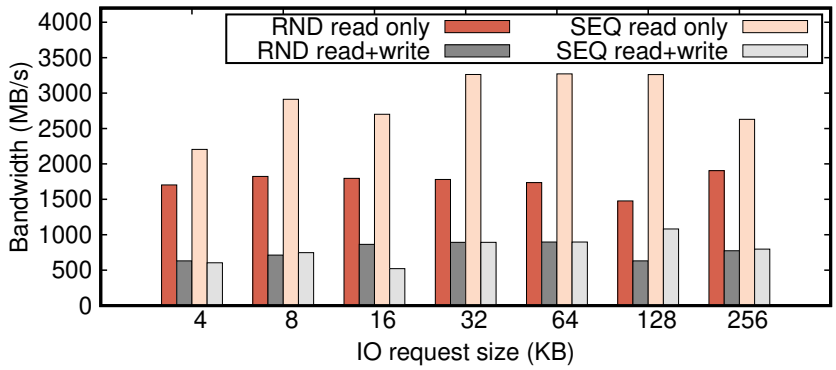


Figure 3.7: Stream1 bandwidth compared between standalone v.s. mixed cases varying the IO request size.

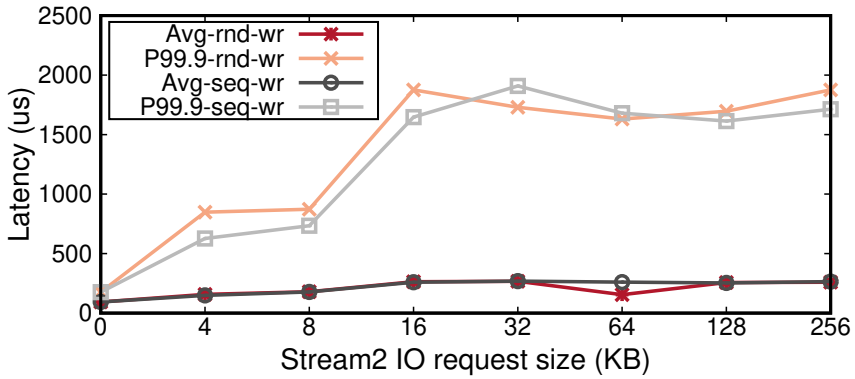


Figure 3.8: Avg./P99.9 latency of 4KB random read mixed with write traffic varying its IO request size.

mix, one would observe significant performance interference in terms of latency and bandwidth. We set up two competing fio streams with the same adequate io depth and io size where stream1/stream2 performs read/write, respectively. As Figure 3.7 shows, compared with the standalone mode, stream1 only achieves 38.9% and 27.3% bandwidth (on average across different IO sizes) for random and sequential cases when mixed IO happens. This is mainly because the read/write request handling within the SSD has lots of overlapping [127, 138], such as device-level IO request queue, FTL engine, write cache, flash chip controller for accessing the planes, etc.

Next, in terms of latency, we run a 4KB random read stream, and couple with another stream issuing random/sequential writes, varying its IO size (Figure 3.8). We also repeat the same experiment by mixing a 4KB sequential write stream with a random/sequential read streams (Figure 3.9). Adding background traffic (stream2) definitely hurts the average and tail latency of the frontend one (stream1). This is due to the head-of-line blocking impact coming from interleaved different

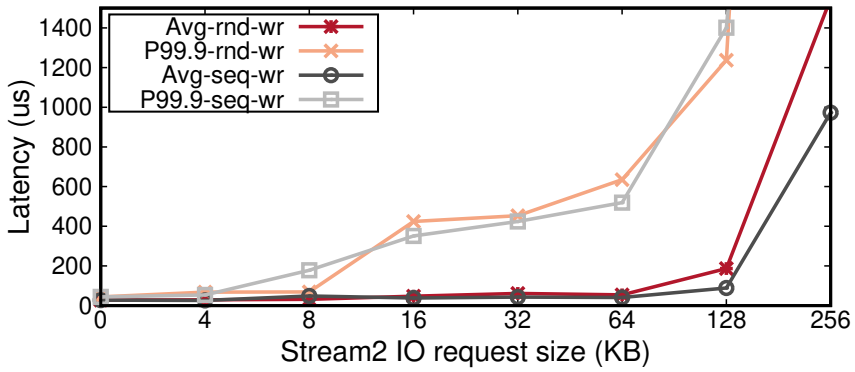


Figure 3.9: Avg./P99.9 latency of 4KB sequential write mixed with read traffic varying its IO request size.

typed IOs. The larger the IO size is, the more latency degradation one would observe. For example, considering random read under sequential write, the average/p99.9 latency of the 128KB case is 1.7× and 2.6× higher than the 4KB case, respectively. Further, the curve becomes flat in Figure 3.8 after 16KB because stream2 has saturated the maximum write bandwidth. Therefore, to isolate different storage streams, one should also take the read/write distribution and carefully monitor its dynamic execution costs at runtime.

We summarize below three challenges to realizing an efficient multi-tenancy mechanism for JBOFs.

Issue 1: IO throughput capacity varies unpredictably with workload characteristics (e.g., random v.s. sequential, read v.s. write, IO size) and SSD conditions. Modern NVMe SSDs favor sequential access due to request coalescing and write-ahead logging [3, 27]. Under the disaggregated setting, the overall IO access patterns become much more random due to request interleaving (i.e., IO blender effect) and impact the IO throughput. SSDs also experience read/write interference. To minimize NAND access contention, an SSD distributes reads/writes across different data channels and NAND chips [30, 34]. Writes are usually slower than reads (unless they are directly served from the SSD controller write buffer) and incur the erase-before-write penalty. A flash page has to clear its data blocks before accepting new values. This causes the write amplification problem [3], where the actual amount of data written to the storage media is more than the intended size. Further, SSDs present asymmetric performance under different IO sizes due to NAND access granularity and die-level parallelism. For example, on a Samsung DCT983 960GB SSD, 128KB read could achieve 3.2GB/s, while the 4KB one maxes out at 1.6GB/s.

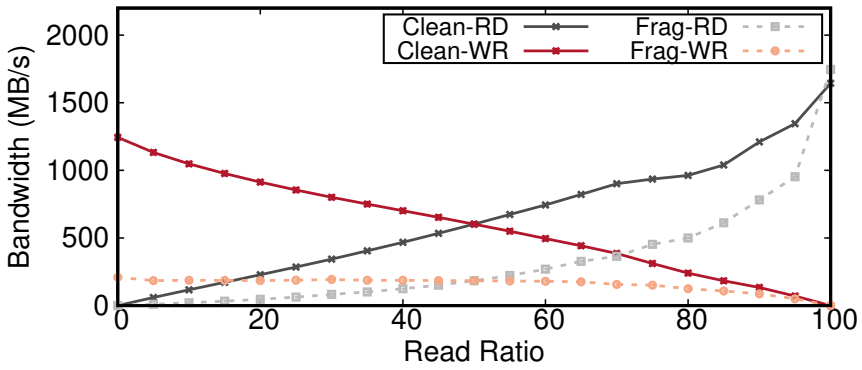


Figure 3.10: 4KB IO performance as increasing the read ratio in clean and fragmented conditions.

The SSD condition, typically characterized by the number of clean flash pages and their location distribution, depends on the previous write history. An SSD uses a flash translation layer (FTL) that maps logical blocks to physical blocks. A new write appends to a pre-erased block first. To improve the write performance and endurance, it also employs wear leveling (i.e., balancing the program/erasure cycles among all data blocks) [3, 144] and garbage collection [3, 110] (i.e., replenishing valid blocks), which complicates estimating the SSD condition. When the SSD is highly fragmented, there are fewer free blocks, and garbage collections are often triggered, which hurts both write and read performance. Figure 3.10 demonstrates the performance impact of write amplification by comparing two conditions: SSDs pre-conditioned with large sequential (clean) and random IO (fragmented) patterns. Adding 5% writes reduces the overall IOPS by 42.6% in the fragmented case. Compared to the clean case, the fragmented one achieves 16.9% and 17.8% of the throughput of write-only and 90/10 W/R scenarios, respectively. Figure 3.11 compares the unloaded latency in a clean SSD v.s. three other scenarios for different IO sizes. On average, across these cases, fragmented SSD, 70/30 read/write mix, and 8 concurrent IOs cause 52.0%, 83.6%, and 80.5% latency increase, respectively. Larger IOs (e.g., 128/256KB) observe more degradation as they are more likely to contend with other requests.

Issue 2: Per-IO cost changes drastically with not only read and write mix but also IO concurrency and IO size. An IO will be delayed when there is execution contention at the SSD controller, head-of-line blocking at the device queue, access contention at the NAND channel/chip/die, or SSD internal activities (e.g., garbage collection). SSD vendors are hesitant to disclose the device execution statistics, making it hard to identify the per-IO cost. This complicates fair

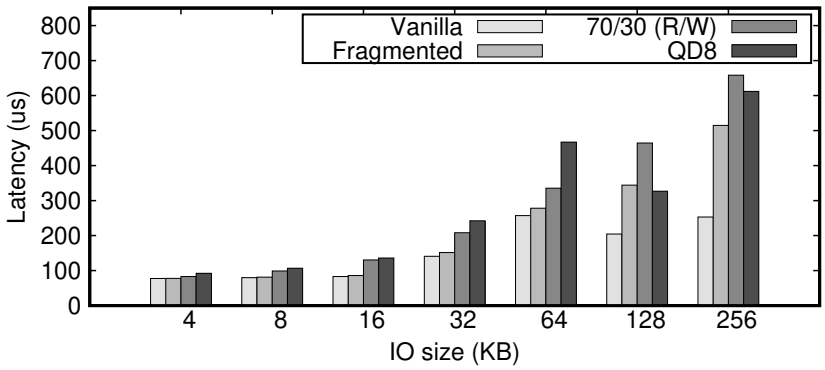


Figure 3.11: Random read latency varying IO request size under different scenarios. QD=queue depth.

IO resource allocations among multiple tenants. Previous studies [121, 84, 66] apply approximate offline-profiled IO cost models, which cannot accurately capture the SSD's dynamic runtime state.

Issue 3: Fairness in a multi-tenant disaggregated storage means that each tenant should receive the same amount of request service time from an SSD controller. The controller's request service time is opaque to the host software stack, complicating the IO scheduler design. Even though modern NVMe SSDs apply a multi-queue interface [17, 64, 127] for multiplexing and multicore scalability, the device usually schedules requests among IO queues in a simple round-robin fashion, impeding the fairness support. Prior works [117, 91, 84, 117, 66] apply deficit/weighted round-robin scheduling or its variants to issue IOs from different tenants. In such approaches, determining an operation's deficit value/weight is vital to achieving fairness. However, as discussed above, simply using IOPS, bandwidth (bytes/s), or some other synthetic static metrics (such as virtual IOPS [121], approximate IO cost mode [84], or SLO-aware token [66]), cannot capture the exact IO execution time. For example, consider a 4KB random read stream mixed with a 64KB one with the same type and IOPS (Figure 3.6 in Appendix). If we use IOPS as the metric, these two streams achieve 91.0MB/s and 1473.0MB/s, indicating that larger IOs dominate the SSD execution; if we use bandwidth instead, smaller IOs could submit four times more requests. Some other works proposed timeslice-based IO schedulers (e.g., Argon [131], CFQ [10], FIOS [117]) that provide time quanta with exclusive device access. These approaches not only violate the responsiveness under high consolidation but also ignore the fact that the IO capacity is not constant (as discussed above).

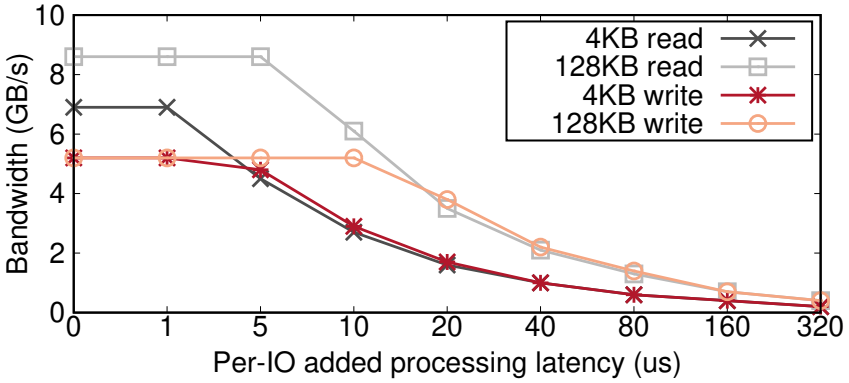


Figure 3.12: 4KB/128KB read/write bandwidth as increasing the per-IO processing cost on SmartNIC JBOFs.

3.1.3 Challenges of SmartNIC-based Disaggregation

SmartNICs have wimpy computing cores compared with the server case. When achieving the same storage load, the request tail latency on SmartNIC JBOFs is higher. For example, the 99.9th latency is 34/66μs on server/SmartNIC if serving ~3000MB/s 4KB sequential writes. To fully drive the storage read/write bandwidth, SmartNICs have little computing headroom for each IO request. We evaluate the achieved bandwidth of 4KB/128KB random read and sequential write as we add to the per-IO processing cost (Figure 3.12 in Appendix). We use all NIC cores in this case. The maximum tolerable latency limit is 1μs and 5μs for 4KB read and write requests, respectively. However, if the request size is 128KB, one can add at most 5μs and 10μ of execution cost for reads and writes without bandwidth loss. Thus, we can only add minimal computation for each storage IO, and the amount of offloading depends on the storage traffic profile.

Fortunately, as we demonstrate in this work, the limited SmartNIC computing capabilities are sufficient to realize a *software storage switch* and equip it with QoS techniques along the ingress/egress data planes for IO orchestration.

3.2 SmartNIC as a Switch

This section describes our design and implementation of Gimbal, a software storage switch with efficient multi-tenancy support (i.e., high utilization, low latency, and fairness) for SmartNIC JBOFs. We first describe its high-level architecture and then discuss each system component in detail.

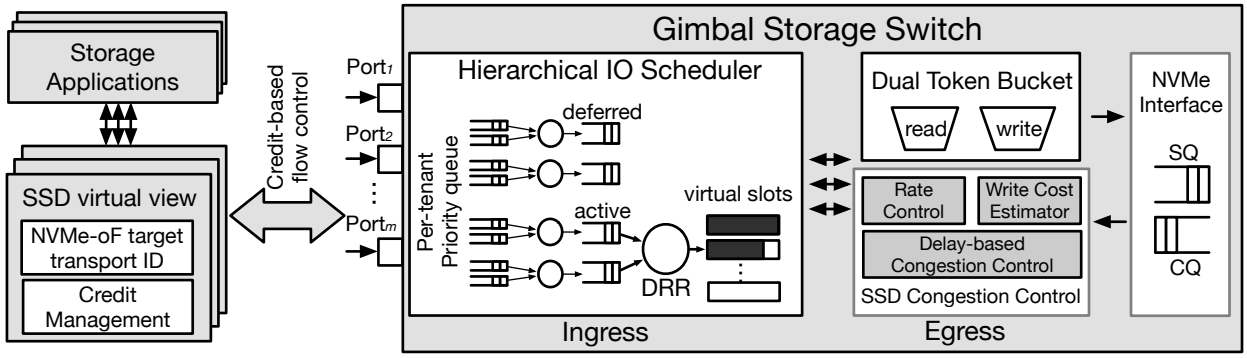


Figure 3.13: An overview of the Gimbal storage switch architecture. On the Broadcom Stingray PS1100R, there are at most 4 NIC ports along with 4 NVMe SSDs.

3.2.1 Overview

Figure 3.13 presents the overall architecture of the software storage switch, inspired by today's SAN switches [31, 21]. It comprises per-SSD pipelines that orchestrate IO flows between NIC ports and NVMe SSDs. Each pipeline is equipped with three major components: (1) *IO scheduler* at the ingress, which provides per-tenant priority queueing and executes IOs in a deficit round-robin fashion (DRR) using a normalized IO unit (called a virtual slot). It exposes a fair queueing abstraction; (2) *delay-based congestion control mechanism* at the egress, which measures the storage bandwidth availability and monitors the SSD load status using IO completion time at runtime. In addition, it employs a rate pacing engine to mitigate congestion and IO burstiness during submission; (3) *write-cost estimator*, dynamically calibrating the SSD write cost based on latency and providing this information to other system components. It implements an approximate performance model that adapts to the workload and SSD conditions.

The switch runs across one or several dedicated SmartNIC cores and listens on active NIC ports. Similar to other works [66], a tenant contains an RDMA qpair (for request send/receive) and an NVMe qpair (for storage command submission/completion). Gimbal focuses on enabling efficient SSD sharing and relies on the remote transport protocol (e.g., RDMA) to address in-network contention.

3.2.2 Delay-based SSD Congestion Control

An SSD has a complex internal architecture. To estimate its instantaneous capacity headroom, we take a black-box approach and borrow the congestion control mechanism from the networking

domain. Specifically, we view the SSD as a networked system, where the controller, circuitry, and NAND chips behave as a router, connection pipes, and end-hosts, respectively. Thus, one can use a TCP-like probing technique to measure its available bandwidth. However, this is non-trivial because (1) the SSD internal parallelism is unknown due to the FTL mapping logic and IO request interference; (2) housekeeping operations (such as garbage collection) are unpredictable and consume a non-deterministic amount of bandwidth when triggered; (3) an SSD is a lossless system without request drops. Therefore, we develop a customized delay-based congestion control algorithm to address these challenges.

Traditional delay-based algorithms (e.g., TCP Vegas [20]) use the measured RTT to calculate the actual bandwidth and take the bandwidth difference (between actual and expected) as a congestion signal to adjust its transmission window. However, the bandwidth metric is ineffective for SSDs because of their opaque parallelism [55, 25]. An SSD employs multiple NAND channels, planes, and dies to improve its bandwidth so that concurrent IO requests could execute in parallel and complete independently. Thus, the IO latency might not be indicative of the consumed SSD bandwidth. Further, a modern SSD usually applies a 4KB-page based mapping mechanism. It splits a large IO into multiple 4KB chunks and then spreads them to different channels as much as possible. Consequently, the latency is not linear with the IO size, and different sized IOs would achieve different maximum bandwidths.

Instead, we explore the direct use of the IO latency as the feedback (like Swift [67]) and take the latency difference between measured and target levels as a congestion signal. This is motivated by our observation that the SSD access latency is very sensitive to the device load and has an impulse response to the congestion as shown in Figure 3.14.

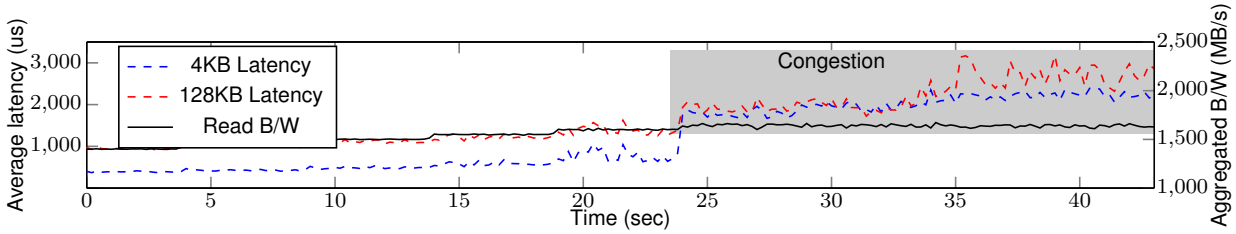


Figure 3.14: Latency increases over time under the 4KB/128KB read mixed workload. Left and right Y-axis represent latency and aggregated bandwidth, respectively.

A key question to realizing this mechanism is determining the latency threshold. We start with a fixed value (e.g., 2ms) and measure the average achieved latency using EWMA (Exponential Weighted Moving Average), where α_D denotes the weight. We find that 2ms fixed threshold is only effective for large IOs (like 64/128KB) but cannot capture the congestion for small IOs promptly. Reducing the threshold (e.g., <1ms) would also not work because it hurts the device utilization. Therefore, we propose a dynamic latency threshold scaling method. It works similar to Reno's congestion control logic [89] for the latency threshold. Specifically, we set up the minimum and maximum threshold and adjust the value based on the EWMA IO latency using,

$$Thres(t) = Thres(t-1) - \alpha_T \times (Thres(t-1) - Latency_{ewma})$$

When the EWMA latency approaches the threshold, it promptly detects a latency increase. Once the EWMA IO latency exceeds the current threshold, it generates a congestion signal, and the threshold is increased to the midpoint of the current and the maximum threshold by ($Thres(t) = (Thres(t-1)+Thresh_{max})/2$. Consequently, Gimbal gets the congestion signal more frequently if the EWMA latency is close to the maximum threshold or grows rapidly. Tuning the min/max threshold should consider the efficiency of congestion detection, device utilization, convergence time, as well as the flash media characteristics (i.e., SLC/MLC/TLC). We plot the typical behavior of the congestion control algorithm as it adapts the delay threshold based on observed latencies. Figure 3.15 shows the latency threshold according to the EWMA latency. As the number of outstanding IO increases, the EWMA latency begins to exceed the threshold, and hit the threshold more frequently as expected.

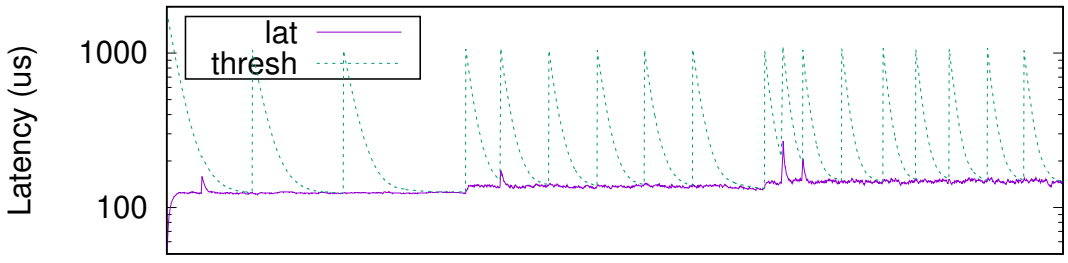


Figure 3.15: Dynamic Latency Threshold (128KB Random Read)

3.2.3 Rate Control Engine

Under a congestion signal, Gimbal applies a rate pacing mechanism to decide the IO submission rate. Traditional congestion window-based approaches are ineffective due to the following reasons. First, since a storage stream contains IOs of various sizes and types, the same window size (i.e., outstanding bytes) would result in different bandwidths for different IO patterns. Second, a short burst of write IOs absorbed by the SSD internal write buffer would cause a significant increase of the available window size. As a result, more IOs would be submitted, overwhelming the device's capability, and the congestion control system would suffer from performance fluctuations. We instead use a rate pacing mechanism with a token bucket algorithm to address this issue. Further, since the single bucket approach would submit write IOs at a wrong rate (i.e., the read rate) and cause severe latency increments, Gimbal employs a dual token bucket algorithm that consists of separate buckets for reads and writes. Tokens are generated by the target rate and then distributed to each bucket according to the IO cost. Specifically, out of the total generated tokens, the amount of $\frac{write\_cost}{write\_cost+1}$ is given to the read bucket and the write bucket receives the remainder or the amount of $\frac{1}{write\_cost+1}$. Lastly, we allow overflowed tokens to transfer between each other. Thus, *dual token bucket* offers the flexibility in rate pacing mechanism while avoiding the burst submission of write I/Os for write intensive workloads. Our dual token bucket works globally and applies to all tenants. Algorithm 1 describes the token update procedure in detail.

Algorithm 1 Dual Token Bucket

```
1:  procedure UPDATE_TOKEN_BUCKETS()
2:      max_tokens ← 256KB
3:      avail_tokens = target_rate × time_since_last_update
4:      read_tokens += avail_tokens × write_cost / (1+write_cost)
5:      write_tokens += avail_tokens × 1 / (1+write_cost)
6:      if read_tokens > max_tokens then
7:          write_tokens += read_tokens - max_tokens
8:          read_tokens = max_tokens
9:      if write_tokens > max_tokens then
10:         read_tokens += write_tokens - max_tokens
11:         read_tokens = min(read_tokens, max_tokens)
12:         write_tokens = max_tokens
```

Token bucket size. Gimbal does not reorder I/Os when dequeued from the DRR scheduler. As a result, one bucket needs to wait for the next IO if (1) the dequeued IO is not for the bucket and (2) the other bucket does not have sufficient tokens to submit the IO. A smaller bucket size would cause one bucket drops lots of tokens during the wait time. Since the read bucket typically waits for the write bucket, a small-sized one would hurt read bandwidth. On the other hand, a large bucket size allows the read bucket to accumulate tokens during the wait time, increasing the read bandwidth. We evaluated the range from 128KB to 512KB and set the size to 256KB empirically, which provides a fair bandwidth allocation for read/write under a mixed-IO workload.

We define four congestion states based on the latency thresholds ($Thresh_{max}$, $Thresh_{min}$, $Thresh_{cur}$) and the measured EWMA latency (Lat_{ewma}), and adjust the target submission rate upon each IO completion. Specifically, these four states are: *overloaded* ($Lat_{ewma} \geq Thresh_{max}$), *congested*($Thresh_{cur} \leq Lat_{ewma} < Thresh_{max}$), *congestion avoidance*($Thresh_{min} \leq Lat_{ewma} < Thresh_{cur}$), and *under-utilized*($Lat_{ewma} < Thresh_{min}$).

For the congestion avoidance and congested states, Gimbal incrementally changes the target rate to either probe for more bandwidth or lower the offered load. The rate is increased/decreased by the IO completion size for the congestion avoidance/congested state. For the overloaded state, the rate is immediately adjusted to be lower than the IO completion rate, and Gimbal discards the remaining tokens in the buckets to avoid a bursty submission. Gimbal periodically measures the completion rate for this. In addition, Gimbal increases the target rate at a faster rate when it observes the underutilized state (parameter β in Algorithm 2).

Gimbal handles the overloaded and under-utilized states in the above manner because the incremental adjustments are meaningful only when the IO pattern doesn't change; incremental adjustments will not converge fast enough for dynamic IO patterns. Specifically, the maximum bandwidth of the SSD may differ dramatically according to the read and write mixed ratio. On a fragmented SSD (defined in Section 3.4.1), the random write bandwidth is about 180MB/s while the random read bandwidth is over 1600MB/s. Consequently, the rate converges slowly if the pattern shifts from write-heavy to read-heavy. Gimbal, inspired by CUBIC[44] and TIMELY[88], adapts an aggressive probing strategy for identifying the desired operating point. This condition only appears when the target rate is insufficient to submit IO before the SSD drains most of the IOs in the internal queue. On the other hand, the target rate may exceed the maximum bandwidth enormously when the

pattern shifts in the opposite direction (i.e., from read-heavy to write-heavy), resulting in the SSD performing at its peak bandwidth but with a latency higher than $Thresh_{max}$. In this case, Gimbal first identifies the current completion rate and sets the target rate to the completion rate. It then further reduces the target rate by an amount equal to that of the size of the completed IO. As a consequence, Gimbal holds the target rate to be lower than that of the SSD's peak capacity until the SSD drains the queued IOs and starts exhibiting a normal latency.

Algorithm 2 depicts the congestion control logic. The submission function is invoked on each request arrival and completion so that it works in a self-clocked manner. With the congestion control mechanism, the SSD maintains an average delay in a stable range providing performance comparable to the device maximum. It adjusts the IO request rate of each SSD while eliminating unnecessary waiting time in the device's internal queue. Crucially, it allows us to determine the performance headroom of an SSD during runtime.

3.2.4 Write Cost Estimation

Performance asymmetry of reads and writes is a well-known issue for NAND devices. Typically, a write consumes more resources than the same-sized read due to the write amplification caused by garbage collection. To capture this, we introduce the *write cost* parameter – the ratio between the achieved read and write bandwidths. For example, let the maximum read and write bandwidths (measured independently) of a given SSD be 1000MB/s and 300MB/s, respectively. The write cost is 3.3 in this case, where we assume that 700MB/s might be used to run internal tasks for the write. We cannot obtain this value directly from the SSD. Instead, we use a baseline setting as the worst case and dynamically calibrate the write cost according to the current SSD state. One can obtain $write\ cost_{worst}$ via a pre-calibration or from the SSD specification, so it is a fixed parameter for a specific SSD.

We update the write cost periodically in an ADMI (Additive-Decrease Multiplicative-Increase) manner. The write cost decreases by δ if the write EWMA latency is lower than the minimum latency threshold and increases to $(write\ cost + write\ cost_{worst})/2$ otherwise. This allows Gimbal to quickly converge to the worst-case when we observe latency increases. By using the latency for adjusting the write cost, Gimbal takes the SSD device optimization for writes into consideration. Specifically, an SSD encloses a small DRAM write buffer and stores user data in the buffer first before flushing it in

a batch to the actual NAND at the optimal time [58]. When the write submission rate is lower than the write buffer consuming capability, writes are served immediately with consistent low latency. In this case, unlike other schemes that have only a fixed cost ratio between read and write, Gimbal reduces the cost down to 1 (i.e., $write\ cost = 1$), same as the read cost. When the write rate rises beyond the write buffer serving capacity, its latency and write cost increase.

3.2.5 Two-level Hierarchical IO Scheduler and Virtual slot

We define the per-IO cost of NVMe SSDs as the average occupancy of operations within the NAND per byte of transmitted data. It is not constant and is affected by numerous factors (Section 3.1.2). This metric reflects how an SSD controller executes different types of NAND commands, such as splitting a large request into multiple small blocks, blocking in an internal queue due to access contention, etc. The IO cost should be considered as a key evaluation parameter for determining fairness since two storage streams would consume significantly different amounts of resources in an SSD even if they achieve the same bandwidth.

IO cost is hard to measure because SSDs do not disclose detailed execution statistics of the NAND and its data channels. IO costs can also be biased by operation granularity. For instance, a large 128KB IO might be decomposed into individual 4KB requests internally and deemed complete only when all individual requests have been processed. In contrast, if we were to pipeline a sequence of $32 \times$ 4KB operations, issuing a new one after each completion, the SSD internal queue occupancy would increase even though the observable outstanding bytes is the same with 128KB IO. We, therefore, use the notion of a *virtual slot*, which is a group of IOs up to 128KB in total (e.g., it might contain up to $1 \times$ 128KB or $32 \times$ 4KB IO commands) and manage IO completion in the granularity of virtual slots. A virtual slot completes when all operations in the slot complete. Each tenant always has the same number of virtual slots. If a tenant runs out of its virtual slots, the IO scheduler defers following IOs until one of its virtual slots completes and becomes available. Gimbal maintains the number of virtual slots per tenant at a minimum and adapts the IO cost variance according to sizes.

The virtual slot mechanism provides an upper bound on the submission rate and guarantees that any sized IO pattern obtains a fair portion of the SSD internal resource. It also addresses the *deceptive idleness* issue [53] (found in many work-conserving fair queuing schedulers) because an

Algorithm 2 Congestion Control with Rate Pacing

```
1: procedure SUBMISSION()
2:     update_token_buckets()
3:     req = DRR.dequeue()
4:     bucket ← dual_token_bucket[req.io_type]
5:     if bucket.tokens ≥ req.size then
6:         io_outstanding += 1
7:         submit_to_ssd(req)
8:     return
1: procedure COMPLETION()
2:     state = update_latency(cpl.io_type, cpl.latency)
3:     if state == overloaded then
4:         target_rate = completion_rate
5:         discard_remain_tokens(dual_token_bucket)
6:     if state == congested or overloaded then
7:         target_rate −= cpl.size
8:     else if state == congestion_avoidance then
9:         target_rate += cpl.size
10:    else
11:        target_rate += β× cpl.size
12:    return
1: procedure UPDATE_LATENCY(IO_TYPE, LATENCY)
2:     lat_mon ← latency monitor for io_type
3:     lat_mon.ewma = (1-α_D) lat_mon.ewma_lat + α_D× latency
4:     if lat_mon.ewma > thresh_max then
5:         lat_mon.thresh = thresh_max
6:         state = overloaded
7:     else if lat_mon.ewma > lat_mon.threshold then
8:         lat_mon.thresh = (lat_mon.thresh + thresh_max)/2
9:         state = congested
10:    else if lat_mon.ewma > thresh_min then
11:        lat_mon.thresh −= α_T× (lat_mon.thresh - lat_mon.ewma)
12:        state = congestion avoidance
13:    else
14:        lat_mon.thresh −= α_T× (lat_mon.thresh - lat_mon.ewma)
15:        state = underutilized
16:    return state
```

allocated slot cannot be stolen by other streams. Gimbal sets the threshold for the number of virtual slots in a single tenant to the minimum number to reach the device's maximum bandwidth if there is only one active tenant. Virtual slots are equally distributed when more active tenants contend for the storage. Since each tenant should have at least one virtual slot to perform IOs, the total number

Algorithm 3 Gimbal IO scheduler

```
1: active_list : tenants that hold at least one slot
2: deferred_list : tenants that wait for slots
3: procedure SCHED_SUBMIT
4:     virt_slot = tenant.curr_virt_slot
5:     io.tenant = tenant
6:     io.virt_slot = virt_slot
7:     virt_slot.submits += 1
8:     virt_slot.size += io.weighted_size
9:     if virt_slot.size > 128KB then
10:        virt_slot.is_full = true
11:        close(virt_slot)
12:        if open_virt_slot(tenant) == fail then
13:            move_to_deferred(tenant)
14: procedure SCHED_COMPLETE
15:    tenant = io.tenant
16:    v_slot = io.virt_slot
17:    v_slot.completions += 1
18:    if v_slot.is_full AND v_slot.submits == v_slot.completions then
19:        reset(v_slot)
20:        if tenant.deferred AND open_slot(tenant) == success then
21:            move_to_active(tenant)
```

of virtual slots may exceed the threshold under high consolidation.

Gimbal integrates the virtual slot concept into the DRR scheduler and ensures the number of slots for each tenant is the same. Similar to the fair queueing mechanism [33, 41], the DRR scheduler divides all tenants that have requests into two lists: *active*, where each one in the list has assigned virtual slots; *deferred*, where its tenants have no available virtual slots and wait for outstanding requests to be completed. Also, the scheduler sets the deficit count to zero when a tenant moves to the deferred list and does not increase the deficit counter of the tenant. Once the tenant receives a new virtual slot, it moves to the end of the active list, and the scheduler resumes increasing the deficit count. Gimbal uses a cost-weighted size for a write IO instead of the actual size ($write\ cost \times IO\ size$) to capture the write cost in the virtual slot. An IO can only be submitted if its deficit count is larger than the weighted size. For example, if the tenant has a 128KB write request when the system is operating under a write cost of 3, the tenant would be allowed to issue the operation only after three round-robin rounds of satisfying tenants in the active list (with each round updating the deficit count associated with the tenant). The whole logic is described in Algorithm 3.

Per-tenant priority queues. The ingress pipeline of Gimbal maintains priority queues for each tenant. The priority is tagged by clients and carried over NVMe-oF requests. When a tenant is being scheduled within an available virtual slot, the scheduler cycles over these priority queues in a round-robin manner, uses each queue's weights to selects IO requests from the queue, and constructs the IO request bundle. This mechanism allows clients to prioritize a latency-sensitive request over a throughput-oriented request.

3.2.6 End-to-End Credit-based Flow Control

The switch applies an end-to-end credit-based flow control between the client-side and the target-side per-tenant queue. This controls the number of outstanding IOs to a remote NVMe SSD and avoids queue buildup at the switch ingress. Unlike the networking setting, where a credit means a fixed-size data segment or a flit [7, 29], the number of credits in our case represents the amount of IO regardless of the size that a device could serve without hurting QoS. It is obtained from the congestion control module (described above). The total credit for the tenant is the number of allotted virtual slots times the IO count of the latest completed slot. A tenant submits IO if the total credit is larger than the amount of outstanding IO.

Instead of using a separate communication channel for credit exchange, we piggyback the allocated credits into the NVMe-oF completion response (i.e., the first reservation field). Similar to the traditional credit-based flow control used for networking, our credit exchange/update scheme (like N23 [69, 68]) also minimizes the credit exchange frequency and avoids credit overflow. However, it differs in that our protocol (1) works in an end-to-end fashion, not hop-by-hop; (2) targets at maximizing remote SSD usage with a QoS guarantee. Algorithm 4 describes the details. Upon issuing a request, if there are enough credits, the command is sending to the target side, and the inflight count is increased; otherwise, applications receive a busy device signal, and storage I/Os will be scheduled later. When receiving a NVMe-OF completion request, it first processes the NVMe-oF command, extracts the credit value, updates the latest credit amount, decrease the inflight count, and then resumes storage workloads via callbacks. Note that all credit manipulation operations are atomic.

Algorithm 4 Credit-based flow control inlined with the NVMe-oF request processing

```
1: procedure NVMEOF_REQ_SUBMIT(req)
2:     target_ssd = req.ssd
3:     if target_ssd.credit_tot > target_ssd.inflight then
4:         ret = submit_nvmeof_req(req)
5:         if ret is success then
6:             target_ssd.inflight = target_ssd.inflight + 1
7:     else
8:         ret = device.busy
       return ret
9: procedure NVMEOF_REQ_COMPLETE(req)
10:    target_ssd = req.ssd
11:    ret = complete_nvmeof_req(req)
12:    if ret is success then
13:        target_ssd.credit_tot = credit_obtain(req)
14:    target_ssd.inflight = target_ssd.inflight − 1
15:    completion_callback(req)                    ▷ Application specific
16:    return ret
```

3.2.7 Per-SSD Virtual View

Our switch provides a managed view of its SSDs to each tenant that indicates how much read/write bandwidth headroom is available at the target so that clients can use the SSD resource efficiently in a multi-tenant environment. In addition, it also supports IO priority tagging, which allows applications to prioritize storage operations based on workload requirements. Such a view enables applications to develop flexible mechanisms/policies, like application-specific IO scheduler, rate limiter, IO load balancer, etc. We later discuss how we integrate it into a log-structured merge-tree key-value store.

3.3 Implementation

3.3.1 Switch Pipeline

We built Gimbal using Intel SPDK [124]. It targets the RDMA transport and extends the basic NVMe-over-Fabric target application by adding three major components: DRR IO scheduler, write cost estimator, and SSD congestion control. The overall implementations follow the shared-nothing architecture and rely on the reactor framework of SPDK. On the Broadcom Stingray platform, we find that one SmartNIC CPU (ARM A72) core is able to fully drive PCIe Gen3 × 4-lanes SSD for any network and storage traffic profile. Therefore, Gimbal uses one dedicated CPU core to handle a specific SSD. Each switch pipeline is dedicated to a specific SSD and shares nothing with other

pipelines that handle different SSDs. It runs as a reactor (i.e., kernel-bypass executor) on a dedicated SmartNIC core asynchronously and will trigger the ingress handler if incoming events come from RDMA qpair (or egress handler when an event is from the NVMe qpair). A reactor uses one or more pollers to listen to incoming requests from the network/storage. Our implementations are compatible with the existing NVMe-oF protocol.

3.3.2 Parameters of Gimbal

Gimbal is affected by the following parameters:

- **Max/Min Delay threshold**. $Thresh_{min}$ is an upper bound of "congestion-free" latency. It should be larger than the highest latency when there is only one outstanding IO, which is 230us on our SSD. We set $Thresh_{min}$ to 250us. $Thresh_{max}$ should ensure that SSDs achieve high utilization for all cases and provide sufficient flexibility to tune other parameters. We characterized this parameter by configuring different types of storage profiles and found that when saturating the bandwidth, the lowest latency is between 500us and 1000us, depending on the workload. Further, to minimize the frequency that Gimbal enters the overloaded state (Section 3.2.3), we set $Thresh_{max}$ to a slightly higher value (i.e., 1500us). Since the minimum latency of the SSD is highly correlated with the NAND characteristics, the parameter values we determine generally apply to other TLC-based SSDs as well. However, 3DXP, SLC, or QLC have completely different device characteristics, and we need to adjust these thresholds appropriately for such devices.
- α_T**,** α_D **and** β. α_T decides how frequently the congestion signal is generated. Although the higher value (e.g., 2^{-8}) has a negligible impact on the result, we set α_T to 2^{-1} and speculatively generate a signal to minimize the trigger rate of the "overloaded" state. α_D is used to tolerate occasional latency spikes and capture recent trends. We set it to 2^{-1}. β determines how fast the target rate is increased in the underutilized state. We set the value to 8. Gimbal can increase the target rate to a peak value within a second (Section 3.2.3).
- **Number of virtual slots and the size:** Gimbal uses 128KB as the slot size as it is the de facto maximum IO size of the NVMe-oF implementation. If the system supports a larger size, the value may be increased accordingly. However, such a larger virtual slot degrades fairness. The threshold of the number of virtual slots for a single tenant is highly related to the outstanding bytes required for the maximum sequential read bandwidth. We found that 8 $\times$ 128KB sequential read I/Os is

the minimum to reach 3.3GB/s bandwidth. Gimbal uses the value 8 for the threshold.

- **Write cost and decrement factor δ:** $Write\ cost_{worst}$ describes the maximum cost of a write IO on the SSD. It can be measured via a microbenchmark or simply calculated by using the maximum random read and write IOPS from the datasheet. We choose the latter way and set the parameter to 9 for the Samsung DCT983 NVMe SSD. The additive decrements factor decides how fast Gimbal reacts to observed write latencies. It should reduce the write cost only when writes are served fast from the SSD (not intermediate low write latency). We set δ to 0.5 empirically.

3.3.3 Case Study: RocksDB over Gimbal

This section describes how we support a log-structured merge-tree (LSM) key-value store (i.e., RocksDB) in a multi-tenant environment. We show how to optimize its various components using the per-SSD virtual view exposed by the storage switch.

Overview. RocksDB [103] is based on an LSM-tree data structure, consisting of two key components: *Memtable*, an in-memory data structure that accumulates recent updates and serves reads of recently updated value; *SSTable*, collections of sorted key-value pairs maintained in a series of levels. When the memtable reaches the size limit, it is persisted as an SSTable by flushing the updates in sequential batches and merging with other overlapping SSTables. Also, low-level SSTables are merged into high-level ones via compaction operations. L_0 SSTables contain the latest data, while $L_1..L_n$ contain the older data. Files within each level are maintained in a sorted-order, with a disjoint key-range for each SSTable file (except in L_0, where each SSTable file can span the entire key-range). Data retrieval starts from the Memtable and will look up multiple SSTables (from level 0 to high levels) until finding the key.

We run the RocksDB over a blobstore file system in an NVMe-oF aware environment. Our modifications include a hierarchical blob allocator over a pool of NVMe-oF backends, an IO rate limiter to control outstanding read/write IOs, and a load balancer that steers read requests based on the runtime loading factor of a storage node.

Hierarchical blob allocator. We allocate storage blobs across a pool of remote storage nodes. Upon an allocation request, it chooses a blob from an available NVMe SSD and then updates blob address mappings. The blob address in our case includes *<NVMe transport identifier, start LBA number, LBA amount, LBA sector size>*. A free operation then releases allocated sectors and puts

them back into the pool. Such an allocator should (1) fully use the storage capacity and provide the appropriate granularity to reduce the storage waste; (2) use a small amount of metadata for block management.

We apply a hierarchical blob allocator (HBA) to satisfy these requirements. First, there is a global blob allocator at the rack-scale that divides total storage into *mega blobs* (i.e., a large chunk of contiguous storage blocks, 4GB in our current implementation), and uses a bitmap mechanism to maintain availability. Within the RocksDB remote environment layer, there is a local agent performing allocation in the granularity of *micro blobs* (i.e., a small chunk of contiguous storage blocks, 256KB in our case). It also maintains a free micro blob list based on allocated mega blobs. A file blob allocation request is served by the local allocator first and will trigger the global allocator when the local free pool becomes empty. To minimize the IO interference impact, we employ a load-aware policy to choose a free mega/micro blob: selecting the one with the maximum credit (i.e., the least load) of the NVMe SSD.

IO rate limiter. Since our RocksDB is built on top of the SSD virtual view, which applies the credit-based flow control with the NVMe-oF target, it automatically supports an IO rate limiter. Specifically, a read/write request is issued when there are enough credits; otherwise, it is queued locally. Under an NVMe-oF completion response, if the credit amount is updated in the virtual view, the database will submit more requests from the queue.

Replication and load balancing We also built a replication mechanism to tolerate flash failures [85, 112] and a load balancer to improve remote read performance. Each file has a primary and a shadow copy that is spread across different remote backends. When there is an allocation request (during file creation/resize), we will reserve primary and secondary microblobs from two different backends in the HBA. If any of their local pools run out of microblobs, a megablob allocation request is triggered. A write operation will result in two NVMe-oF writes and is completed only when the two writes finish. In terms of read, since there are two data copies, RocksDB will issue a read request to the copy whose remote SSD has the least load. We simply rely on the number of allocated credits to decide the loading status on the target. As described before, since credit is normalized in our case, the one with more credits is able to absorb more read/write requests.

3.4 Evaluation

3.4.1 Experiment Methodology

Testbed setup. Our testbed comprises a local rack-scale RDMA-capable cluster with x86 servers and Stingray PS1100R storage nodes, connected to a 100Gbps Arista 716032-CQ switch. Each server has two Intel Xeon processors, 64/96GB memory, and a 100Gbps dual-port Mellanox ConnectX-5 NIC via PCIe 3.0 × 16-lanes. Section 3.1.1 describes the Stingray setup, and we configure it to use 4 × NVMe SSDs. We use Samsung DCT983 960GB NVMe SSDs for all experiments unless otherwise specified. We run CentOS 7.4 on Intel machines and Ubuntu 16.04 on Stingray nodes, where their SPDK version is 19.07 and 19.10, respectively.

Comparison Schemes. We compare Gimbal against the following multi-tenancy solutions designed for shared storage. Since none of these systems target NVMe-oF, we ported these systems onto SmartNIC JBOFs and tailored them to our settings.

- **ReFlex** [66], enables fast remote Flash access over Ethernet. It applies three techniques: kernel-bypass dataplane executor for IO request processing, SLO-based request management, and DRR-like QoS-aware scheduler. We implemented their scheduler model within the SPDK NVMe-oF target process and used the proposed curve-fitting method to calibrate the SSD cost model.
- **Parda** [42], enforces proportional fair share of remote storage among distributed hosts. Each client regulates the IO submission based on the observed average IO latency. We emulated Parda at the NVMe-oF client-side and applied a similar flow control. To obtain the IO RTT, we encode a timestamp into the NVMe-oF submission command and piggy it back upon completion.
- **FlashFQ** [117], is a fair queueing IO scheduler. It applies the start-time fair queueing mechanism [41] and combines two techniques to mitigate IO interference and deceptive idleness: throttled dispatching and dynamically adjusting the virtual start time based on IO anticipation. It uses a linear model to calculate the virtual finish time. We implemented these techniques and calibrated the parameters (including the dispatching threshold and model) based on our SSD.

SSD and Workloads. We emulate two SSD conditions: *Clean-SSD*, pre-conditioned with 128KB sequential writes; *Fragment-SSD*, pre-conditioned with 4KB random writes for multiple hours. They present different "bathtub" characteristics (Figure 3.10). We use a variety of synthetic FIO [37] benchmarks to evaluate different aspects of Gimbal and compare with the other three approaches.

We set the maximum outstanding IOs (i.e., queue depth or QD) 4 and 32 to 128KB and 4KB workloads, respectively. All read workloads in the microbenchmark are random, while 128KB write is sequential and 4KB write is random. We re-condition the SSD with the same pattern before each test. We use YCSB [32] for the RocksDB evaluation.

Evaluation metric. When multiple applications run simultaneously over an SSD, the bandwidth allocated to each application depends on its storage profile and may differ from each other significantly. To examine fairness quantitatively, we propose the term *fair utilization* or *f-Util* as a worker-specific normalized utilization ratio. As shown below, it is calculated by dividing the per-worker achieved bandwidth over its standalone maximum bandwidth when it runs exclusively on the SSD, and the ideal ratio is 1.

$$f\text{-}Util(i) = \frac{per\_worker\_bw(i)}{standalone\_max\_bw(i)/total\_\#\_of\_workers}$$

3.4.2 Utilization

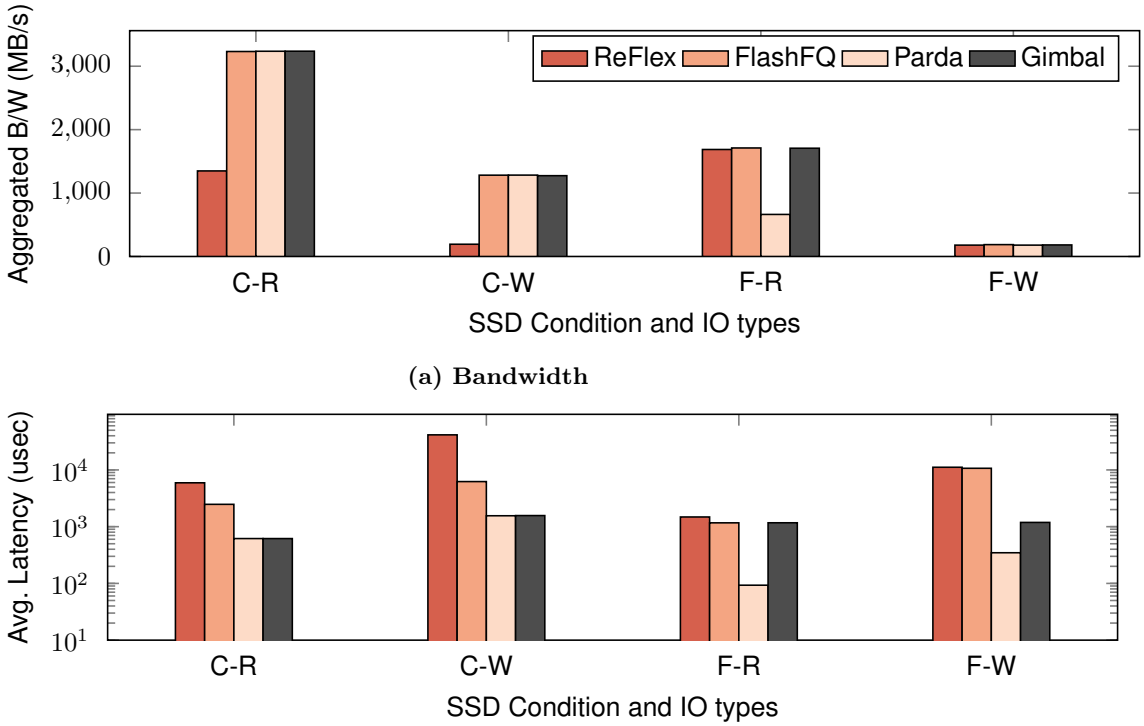
We run 16 workers of the same workload (i.e., standalone benchmark) to evaluate the utilization of each scheme. Figure 3.16 describes the standalone bandwidth of workloads and their average latency. Gimbal performs similarly to the FlashFQ on both Clean-SSD and Fragment-SSD but outperforms ReFlex on Clean-SSD by x2.4 and x6.6 for read and write, respectively. It also outperforms Parda on Fragment-SSD by x2.6 in the read utilization. This is mainly because our SSD congestion control mechanism can estimate the accurate bandwidth headroom so that the scheduler submits just the right amount of IOs. The offline-profiled cost model (used by ReFlex) overestimates the IO cost for writes and large IOs, resulting in lower throughput. Parda fails to reach the maximum bandwidth in the 4KB read workload on Fragment-SSD because the end-to-end RTT between clients and NVMe-oF targets in Parda is too large to detect both the throughput capacity and congestion. In terms of latency, FlashFQ is a work-conserving SFQ scheduler and will issue much more IOs to the SSD without considering the request QoS. Gimbal keeps the latency low by employing the credit-based flow control and performs similar to the Parda for Clean-SSD. The write latency of Gimbal on Fragment-SSD is higher than Parda by x3.4, but it outperforms other schemes by x9 on average. Note that Gimbal does not improve the read latency for Fragment-SSDs because the maximum

number of outstanding IOs in the workload is the same as the total credit count for the tenant.

3.4.3 Fairness

Different IO Sizes. On our testbed, the maximum 128KB random read performance on Clean-SSD (3.16GB/s) is 89% higher than the 4KB read performance (1.67GB/s). We run the benchmark with 16-workers of 4KB read and 4-workers of 128KB read. Figure 3.17a and 3.17d present the bandwidth of one worker for each IO size and the *f-Util* for each scheme. We define the utilization deviation as $\frac{|actual\_Util - ideal\_Util|}{ideal\_Util}$ to identify how each tenant achieves its fair bandwidth share. Gimbal shows the closest bandwidth to the ideal value and the utilization deviation of the 128KB IO case is x2.1, x8.7, and x6.4 less than ReFlex, FlashFQ, and Parda, respectively (x1.9, x4.1 and x7.1 for 4KB). Unlike other schemes, Gimbal considers the cost difference in the virtual slot mechanism so that it is able to allocate 22.0% more bandwidth for 128KB IO in the experiment. The IO cost in ReFlex is proportional to the request size and shows the same bandwidth for both cases.

Different IO Types. In this experiment, we run 16 workers for each read and write and compare the *f-Util* of each scheme. The Clean-SSD case executes 128KB sequential read and 128KB random write, while the Fragment-SSD one contains 4KB random read and 4KB random write. Figure 3.17b, 3.17c, 3.17e and 3.17f describe the aggregated read/write bandwidth and *f-Util*. Gimbal only shows a difference of 13.8% for Clean-SSD and 3.8% for Fragment-SSD between read and write *f-Util*, and outperforms ReFlex, FlashFQ, and Parda by x12.8, x10.4 and x7.5 on Clean-SSD (x4.2, x184.2 and x330.2 on Fragment-SSD), respectively. Gimbal improves the fair bandwidth allocation for read/write cases because it addresses the cost of different IO types and the SSD conditions with the virtual slot and the write cost mechanism. Note that Gimbal shows a lower utilization on Clean-SSDs because the congestion control prevents the latency from growing beyond the maximum threshold. Parda fails to allot the read bandwidth on Fragment-SSD since the write latency for small IO size is not correlated to the IO cost (possibly lower than the same-sized read latency). The linear model of FlashFQ does not provide fairness in modern SSDs, and the read and write bandwidths are the same on both Clean-SSD and Fragment-SSD. ReFlex has a fixed pre-calibrated model, irrespective of SSD conditions, and limits the write bandwidth substantially on Clean-SSD. As a result, it only works on Fragment-SSD; it would require re-calibration of the model in an online manner to adapt to SSD conditions.



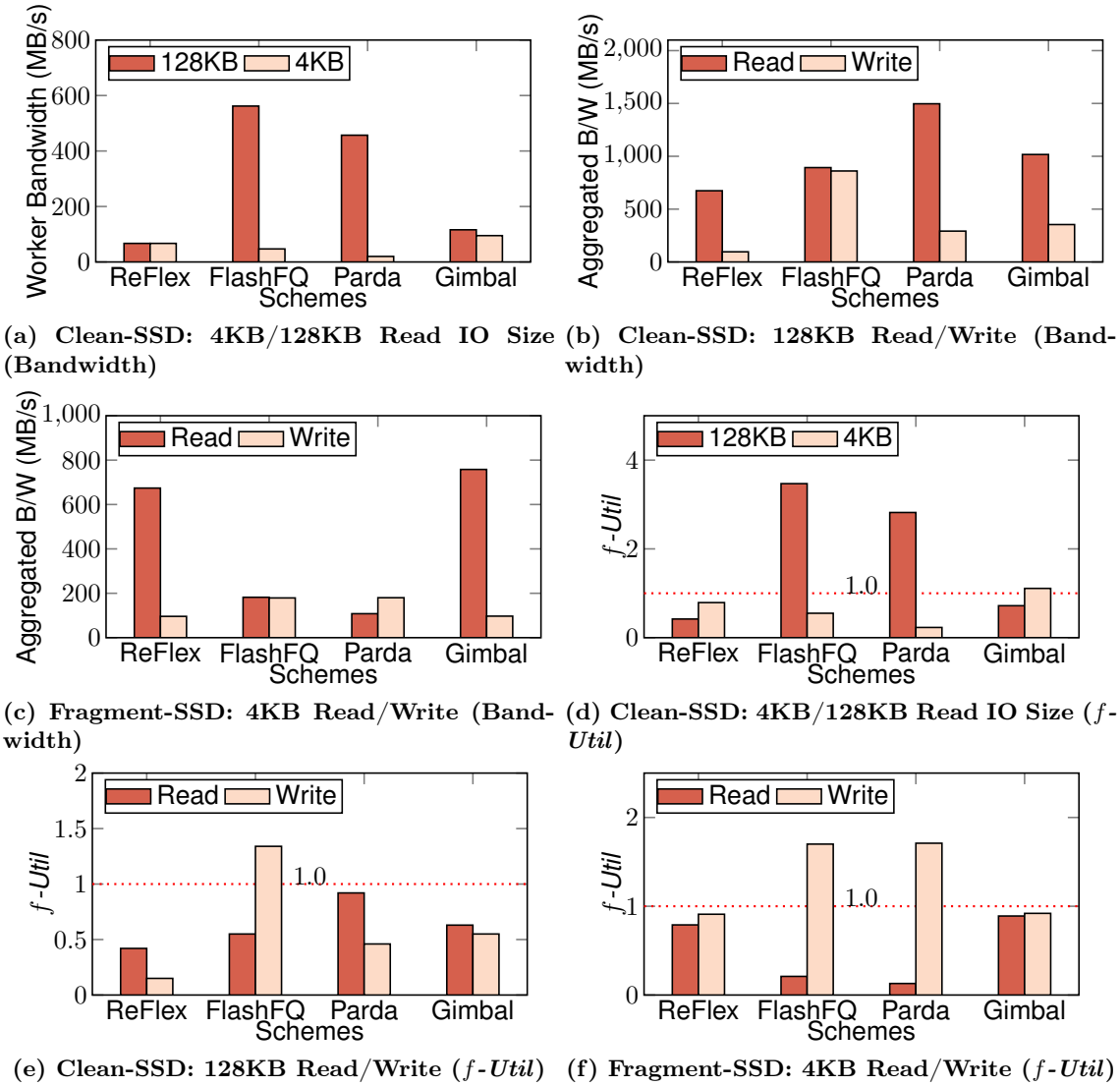
(a) Bandwidth

(b) Avg. Latency (Log-scale)

Figure 3.16: Device utilization on different schemes. IO size is 128KB and 4KB for Clean-SSD and Fragment-SSD, respectively. (C:Clean-SSD, F:Fragment-SSD, R:Read, W:Write)

3.4.4 Latency

Figures 3.18 presents the average and tail latency of different IO types from the previous experiment (i.e., read and write mixed workload). We report the end-to-end average, 99th and 99.9th latency including in-network queueing delay along the data path as well as the request execution time. Gimbal not only maximizes the SSD usage but also provides the best service guarantee compared with Parda. The benefits mainly come from the fact Gimbal can (1) balance the number of outstanding reads and writes to mitigate the IO interference at the device; (2) use credits to control the request rate. On average, compare with Parda, Gimbal reduces the 99th read and write latency by 48.6% and 57.1% for a Clean-SSD (57.5% and 62.6% for a Fragment-SSD). Parda shows an average latency lower than Gimbal on Fragment-SSD, but it suffers from poor utilization and imbalanced resource allocation. The IO RTT observed by the client alone is not sufficient to avoid the congestion on the shared SSD, and Gimbal outperforms Parda in the 99th and 99.9th latency across all workloads. FlashFQ and ReFlex have no flow control mechanisms and incur high latencies under high worker



(a) Clean-SSD: 4KB/128KB Read IO Size (Bandwidth)

(b) Clean-SSD: 128KB Read/Write (Bandwidth)

(c) Fragment-SSD: 4KB Read/Write (Bandwidth)

(d) Clean-SSD: 4KB/128KB Read IO Size (f-*Util*)

(e) Clean-SSD: 128KB Read/Write (f-*Util*)

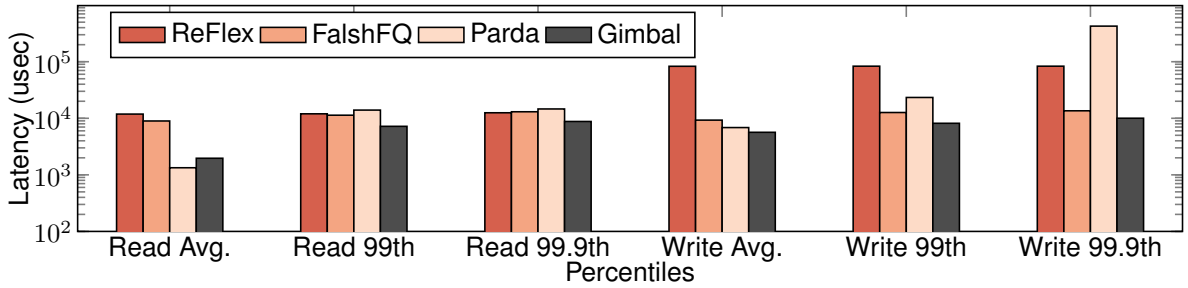
(f) Fragment-SSD: 4KB Read/Write (f-*Util*)

Figure 3.17: Fairness in various mixed workloads on different schemes and SSD conditions

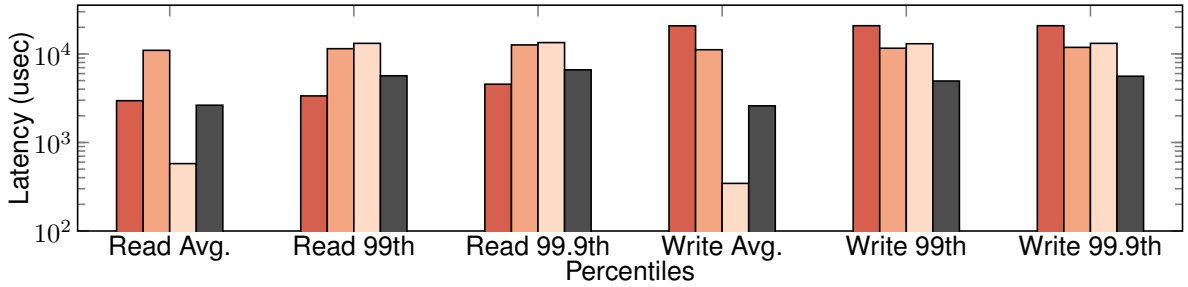
consolidation or a large number of outstanding IOs.

3.4.5 Dynamic Workloads

This experiment changes the workload dynamically and demonstrates the importance of estimating the dynamic write cost. We initiate 8 read workers at the beginning of the experiment and add a write worker at a 5-second interval until the number of read and write workers becomes the same (i.e., 8 read and 8 write workers run simultaneously at the maximum congestion). We then drop one read worker at a time at the same interval. Additionally, we limit the maximum IO rate of the single



(a) Clean-SSD (128KB IO latency, Log-scale)



(b) Fragment-SSD (4KB IO latency, Log-scale)

Figure 3.18: Read/Write IO latency comparison. 16 workers each for Read and Write.

worker to 200MB/s and 60MB/s for read and write, respectively. This simulates an application with a specific IO rate.

Figure 3.19 shows the bandwidth over time for each worker and the latency for each IO type. Gimbal shows that it adapts to the workload characteristics and can accelerate write IOs accordingly. The first write worker benefits from the internal write buffer so that most IOs complete immediately. The latency thus is about 70 usec (i.e., less than the minimum latency threshold) on average, while the average read latency is about 1000 usec at the same moment. As discussed in Section 3.2.4, Gimbal reduces the write cost to 1 and benefits from the SSD device optimization for writes in this case. After the second writer arrives, the write rate starts to exceed the capability of the internal buffer and the latency grows more than 10 times. Gimbal detects such a latency change at runtime and increases the write cost. As a result, the bandwidth for write workers converges to the fair bandwidth share.

3.4.6 Application Performance

In this experiment, we run 24 RocksDB instances over three SmartNIC JBOFs that are configured with different mechanisms on fragmented SSDs. As described before, we enhanced RocksDB with a rate limiter and a read load balancer. We report the aggregated throughput and average/99.9th

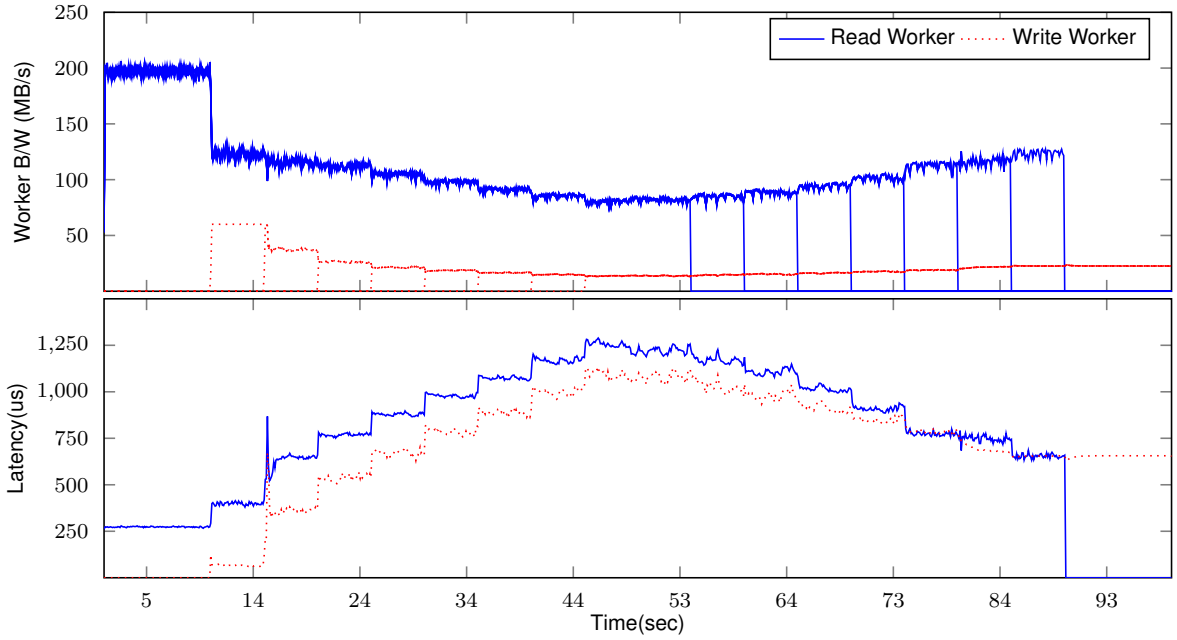
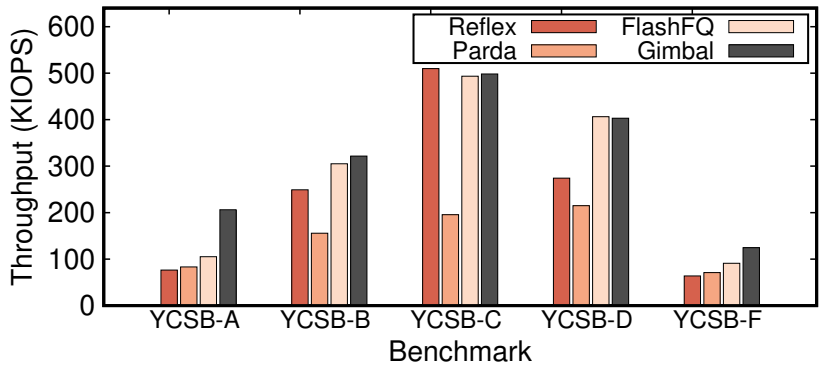


Figure 3.19: Performance over time as the number of workers changes (the latency is a raw device latency measured directly in Gimbal and averaged out every 100ms)

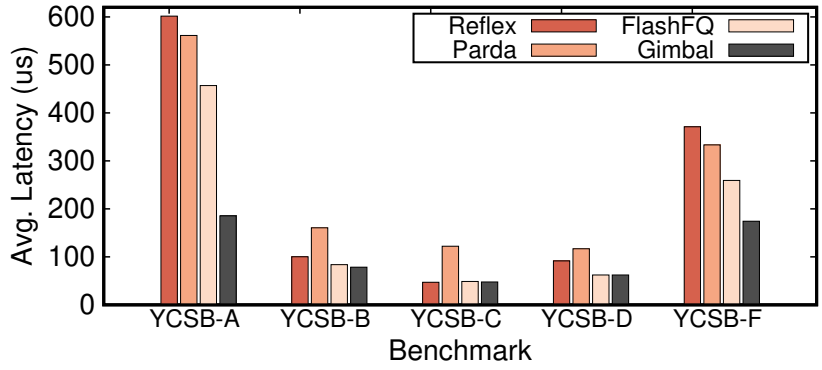
latency (Figure 3.20). On average, across five benchmarks, Gimbal outperforms ReFlex, Parda, and FlashFQ by x1.7, x2.1, x1.3 in terms of throughput, and reduces the average/99.9th latency by 34.9%/48.0%, 54.7%/30.2%, 20.1%/26.8%, respectively. Among them, YCSB-A and YCSB-F (i.e., update-heavy and read-modify-write) workloads benefit the most while YCSB-C (i.e., read-only) observes the least performance improvements. This is because Gimbal can schedule a mix of read/write IOs more efficiently to maximize the SSD performance.

Scalability. We used the same RocksDB configuration and scaled the number of RocksDB instances over three SmartNIC JBOFs (as above). Figures 3.21 and 3.22 present throughput and average read latency, respectively. YCSB-A, YCSB-B, and YCSB-D max out their performance with 20 instances, while YCSB-F achieves the max throughput under 16 instances. For example, the average read latency of YCSB-F with 24 DB instances increases by 38.1% compared with the 16 instance case. YCSB-C is a read-only workload, where 24 instances still cannot saturate the NVMe-oF target read bandwidth, and the average read latency varies little.

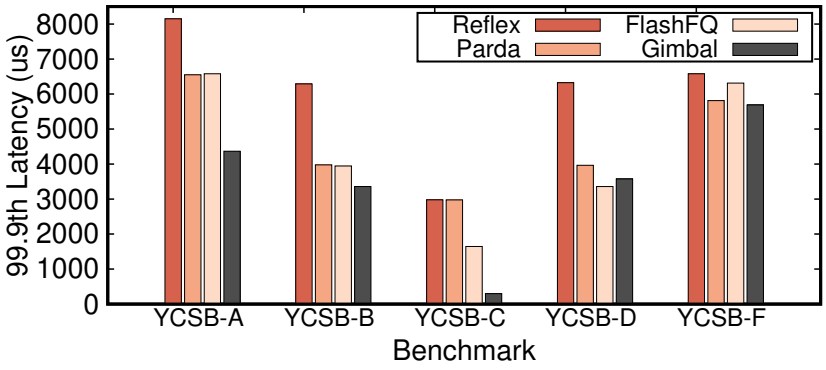
Virtual view enabled optimizations. This experiment examines how the virtual view provided by Gimbal could help improve the application performance. We followed the above RocksDB setting and ran 8 DB instances (from two client servers) over one SmartNIC JBOF. Figure 3.23 presents



(a) Throughput.



(b) Average read latency.



(c) 99.9th read latency.

Figure 3.20: RocksDB performance comparison over four approaches. We configure the YCSB to generate 10M 1KB key-value pairs with a Zipfian distribution of skewness 0.99 for each DB instance.

the read tail latency of five benchmarks comparing three cases (i.e., vanilla w/o optimizations, w/ flow control, w/ flow control and load balancing). On average, across the five workloads, the IO rate limiter enabled via our credit scheme reduces the 99.9th latencies by 28.2% compared with the

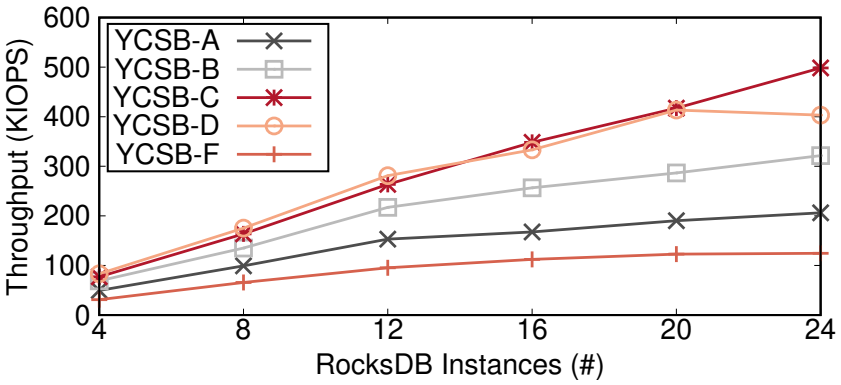


Figure 3.21: Throughput as increasing the number of RocksDB instances.

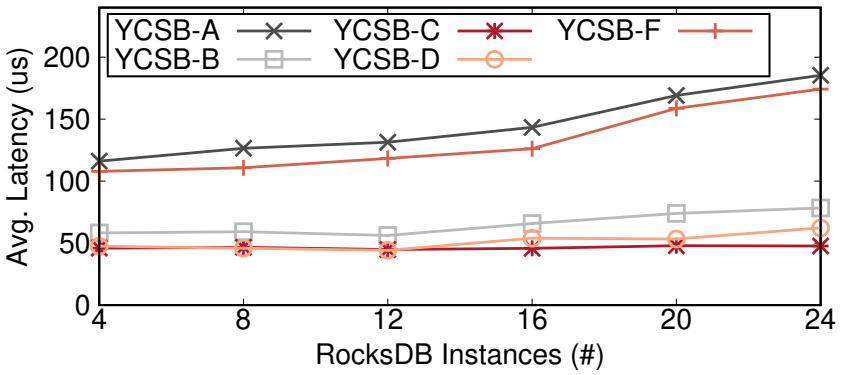


Figure 3.22: Average read latency as increasing the number of RocksDB instances.

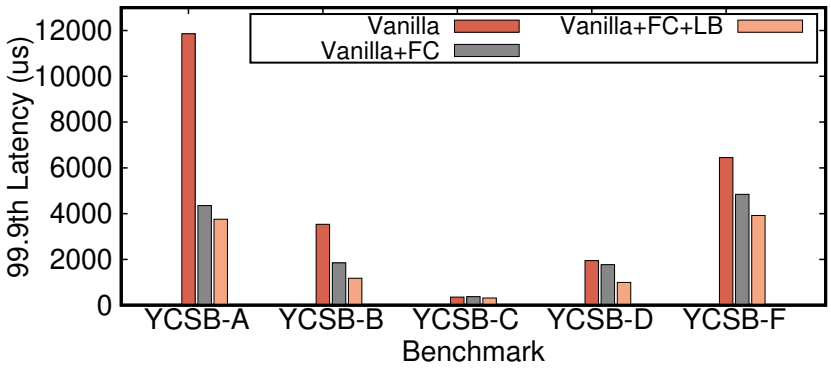


Figure 3.23: Performance improvement via the SSD virtual view enabled optimizations.

		Vanilla SPDK	Gimbal
1 workers (QD1)	Submit	32	52 (+62.5%)
	Complete	16	22 (+37.5%)
16 workers (QD32)	Submit	21	30 (+42.9%)
	Complete	17	25 (+47.1%)

(a) CPU cycle comparison (4KB Read, QD=Queue Depth, 125cycles=1usec)

	Vanilla SPDK	Gimbal
1 CPU core, 1 worker	937 KIOPS	821 KIOPS (-12.4%)
4 CPU cores, 8 workers	2692 KIOPS	2446 KIOPS (-9.2%)

(b) The maximum IOPS with NULL device (4KB Read IO)

Table 3.1: Overhead comparison with vanilla SPDK

vanilla one. The request load balancer, which could choose one replica that has more bandwidth, further reduces the tail by 18.8%.

3.4.7 Overheads

We evaluate the overhead of Gimbal in two ways. First, we compare the average CPU cycles of the submission and completion procedure with a vanilla SPDK NVMe-oF target on SmartNIC. As shown in Table 3.1a, Gimbal adds 37.5-62.5% more cycles to realize the storage switch. Although Gimbal adds some overhead on average in the pipeline, it does not hurt the performance for PCIe Gen3 SSD as discussed in Section 3.1.3. Future PCIe Gen4 SSDs could achieve ~7GB/s bandwidth or 1 MIOPS. We run the 4KB read benchmark with a NULL device (which does not perform actual IO and returns immediately) to measure the maximum IOPS of Gimbal on SmartNIC. Gimbal performs 821 KIOPS with 1 SmartNIC core. Gimbal could support a multi-core implementation if one can distribute the active tenants to each core in a balanced manner. For example, we extend the experiment to the 4-core case and find that Gimbal achieves 2446 KIOPS, indicating that SmartNIC-based Gimbal can support next-generation SSDs. We also expect Gimbal to scale up with future powerful ARM cores or specialized engines.

3.4.8 Generalization

This experiment evaluates how Gimbal performs on a different type of SSD. We run the same microbenchmark (Section 3.4.3) using Intel DC P3600 1.2TB model. This SSD uses 2-bit MLC NAND, presenting 33.5% lower 128KB read (2.1GB/s) and 35.0% higher 4KB random write (243MB/s) in

	ReFlex	Parda	FlashFQ	Gimbal
BW estimation	Static	Dynamic	✗	Dynamic
IO cost & WR tax	Static	✗	Static	Dynamic
Fair queueing	@Target	@Client	@Target	@Target
Flow control	✗	✓	✗	✓

Table 3.2: Comparison of four multi-tenancy mechanisms.

terms of bandwidth. We tune the $Thresh_{max}$ to 3ms for better read utilization as it achieves higher tail latency than DCT983 for 128KB read. Gimbal adapts the characteristics of the SSD and performs similarly to the DCT983 case in terms of the *f-Util*. Specifically, it shows 0.63 and 0.72 of *f-Util* for read and write under the clean condition and 0.58 and 0.90 for read and write under the fragmented condition.

We also run Gimbal on Xeon E5-2620 v4 CPU and compare the overhead with the vanilla SPDK. Gimbal performs 10.8% lower (1368 KIOPS) than the vanilla SPDK (1533 KIOPS) for 4KB read performance with the NULL device, similar to the result on SmartNIC.

3.4.9 Summary

Table 3.2 summarizes the high-level differences between Gimbal and the other approaches. Gimbal outperforms the other approaches because it dynamically estimates the available bandwidth and IO costs for each storage device based on current conditions and workloads, performs fine-grained fair queueing at the NVMe-oF target across tenants, and uses credit-based flow control to adjust the client behavior. The other approaches lack one or more such support. For example, ReFlex and FlashFQ use an approximate offline-profiled SSD cost model that hurts scheduling efficiency. They also don't regulate client-side IOs, causing significant delays. Parda employs a client-side mechanism, which uses the end-to-end delay as the congestion signal for its flow control, and performs limited fair queueing within a server without coordination with other hosts. Such a high latency feedback control loop is unsuitable for low-latency high-bandwidth NVMe SSDs.

3.5 Related Work and Discussion

Shared storage with QoS guarantee. Prior work has explored how to achieve high device utilization and fairness for local and remote shared storage [131, 98, 117, 47, 42, 66, 84, 121, 91, 129]. As discussed in Section 3.4, these approaches cannot precisely estimate the IO capacity (under different SSD conditions) and per-IO cost (under various mixed workloads), causing inefficiencies

when applied to the NVMe-oF based disaggregated setting. There are also some timeslice-based IO schedulers (e.g., CFQ [10], Argon [131]) that allocate time quanta for fairness and provide tenant-level exclusive access. They usually target millisecond-scale slow storage and would hurt responsiveness, utilization, and fairness when used with fast NVMe SSDs. Recently, researchers have tried to leverage ML techniques (e.g., a light neural network) to predict the per-IO performance [46]. The predicted result is a binary model (indicating if an IO is fast or slow) and can only help improve the admission control of storage applications for predictable performance, which is insufficient to achieve fairness. IOFlow [129] proposes a software-defined architecture for enforcing end-to-end policies of storage IOs running in the data center. It adds queueing abstractions along the data plane and relies on a centralized controller for translating policies to queuing rules. We believe Gimbal can serve as one of its execution stages at the storage server.

Remote storage IO stacks. Researchers [65] have characterized the performance of iSCSI-based disaggregated storage and proposed optimization opportunities, such as parallel TCP protocol processing, NIC TSO offloading, jumbo frames, and interrupt affinity pinning. ReFlex [66] provides a kernel-bypass data-plane to remove the NVMe storage processing stack, along with a credit-based QoS scheduler. Also, i10 [49] is an efficient in-kernel TCP/IP remote storage stack using dedicated end-to-end IO paths and delayed doorbell notifications. Researchers [43] have also characterized the server-based NVMe-oF performance. Our work targets SmartNIC JBOFs that use NVMe-oF protocols.

Disaggregated storage architecture. New hardware designs have been proposed to address the limitations of existing disaggregation designs. Sergey et al. [73] proposes four kinds of in-rack storage disaggregation (i.e., complete, dynamic elastic, failure, and configuration disaggregation) and explores their tradeoffs. Shoal [120] is a power-efficient and performant network fabric design for disaggregated racks built using fast circuit switches. LeapIO [75] provides a uniform address space across the host Intel x86 CPU and ARM-based co-processors in the runtime layer and exposes the virtual NVMe driver to an unmodified guest VM. As a result, ARM cores can run a complex cloud storage stack. We focus on emerging SmartNIC-based disaggregated storage solutions.

Programmable packet scheduling. Researchers have explored new HW/SW programmable packet schedulers. PIFO [123] proposes a push-in first-out queue that enables packets to be pushed into an arbitrary position based on their calculated rank and dequeued from the head. PIEO [119]

then improves PIFO scalability and expressiveness via two capabilities: providing each scheduling element a predicate; dequeuing the packet with the smallest index from a subset of the entire queue. SP-PIFO [5] further approximates the PIFO design via coarse-grained priority levels and strict FIFO queues. It dynamically adjusts the priority range of individual queues by modifying the queuing threshold. PANIC [78] combines PIFO with a load balancer and a priority-aware packet dropping mechanism to explore efficient multi-tenancy support on SmartNICs. Carousel [106] and Eiffel [107] are two efficient software packet schedulers, which rely on a timing wheel data structure and bucketed integer priority for fast packet operations. Compared with these studies, a key difference is that Gimbal focuses on scheduling storage IO requests among NVMe SSDs instead of individual packets. Exposing programmability from the Gimbal traffic manager to realize flexible scheduling policies will be our future work.

Emerging storage media. QLC NAND has gained significant attention because of its cost and capacity advantage over TLC. However, its performance characteristic is worse than TLC, presenting a higher degree of read and write asymmetry [81]. We expect the techniques introduced by Gimbal could also apply to QLC SSDs. Gimbal could also support the 3DXP device. However, it is suitable for a local cache rather than the disaggregated storage [141]. 3DXP also provides a similar performance of read and write with in-place update [140] and might not benefit from Gimbal as much as NAND devices.

Hardware-accelerated Gimbal. The pipelined architecture of Gimbal has a standard interface for both ingress and egress and can be plugged into any hardware-accelerated NVMe-oF implementation. In addition, Gimbal itself is portable to a hardware logic with ease using a framework such as Tonic [8].

3.6 Conclusion

This paper presents Gimbal, a software storage switch that enables multi-tenant storage disaggregation on SmartNIC JBOFs. Gimbal applies four techniques: a delay-based congestion control mechanism for SSD bandwidth estimation, a new IO cost measurement scheme, a two-level hierarchical I/O scheduling framework, and an end-to-end credit-based flow control along with an exposed SSD virtual view. We design, implement, and evaluate Gimbal on the Broadcom Stingray PS1100R platforms. Our evaluations show that Gimbal can maximize the SSD usage, ensure fairness among

multiple tenants, provide better QoS guarantees, and enable multi-tenancy aware performance optimizations for applications. This work does not raise any ethical issues.

Chapter 4

eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization

In modern data centers, performance isolation for storage systems has become a critical consideration, particularly to prevent long-tail latencies and ensure Service Level Agreements (SLAs) with users. Some storage systems address this by physically segregating tenants onto different SSD devices, a strategy favored by latency-sensitive applications. However, this approach can lead to inefficiencies, as SSD performance typically scales with NAND capacity, resulting in wasted storage space. Instead, in shared storage systems, researchers have explored various solutions [76, 80, 139], leveraging features such as IO Determinism [36] and Open-Channel SSD [18], previously proposed for device management.

The NVMe Zoned Namespace (ZNS) is a newly introduced storage interface and has received significant attention from data center and enterprise storage vendors. By dividing the SSD physical address space into logical zones, migrating from device-side implicit garbage collection (GC) to host-side explicit reclaim, and eradicating random write accesses, a ZNS SSD significantly reduces device DRAM needs, resolves the write amplification factor (WAF) issue, minimizes costly overprovisioning, and mitigates I/O interference. However, the performance characteristics of the ZNS interface are not well understood. In particular, to build efficient I/O stacks over it, we should be cognizant of (1) how the underlying SSD exposes the zone interface and enforces its execution restrictions; (2) what trade-offs the device's internal mechanisms make to balance between cost and performance. For example, the device-enforced zone placement makes the actual I/O bandwidth capacity of a zone contingent on how a ZNS SSD allocates zone blocks across channels/dies. Further, a zone is not

a performance-isolated domain, and one could observe considerable I/O interference for inter-zone read and write requests. Therefore, there is a strong need to understand its idiosyncratic features and bring enough clarity to storage applications.

We perform a detailed performance characterization of a commodity ZNS SSD, investigate its device-internal mechanisms, and analyze the benefits and pitfalls under different I/O profiles in both cases: application running on the exclusive device (Standalone), and applications sharing a device with other tenants (Co-located). Using carefully calibrated microbenchmarks, we examine the interaction between zones and the underlying SSD from three perspectives: zone striping, zone allocation, and zone interference. We also compare with conventional SSDs when necessary to investigate the peculiarity of a ZNS SSD. Our experiments highlight the interface's capabilities to mitigate the burden on I/O spatial and temporal management, identify constraints that would cause sub-optimal performance, and provide guidance on overcoming the limitations.

We propose eZNS, a new interface layer that provides a device-agnostic zoned namespace to the host system. It mitigates inter-/intra-zone interference and improves device bandwidth by allocating active resources based on the application workload profile. eZNS remains transparent to upper-layer applications and storage stacks. Specifically, eZNS comprises two components: the zone arbiter on the control plane and a tenant-cognizant I/O scheduler on the data plane. The zone arbiter maintains the device shadow view, which manages zone allocations and realizes dynamic resource allocation through a zone ballooning mechanism. It manages available active resources in two groups: essentials and spares. Essentials provide a minimum guarantee to applications, determining the number of logical active zones, while spares boost the bandwidth of zones adaptively, based on the zone utilization of applications, using the zone overdrive technique. By doing so, the zone arbiter allows serving applications to maximize the device capability by enabling the maximum device parallelism given the workload profile and rebalancing inactive bandwidth across namespaces. It also employs a zone reclaiming mechanism to prevent one from holding spares, thus avoiding trapping bandwidth in idle zones. The I/O scheduler of eZNS leverages the intrinsic characteristics of ZNS, where there are no hardware-hidden internal bookkeeping operations. eZNS applies a local congestion control for reads and a global admission control for writes. Read I/Os become more predictable in ZNS interface, and one can directly harness this property to examine inter-zone interference. Thus, the read congestion control applies to each logical zone independently, monitoring read latency and

maintaining the congestion window per zone. On the other hand, write I/Os share a performance domain due to the write cache architecture of the SSD, which causes global congestion across all active zones. To address this, eZNS applies the same I/O admission rate to active logical zones, thereby preventing excessively busy writing zones from monopolizing the write cache. It determines the admission rate based on the average write latency of the device, allowing write I/Os from small writers to bypass the admission control while throttling those from busy writers. Our I/O schedulers mitigate the interference independently but improve overall system performance cooperatively. We demonstrate benefits in the evaluation (§4.4) over micro-benchmarks and RocksDB.

In summary, our study not only empirically demonstrates the benefits of using a coarse-grained zoned namespace interface but also reveals the inadequacies of underlying device mechanisms causing performance non-determinism. Essentially, a ZNS SSD is a *semi-transparent* storage device to the host and should be viewed as a *gray box*. Our insights emphasize the importance of imbuing data locality into storage allocation for high utilization, the necessity of employing a global centralized control for fairness guarantees, and the importance of coordinating co-located zone execution for performance predictability. Meanwhile, eZNS keeps the promises of the original ZNS interface. For example, eZNS does not overly utilize the write bandwidth as there is no housekeeping writes (except the zone reclaiming which is very rare). In other words, it does not generate write I/Os in addition to what the application requested. As long as the I/O demand of applications is consistent, eZNS will only improve the QoS without sacrificing the lifetime of devices.

In summary, our study not only empirically demonstrates the benefits of utilizing a coarse-grained zoned namespace interface but also exposes the shortcomings of underlying device mechanisms that lead to performance non-determinism. A ZNS SSD is a *semi-transparent* storage device to the host, akin to a gray box. Our insights underscore the necessity of implementing a global centralized control for fairness guarantees, and the significance of coordinating co-located zone execution for performance predictability. Meanwhile, eZNS remains faithful to the original ZNS interface's promises. For instance, eZNS ensures that write bandwidth isn't excessively utilized, as it doesn't have housekeeping writes (aside from rare zone reclaiming events). In other words, it doesn't generate additional write I/Os beyond what the application requests. As long as the applications' I/O demands remain consistent, eZNS enhances Quality of Service (QoS) without compromising device lifespan.

4.1 Background and Motivation

This section introduces the ZNS SSD and its features, and discusses the problems with the existing zoned interface.

4.1.1 Zoned Namespace SSDs

ZNS SSDs, a successor to Open-Channel (OC) SSDs [14, 18], have recently been developed to overcome the aforementioned limitations of conventional SSDs. There are several commodity ZNS SSDs from various vendors [101, 109, 108, 136]. A ZNS SSD applies the same architecture as a conventional one but exposes the zoned namespace interface. A namespace is a separate logical block address space, like a traditional disk partition, but managed by the NVMe device controller rather than the host software. The device may control the internal block allocation of namespaces to optimize the performance based on the device-specific architecture. In ZNS SSD, the namespace comprises multiple zones instead of blocks in the conventional one, and each namespace owns dedicated *active resources* that are used to open and write a zone.

A ZNS SSD divides the logical address space of namespaces into fixed-sized zones, where each one is a collection of erase blocks and must be written sequentially and reset explicitly. ZNS SSDs present three benefits: (1) Maintain coarse-grained mappings between zones and flash blocks and apply wear-leveling at the zone granularity, requiring much smaller internal DRAM. For example, a ZNS SSD with a 24MB block size has at least 6,144 times smaller table size compared to the traditional 4KB page mapping [122]; (2) Eliminate the device-side GC and reclaim NAND blocks via explicit zone resets by host applications, which mitigates the WAF and log-on-log [142] issues and minimizes the over-provisioning overhead; (3) Enable the placement of opened zones across different device channels and dies, providing isolated I/O bandwidth and eliminating inter-zone write interference.

A zone has six states (i.e., *empty*, *implicitly open*, *explicitly open*, *closed*, *full*, *read only*, and *offline*). State transitions are triggered by either write I/Os or zone management commands (i.e., RESET, OPEN, CLOSE, and FINISH), as shown in Figure 4.1. A zone must be opened before issuing writes, but it is capable of serving reads in any state except the *offline* state. A device internal error will cause the zone to enter either a *read only* or an *offline* state, where it cannot transit to states other than *offline*. Zones in empty or closed state transition to the open state in

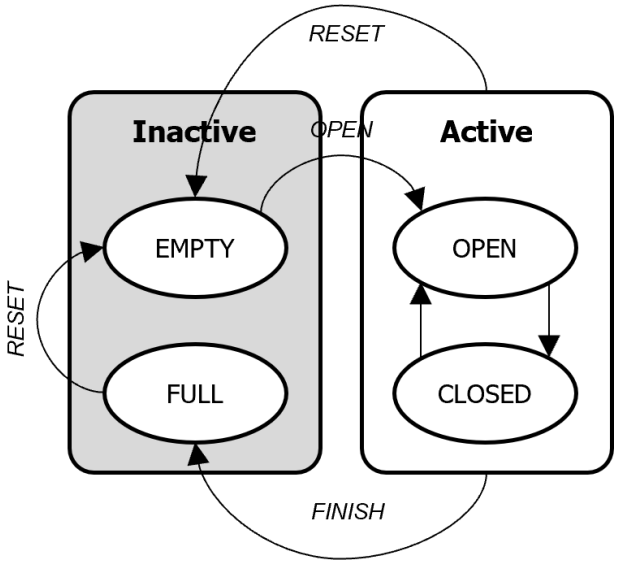


Figure 4.1: The Zone State Diagram of ZNS SSD.

either explicit (by OPEN management command) or implicit (by write I/O) way. An opened zone can transition to the closed state (by CLOSE management command) or the full state (by FINISH or write I/O reaching the end of the zone). *closed* and *open* (both implicit and explicit) are *active* states that require the device to maintain NAND metadata for incoming user write I/Os, limiting the maximum number of active zones. SSDs employ the write cache in DRAM to align the wide range of user I/O sizes to the NAND program unit and comply with the NAND-specific requirements (timings and program order). In case of a sudden power-off failure, the device flushes uncommitted data in the cache using batteries or capacitors as an emergency power source [130, 146]. Since active zones must have a buffer backed by energy devices for at least one NAND program unit in the cache, the maximum number of active zones is also constrained by the size of the write cache.

A zone provides three I/O commands: *read*, *write*, and *append*. The *append* works similarly to the nameless write [145] but improves the host I/O efficiency rather than the internal NAND page allocation. Compared with the normal write, a zone *append* command does not specify the LBA in the I/O submission request, whilst the SSD will determine it at processing time and return the address in the response. Thus, user applications can submit multiple outstanding operations simultaneously in contrast with the normal writes that submits only one I/O at a time to avoid out-of-order execution violating the sequential constraint of the zone interface in the storage stack or device queues. Random writes are disallowed on ZNS SSDs, and the zone is erased as a whole (via

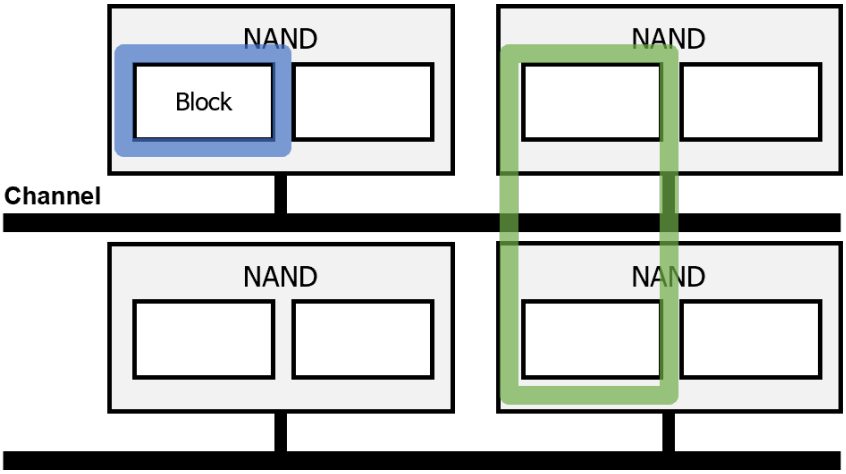


Figure 4.2: The examples of physical zone placement in small (Blue) and large (Green) zone SSDs.

the RESET). A ZNS SSD delegates the FTL and GC responsibilities to user applications, where they are performed at the zone granularity, thus eliminating traditional SSD overheads. While the user software consumes more host memory (as it is now responsible for the performing the mapping), it can be minimized by integrating the zone management into the application logic [15] or file system [72, 71].

4.1.2 Small-zone and Large-zone ZNS SSDs

Zones can be classified into two types: *physical zone* and *logical zone*. Physical zones are the smallest unit of zone allocation and consist of one or more erasure blocks on a single die. They are device-backed and offer fine-grained control over storage resources. In contrast, logical zones refer to a striped zone region consisting of multiple physical zones. They can be implemented by either the device firmware or application and provide higher bandwidth through striping. Figure 4.2 presents the physical zone placement in different types of ZNS SSDs. Large-zone ZNS SSDs provide coarse-grained large logical zones with a fixed striping configuration that spans multiple dies across all internal channels but offers limited flexibility for controlling device behavior from the host software. This simplifies zone allocation but exposes a small number of active zones available for application allocation (e.g., 14 zones [136]). As a result, large-zone SSDs are more suitable for scenarios with small numbers of tenants, where the number of active zones required is not high. In addition, the application-agnostic fixed striping configuration does not adapt to workload profiles, resulting in low bandwidth utilization. Small-zone ZNS SSDs operate under similar hardware constraints but

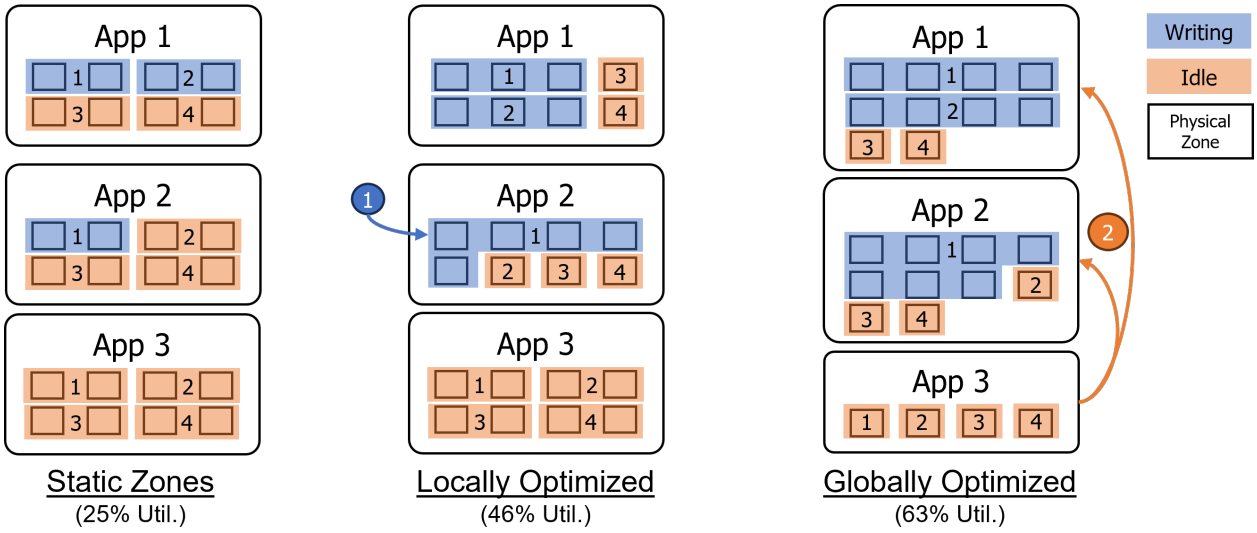


Figure 4.3: An example of the static striping configuration and the optimizations.

expose finer-grained physical zones. Each zone is contained within a single die but sufficiently large to encompass at least one erasure block. Small-zone SSDs provide greater flexibility and more active resources (e.g., 256 zones in our testbed ZNS SSD) to support more I/O streams. In addition to increased flexibility, small-zone SSDs reduce the need for application-level garbage collection, especially while managing large numbers of small objects. Recent studies also corroborate some of these points. Specifically, Bae et al. [11] advocate a zone to be as small as possible to reduce the interference caused by high zone-reclaiming latencies. ZNS+ [45] also prefers small zones as it minimizes the latency of COPY operations performed frequently in its F2FS implementation.

4.1.3 Need for an Elastic Interface

The ZNS SSD brings in two key benefits. First, it exposes controllable garbage collection to host applications, eliminating obtrusive I/O behaviors precipitated by device internal bookkeeping I/Os. This also alleviates write amplification and reduces flash over-provisioning. Second, it only allows sequential writes within a zone and thereby mitigates certain I/O interference observed in a conventional SSD. Both prior studies [11, 16, 128, 45] and our characterizations (§4.2) below demonstrate these points. However, existing ZNS SSDs have one significant drawback: **the zoned interface is static and inflexible**. After a zone is allocated and initialized, its maximum performance is fixed regardless of the underlying device capability, its I/O configurations cannot adapt to runtime workload characteristics, and cross-zone I/O interference yields unpredictable I/O executions.

First, the performance profile of a zone-sized storage partition hinges on physical zone placement and stripe configuration, which should align with application requirements. Despite significant benefits from the flexibility of the user-defined logical zone, application-managed zone configuration would sustain sub-optimal performance due to the lack of knowledge of other tenants sharing the device. In addition, it imposes another burden on application developers, as with OC SSDs.

Figure 4.3 depicts an illustrative scenario involving three applications that employ different optimization strategies. In this scenario, the computing device comprises 24 physical zones, with each application having access to a maximum of four logical zones for application-specific zone management. The static striping configuration, typically managed by the device, allocates physical zones evenly, provisioning two physical zones for each logical zone. However, this static allocation may lead to suboptimal device utilization, particularly when applications only actively use a subset of available zones, resulting in a utilization rate as low as 25%, as shown in Figure 4.3. To address this issue, one may introduce an application-managed zone configuration. With this approach, each application can concentrate its available physical zones on actively writing zones, as illustrated in the second column of Figure 4.3. This dynamic allocation significantly improves utilization, raising it to 46%. However, it is important to note that this application-local optimization may face challenges in efficiently allocating resources, particularly when there is an idle application that holds valuable zones. One alternative is implementing a centralized resource manager responsible for redistributing physical zones from idle applications to active tenants. This centralized approach has the potential to further enhance utilization, achieving a utilization rate of 63%. However, it necessitates a carefully designed mechanism to prevent resource starvation when previously idle applications become active, ensuring a balanced allocation of resources.

It is non-trivial to develop a complete application profile that captures every aspect of I/O execution characteristics, such as read/write block size and distribution, I/O concurrency, and command interleaving degree. The existing zoned interface fails to adapt to the changing workload behavior. Users have to over-provision the zone resources when configuring a zone based on the worst-case estimation. In Figure 4.4, it is shown that the RocksDB over ZenFS [15] actively writes to only a fraction of the zones it maintains in the *active* state. This leads to inefficient utilization of valuable active resources in the ZNS SSD. Similarly, file systems like BtrFS [104] and F2FS [72] support ZNS SSDs but write user data to only one zone at a time, resulting in suboptimal utilization of the avail-

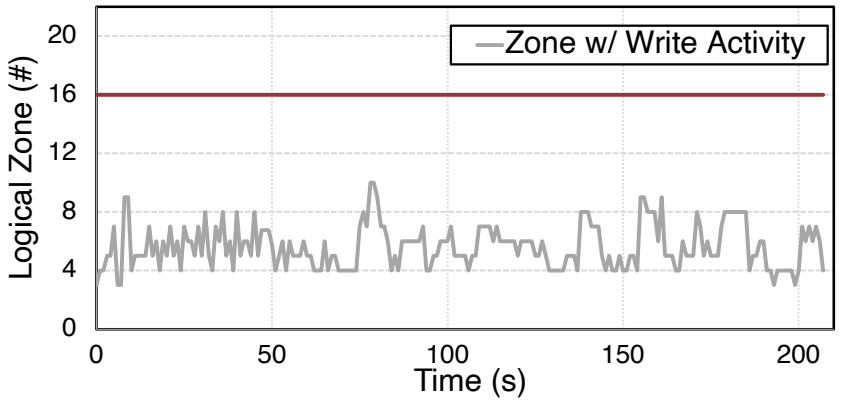


Figure 4.4: The number of zone with actual write activity when running the *fill-random* workload over the RocksDB. The storage backend is ZenFS. The maximum number of active zones is 16 (red line).

able active resources. This issue is further exacerbated when the device has multiple namespaces serving different applications. In such cases, each application only utilizes a fraction of the available bandwidth, wasting valuable active resources in the ZNS SSD.

4.1.4 Lack of Performance Isolation in Zone

Existing SSD interfaces and a ZNS SSD have difference models for the host responsibility and the I/O processing. A conventional SSD is a *black-box* entity whose device-internal condition depends on the past I/O execution history and hidden firmware logic. It is the storage system's responsibility to estimate its current I/O bandwidth capacity and schedule requests accordingly. A OC SSD is a *white-box* system. It exposes the underlying NAND flash memory and storage management functions directly to the host system, giving the host system full control over executing user I/O. This results in deterministic performance and the ability to adapt to specific workloads.

On the other hand, the ZNS SSD is a *semi-transparent* storage device to the host and should be viewed as a *gray box*. The interface holds much potential for reducing cost, improving performance, and developing device-independent systems. It handles wear-leveling, LBA-to-PPA mapping, and any NAND device constraints with the zone view hiding the device's geometry. However, a zone is not a completely performance-isolated domain, and co-located zones interact with each other in a non-deterministic fashion. As a consequence, performance unpredictability still rises, and inefficient use will cause sub-optimal performance. Ideally, each tenant should receive a weighted share based on the consolidation degree. Specifically, its housing application should achieve its targeted

performance when the SSD is under-utilized but receive a proportional degradation when the SSD is over-subscribed. But, unlike its predecessor OC SSD, ZNS SSDs manage zone allocation and wear-leveling internally with no strong isolation support and expose an opaque view to applications, yielding unpredictable performance interference and I/O execution unfairness. Such an issue could be mitigated in a conventional SSD where FTL and GC blend and distribute blocks across channels and dies uniformly regardless of the original command flow, ensuring the attainment of the maximum bandwidth and equal utilization of channel and die.

In summary, a ZNS SSD requires a systematic understanding of its capabilities and limitations so that one can efficiently integrate it into the storage application stack. One cannot directly carry over the observations and practices of using the conventional or OC SSD. The goal of this work is to develop a systematic understanding about these issues and propose potential solutions.

4.2 Performance Characterization of a ZNS SSD

This section characterizes a ZNS SSD with a focus on understanding why existing ZNS interfaces are static and inflexible. We then discuss the possibilities of addressing the problem.

4.2.1 Experimental Setup

Device HW Parameters	Specification
Capacity	3,816 GB
Channels #	16 Channels
NAND Dies #	128 Dies
NAND Page Size	16 KB
NAND Channel B/W	∼600 MB/s
Physical Zone Size	96 MB
Read B/W per Physical Zone	∼200 MB/s
Write B/W per Physical Zone	∼ 40 MB/s
Maximum Active Zones #	256

Table 4.1: The commodity ZNS SSD specification.

ZNS SSD and testbed. We use a commodity ZNS SSD for characterization. Table 4.1 presents its hardware details. It has 40,704 physical zones, where each 96MB-size zone consists of NAND erase blocks solely on a single die, and supports a maximum of 256 open zones simultaneously. We then configure various logical zones using such fine-granular units. We also prepare a conventional

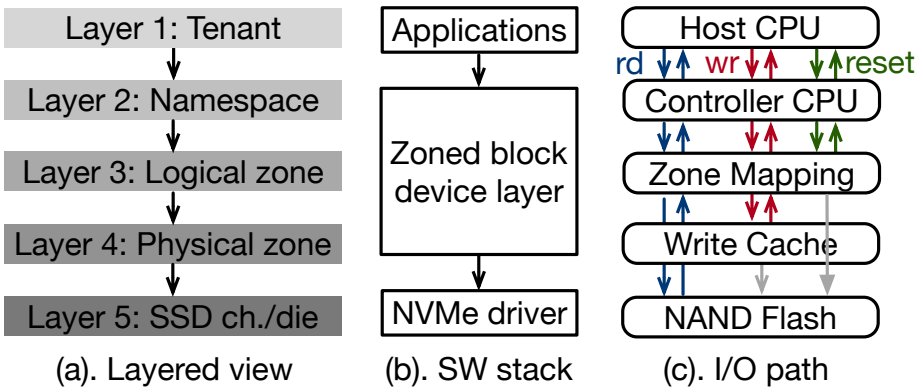


Figure 4.5: System model, SW stack, and I/O path of a multi-tenant ZNS SSD deployment. RD/WR=Read/Write. The write cache flushes data to the NAND flash asynchronously. Zone resets are completed after invalidating the mapping layer, where NAND blocks are erased lazily.

SSD with an equivalent architecture for a fair comparison. Our server has two 2.50GHz E5-2680v3 Xeon processors with 256GB DDR4 DRAM, and both SSDs are connected to ×4 PCIe Gen3 slots directly.

Workloads and performance metrics. We use the Fio benchmark tool [37] running on the SPDK framework [124] to generate synthetic workloads. We report both per-IO average/tail latency as well as achieved bandwidth. We add a thin layer to the SPDK to implement the logical zone concept and realize different zone configurations. Given the ZNS protocol, we regulate the write workloads to sequential accesses on a single logical zone in the following experiments, where read workloads issue random I/Os unless specified.

4.2.2 System Model

We consider a typical system setup with a five-layered view to facilitate the understanding of a multi-tenant ZNS SSD deployment and dissect the I/O behavior (Figure 4.5-a). From the top-down perspective, the first layer contains a few co-located tenants, each running a storage application (e.g., blob store, F2FS, and RocksDB). Next, a tenant exclusively owns one or several namespaces based on the required capacity. A namespace provides independently configurable logical zones (layer 3), exposing a private logical block address space. By manipulating the logical zone setup, a namespace can be configured differently to meet the capacity and parallelism requirements. Within a logical zone, reads happen everywhere, while writes are only issued in an append-only manner. This is unique to a ZNS SSD and in significant contrast to a conventional SSD, which can be viewed as a

fixed or statically configured SSD.

A logical zone comprises several physical zones (fourth layer). The number of physical zones per logical zone is typically fixed within a namespace. The logical-to-physical zone mapping can be arbitrary regardless of the request serving order and device occupancy. However, the logical zones must not share their physical zones to conform with the ZNS protocol. At the bottom layer, a physical zone is placed on one channel/die following the device specification. The zoned block device (ZBD) layer (Figure 4.5-b) is the central component across the storage stack that abstracts away architectural details of a ZNS SSD. It provides three functionalities: (1) interacting with the application on namespace/logical zone management; (2) orchestrating the logical-to-physical zone mapping in consideration of the application requirement; (3) scheduling a sequence of I/O commands to maximize device utilization and avoid head-of-line blocking. Figure 4.5-c shows the IO path of read/write/reset requests. We carefully configure each layer when designing characterization experiments.

4.2.3 Zone Striping

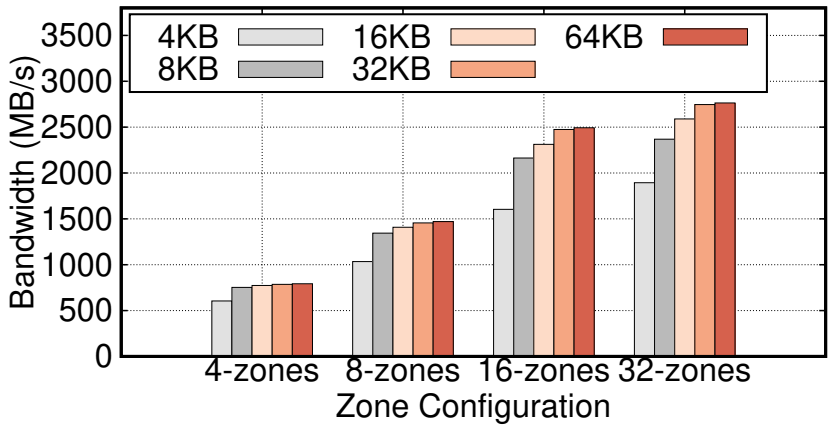


Figure 4.6: Read bandwidth varying the stripe size for different types of zones.

Since a logical zone is usually configured as an array of physical zones spatially, similar to RAID 0, one could apply the *striping* technique to achieve higher throughput, especially for large-sized I/Os. Zone striping segments data blocks across multiple physical zones and accesses them concurrently. There are two configuration parameters: (1) *Stripe size* is the smallest data placement unit in a

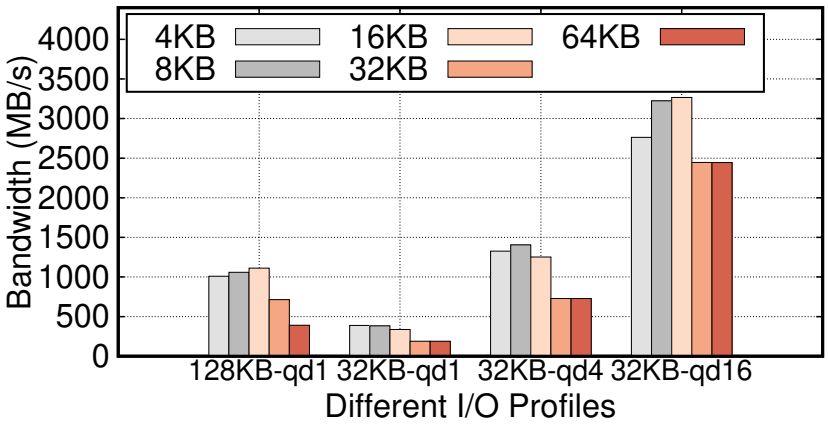


Figure 4.7: Read bandwidth varying the stripe size under the stripe width of 16.

Stripe Size	Avg. Lat(us)	P99.9 Lat. (us)	B/W (MB/s)
4KB	64	76	59
8KB	71	84	108
16KB	88	103	175
32KB	163	269	190
64KB	314	619	198

Table 4.2: Read I/O average/P99.9 latency and bandwidth varying the stripe size on a physical zone.

stripe, and (2) *Stripe width* defines the number of physical zones in an active state and controls the write bandwidth.

Basic Performance

When there are enough outstanding I/Os submitted to an SSD, unsurprisingly, the optimal striping efficiency is achieved when the stripe size matches the NAND operation unit (i.e., NAND page size). As shown in Table 4.2, the achieved per-die bandwidth increases slowly after the 16KB stripe size. In terms of latency, the access time reduction is non-linear for sizes smaller than a NAND page (16KB). When the I/O size is larger than 16KB, the average latency rises proportionally to the I/O unit because each request has to access the die multiple times sequentially. Next, we change the logical zone setup and see the efficiency of different stripe sizes. We use *N-zones* to refer to a logical zone configuration, where N is the number of physical zones in a striping. As shown in Figure 4.6, when issuing 2MB reads (which generates enough I/O to construct a full stripe I/O on each physical zone), for different zone configurations, the bandwidth over various stripe sizes shows

a similar result with the single-die performance. On the other hand, a wider width that fully uses the stripe size ($stripe\_size \times stripe\_width$) achieves higher bandwidth. For example, the 4KB stripe size in 8-zones achieves 37.3% higher read bandwidth than the 8KB stripe size in 4-zones. Note that the stripe size does not significantly affect the write performance as one can coalesce stripes on the same physical zone into a single device I/O and submit it at once. Instead, the stripe width determines the maximum write bandwidth.

Challenge #1: Application-agnostic Striping

When deciding the optimal stripe size and width, one should consider the application I/O profile dynamically, including request type, size distribution, I/O size efficiency, and concurrency. However, the existing zoned interface lacks such support and hinges on users' domain knowledge during configuration. A large stripe may hurt performance if the size of user I/O is smaller than that of a full stripe. On the other hand, too small a stripe also hurts the I/O efficiency of the device; a 4KB stripe with an 8-zone or wider width significantly lags behind 8KB or larger stripes in Figure 4.6. A wide stripe width sustains high performance per logical zone. However, since the device has a limited amount of active resources, it will instead limit the maximum number of active logical zones and jeopardize application concurrency.

Figure 4.7 presents the sustained read bandwidth varying the stripe size from 4KB to 64KB for four I/O profiles under the 16-zone configuration. For the 32KB I/O with a queue depth of 16 (32KB-QD16) and the 128KB synchronous I/O (128KB-QD1), the 16KB stripe performs the best because NAND operates most efficiently when the stripe size aligns with the NAND page size. On the other hand, for the 32KB with 1 and 4 queue depth cases (32KB-QD1, 32KB-QD4), 4KB/8KB stripes outperform the 16KB one by 15.1%/13.6% and 5.8%/12.2%, respectively, because they activate significantly more channels and dies for a single user I/O. The larger stripe sizes, 32KB and 64KB, fail to explore any parallelism, resulting in poor performance due to narrower user I/Os. Overall, the results demonstrate the importance of leveraging more parallelism, particularly for small I/O sizes. While a large I/O may encounter inefficiencies in NAND operations due to a small stripe size, it can be optimized through simple techniques such as merging adjacent I/Os within the same physical zone.

Observation: The use of logical zones with striping is beneficial for the application, but the

zone stripe width and size should be considered in combination with each other. However, it is essential to carefully consider both the zone stripe width and size in tandem. It's not desirable to have a stripe size larger than the NAND page size, and adjustments should be made to the stripe size when widening the stripe width to provide the most parallelism for user I/O of a specific size.

4.2.4 Zone Allocation and Placement

A ZNS SSD allocates physical zones across dies/channels, mainly taking access parallelism and wear-leveling into consideration. Upon an allocation request, the ZNS SSD traverses the die array following a certain order and then selects the next available die to place each physical zone. Within a determined die, it chooses blocks with the least P/E cycles based on opaque wear-leveling policies.

Basic Performance

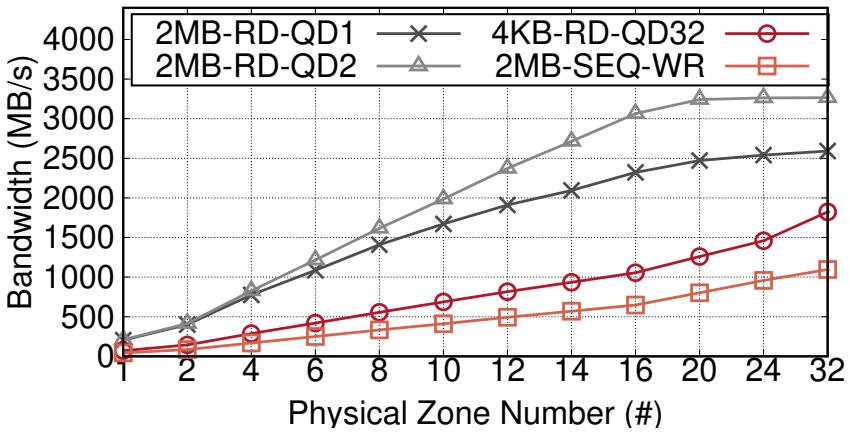


Figure 4.8: Read/Write bandwidth varying the number of physical zones.

Zone allocation should be locality-aware and parallelism-aware. A larger-sized logical zone is expected to observe higher read/write bandwidth because it spreads physical zones across *different* channels and dies in a deterministic sequence and achieves more I/O parallelism. The maximum performance is obtained when I/Os access all channels and dies without blocking. We configure the stripe size to 16KB and increase the number of physical zones in a logical zone (N), then measure the I/O bandwidth of a single logical zone under four I/O profiles (Figure 4.8). The performance of 2MB reads with queue depths 1 and 2 (i.e., 2MB-RD-QD1/2MB-RD-QD2) keeps increasing until the number of physical zones approaches 20. But they max out for different reasons. The QD2 case

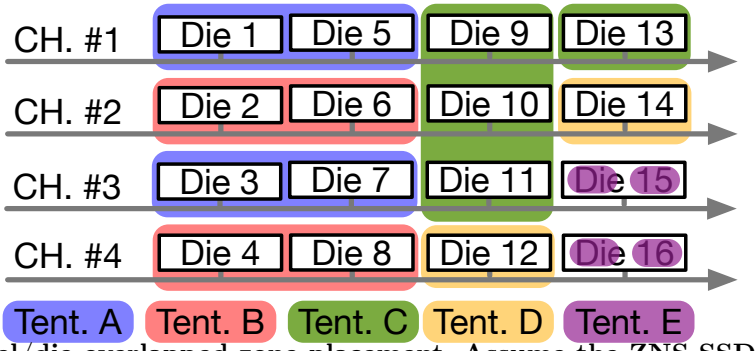


Figure 4.9: Channel/die-overlapped zone placement. Assume the ZNS SSD has 16 dies across 4 channels and each tenant has its own namespace. In the case (1), the SSD serves two allocations from tenants A and B simultaneously (both asking for 4 physical zones). Due to the overlapped placement, both A and B benefit from two channels. Similarly, in the case (2), the SSD serves four requests (which allocate 3 zones, 1 zone, 1 zones, 1 zone) from tenants C and D asynchronously. Because of the overlapped placement, tenants C and D obtain three and two channels, respectively. In the last case (3), tenant E allocates four zones spanning across two dies, limited by the die bandwidth.

is bounded by the PCIe bandwidth (i.e., four Gen3 lanes or 3.2GB/s), whilst the QD1 scenario is simply limited by the application as it cannot issue enough outstanding I/Os at that queue depth. In terms of 4KB random read with 32 queue depth and 2MB sequential write, they sustain 80MB/s read and 40MB/s program bandwidth per physical die, respectively, requiring much more physical zones ($\sim$ 40 and 80) to utilize the channel or PCIe bandwidth fully.

Challenge #2: Device-agnostic Placement

An ideal allocation process should expose all of the internal I/O parallelism of a ZNS SSD to a tenant. However, the existing mechanism is opaque to housed tenants, where the global allocation pointer picks the next available die without considering the application's prior allocation history or how it interacts with other tenants. This causes unbalanced zone placement, hurts I/O parallelism, and jeopardizes performance. We find two types of inefficient placements:

- **Channel-overlapped placement:** As shown in the case (1) and (2) of Figure 4.9, concurrent zone allocations might cause overlapped zone placements across channels, limiting the maximum channel parallelism. Similarly, synchronized allocation requests might prevent placement alignment, again limiting the aggregated bandwidth. Figure 4.10 presents 4KB and 128KB random read bandwidth when increasing the QD for three inferior placements, where 2/4/8 physical zones contend for the same channel in a 16-zone configuration. Physical zones stay across 16 different dies that limit the maximum bandwidth. The 2-overlapped allocation outperforms the other two

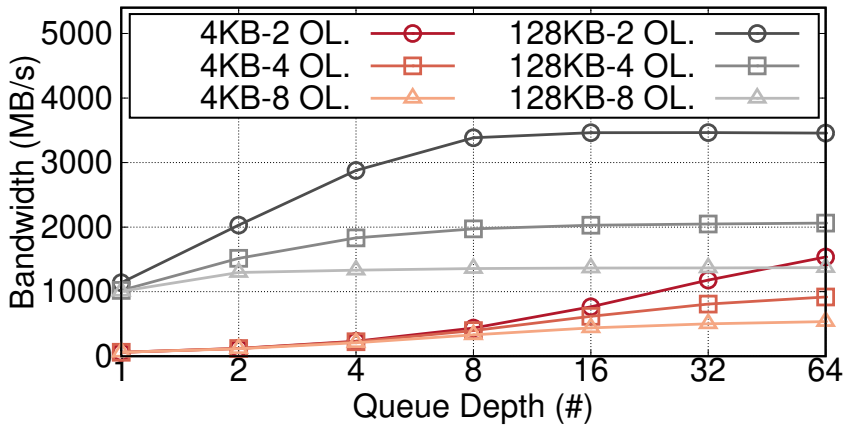


Figure 4.10: Read bandwidth under three channel overlapping (OL) allocations.

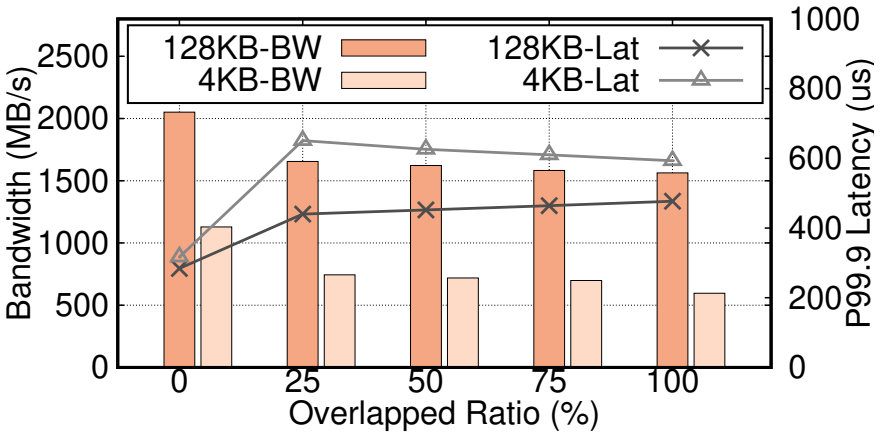


Figure 4.11: Bandwidth and tail latency varying with the die overlapping ratio.

(i.e., 4-overlapped/8-overlapped) by 1.7×/2.9× and 1.7×/2.5× for 4KB and 128KB cases, respectively.

- **Die-overlapped placement:** The case (3) of Figure 4.9 describes this issue. An intra-namespace die overlapped placement limits the bandwidth and can be even more detrimental because a die can only process one operation at a time. We configure such an experiment by placing physical zones in the same die and gradually increasing the overlapping ratio. Figure 4.11 reports the logical zone's sustained bandwidth and tail latency under two I/O profiles. When no physical zones share the same die, it achieves 1,128MB/s and 2,051MB/s along with 317us and 284us p99.9 tail latency for the 4KB random read and 128KB sequential read cases, respectively. With full overlap, we observe 47.2%/23.8% bandwidth drop and 87.1%/28.0% tail latency increase. Such performance degradation happens even when the overlapping ratio is lower than 25%, because both types of

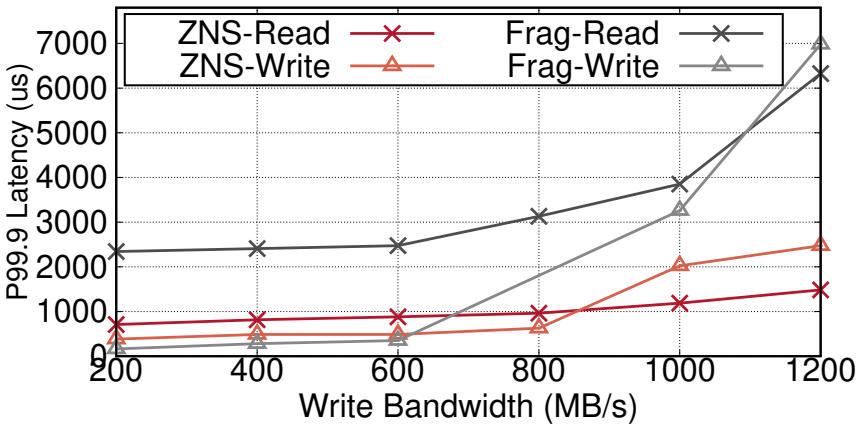


Figure 4.12: Read tail latency varying the write bandwidth (ZNS vs Conventional SSD)

I/Os suffer from the head-of-line blocking issue at the overlapped dies.

Observation: It is challenging to infer the zone's physical location without knowing the device's internal specification. One may run a profiling tool in the runtime to extract the relation among different zones [11]. However, it does not eliminate the imprinted overlap at the allocation time. To maximize the I/O parallelism, one could build a device abstraction layer that (1) relies on a general allocation model of the device; (2) maintains a shadow view of the underlying physical device; (3) profiles its placement balanced level across different physical channels and dies.

4.2.5 I/O Execution under ZNS SSDs

A ZNS SSD eradicates background GC I/Os, thereby removing one form of performance non-determinism. Within a logical zone, writes happen sequentially, but reads are issued arbitrarily. When reads are congested, one would observe latency spikes under die/channel contention. If considering cross-zone cases, either *intra* or *inter* namespace, interference would be more severe than a conventional SSD because ZNS SSDs impose no physical resource partitions, and per die/channel bandwidth is narrow.

Basic Performance

Irrespective of the NAND block layout of a logical zone, its I/O access latency highly correlates with achieved bandwidth because there are no device internal I/Os that consume bandwidth and are hidden from user applications. To demonstrate this, we prepare a conventional SSD having the same hardware as the ZNS SSD and compare two SSDs under the mixed read-write scenario. We

configure a logical zone for the ZNS SSD that spreads across all the channels and dies (i.e., 128-zone configuration with 16KB stripe size) to match the conventional one. The fragmented conventional SSD is 70% filled and preconditioned with 128KB random writes. Then we run eight read threads– where each issues one 128KB read I/O to all the dies uniformly random–and one write thread that performs sequential write at a fixed rate. Figure 4.12 reports the read/write tail latency as we increase the write bandwidth. More writes on a ZNS SSD leave less bandwidth headroom for reads and cause the latency to increase. However, for the fragmented conventional SSD, the internal GC activities make even less bandwidth available to serve reads due to write amplification. For example, when the write bandwidth is 1,000MB/s, the p99.9 read and write latency of the conventional SSD is 4.3× and 2.8× worse than the ZNS one. In terms of the read throughput, the conventional SSD shows 1.1× and 1.6× lower throughput than the ZNS SSD at the 200MB/s and 1,000MB/s write bandwidth, respectively.

Challenge #3: Tenant-agnostic Scheduling

Existing zoned interfaces of ZNS SSDs provide little performance isolation and fairness guarantees for the inter-zone case, regardless of deployed workloads. One cannot overlook the read interference on a die because (1) an arbitrary number of zones can collide on a die, (2) the bandwidth of a single die is poor, and hence, the interference becomes severe even under a very low load on the device, and (3) it causes a severe head of line blocking problem and degrades the performance of the logical zone. Since there is no internal GC in the ZNS SSD, The I/O determinism [76] proposed for the conventional SSD does not apply as well. Similar to conventional SSDs, the write cache, shared among all NAND dies, is an indispensable component of the ZNS SSD, buffering incoming writes and flushing to the NAND dies in a batch. Host applications will observe prompt write I/O completions when they are absorbed by the cache but experience considerable latency spikes when the cache overflows. This has not been an intractable issue in conventional SSDs because the device firmware blends all incoming write I/Os and constructs a single large flow spanning entire NAND dies, maintaining the cache eviction rate to the maximum device bandwidth. However, in the ZNS SSD, a write I/O must be flushed out to the designated NAND die with an inadequate program bandwidth, even with zone striping. In this situation, a heavy writer exhausts the available cache capacity and severely disturbs other short flows.

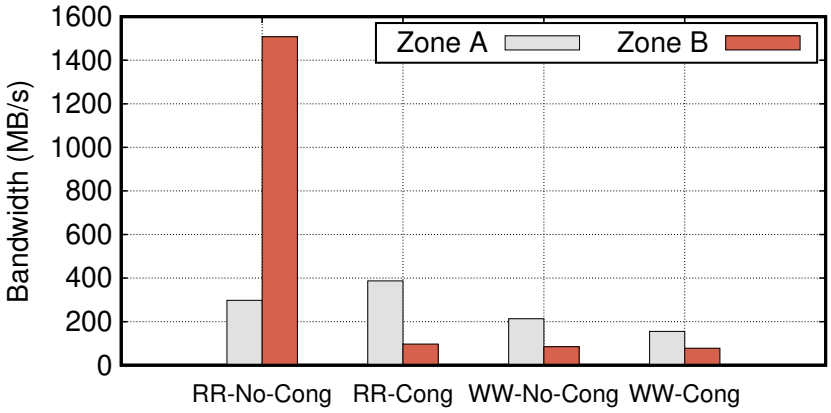


Figure 4.13: Bandwidth under RD-RD and WR-WR congestion due to the die-collision.

We set up two readers performing 128KB read I/O in different profiles: (1) queue depth 8 with a two-zone configuration and (2) queue depth 2 with an eight-zone configuration. Figure 4.13 shows the interference between two readers in a die-collision. The QD-8 reader easily obtains 97.2% of the total bandwidth of collision dies. Note that the interference and unfair bandwidth share also occurs in the conventional one, but only when the device bandwidth is fully saturated [117, 66]. We also demonstrate the write cache congestion in Figure 4.13. We first populate 15 logical zones with a stripe width of 8, and each physical zone is allocated to a dedicated die. The cumulative write bandwidth of 15 zones maxes out the PCIe bandwidth (3.2GB/s), and a single zone performs at ∼213.3MB/s. In this case, a physical zone in the logical zone receives write at a lower rate than the maximum bandwidth (∼26.7MB/s), and the write cache does not overflow. Then, we add one more writer with a narrow width of 2, which also runs on dedicated dies. Write I/Os towards the narrow zone are equally fetched by the device, but it soon consumes all available cache because of the scarce bandwidth (∼85MB/s) of underlying physical zones. It degrades others' bandwidth by 27.3% or 155MB/s, and the device even fails to max out the PCIe bandwidth (∼2.4GB/s).

We set up three namespaces with different zone configurations (i.e., 4-zones, 8-zones, and 16-zones) and ensured they have the same capacity using all the channels and dies. Figure 4.14 shows the achieved bandwidth when these three namespaces run the same type of workload. In terms of the 128KB write (QD=1), namespaces 2 and 3 achieve twice the bandwidth as namespace 1 because the maximum sequential write I/O is bounded to four channels in the 4-zone case. Since the stripe size is 16KB, a 128KB write generates 8 outstanding I/Os that can harness 8 dies at best,

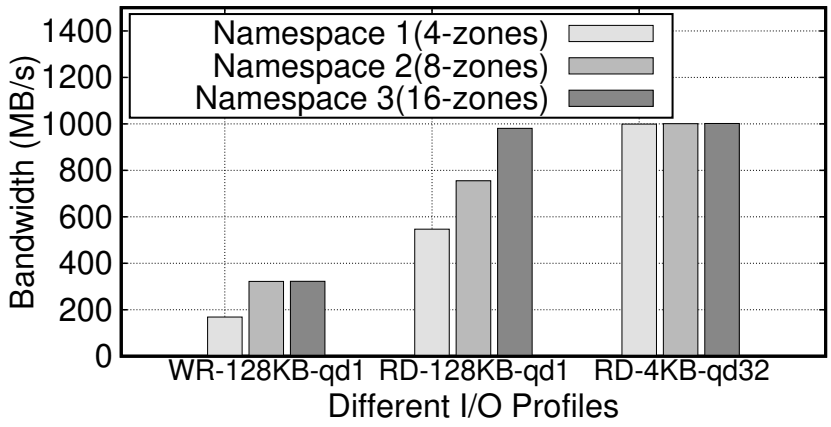


Figure 4.14: Performance comparison of three namespaces (with different logical zones) running the same workload.

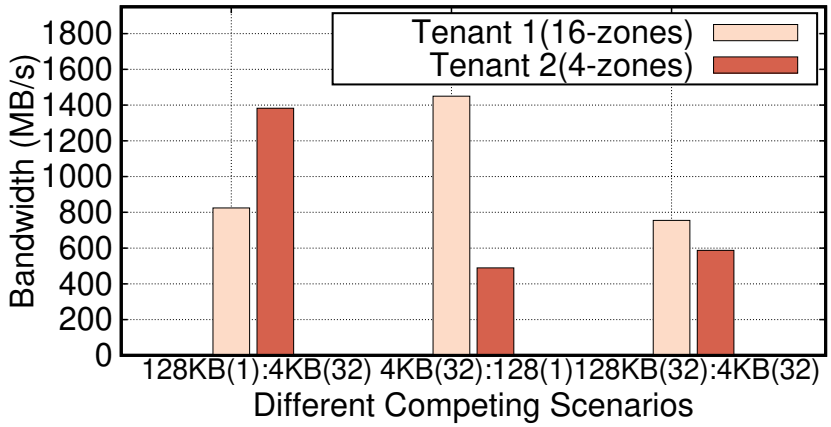


Figure 4.15: Performance comparison of two tenants (with different logical zones) running different workloads.

leading to the same performance of namespace 2 and 3. Considering the other two read scenarios, even though three namespaces submit the same amount of I/Os, we observe different bandwidth shares proportional to their underlying parallelism in the logical zone for large 128KB sequential reads. However, the three namespaces show the same bandwidth for a small 4KB I/O with 32 queue depth because each random I/O could fall into any of the channels and dies regardless of the zone configuration. Next, we consolidate two namespaces and issue different types of I/Os. Figure 4.15 reports the results, where the number in parentheses indicates the queue depth. When two tenants compete for the read bandwidth, irrespective of the logical zone configuration, the running workload that can utilize more dies achieves higher bandwidth, as the 4KB (QD=32) in the first two scenarios and 128KB (QD=32) in the third case.

ZNS v.s. Conventional SSDs

There is no performance isolation support on a conventional SSD. Some recent proposals employ I/O determinism and NVM Sets [70, 76, 99] to minimize the interference. Although they mitigate the interference in a given environment, such mechanisms have a restriction due to static partitioning (NVM Sets) or limited predictable time window (I/O Determinism). Instead, OC SSDs grant full access to the device geometry to applications, and thus, one can predict possible interference using a complete view of physical allocation. However, one cannot prevent interference even with this knowledge because read I/Os may target any of the dies associated with a zone unpredictably. Some researchers propose a read reconstruction using redundant arrays [80, 139], but such mechanisms sacrifice both capacity and bandwidth for reads and fail to address a multi-tenant setup in a modern data center.

Observation: When using ZNS SSDs in a multi-tenant scenario, one should first understand how different namespaces and logical zones share the channels and NAND dies of the underlying device, classify their relationships into competing and cooperative types, and employ a congestion avoidance scheme for the inter-zone scenario to achieve fairness. Since there are no device bookkeeping operations, I/O latencies represent the congestion level on colliding dies. In addition, write cache congestion needs to be addressed globally. Thus, a possible solution is to design (1) a global central arbiter that decides the bandwidth share among all active zones; (2) a per-zone I/O scheduler that orchestrates the read I/O submission based on the congestion level.

4.2.6 Zone Reset

Reset #	Avg. Lat(us)	P99.9 Lat. (us)	B/W (GB/s)
1	204	233	455
2	220	338	841
3	239	449	1158
4	275	445	1342
5	313	889	1478
6	381	2343	1456

Table 4.3: Reset average/P99.9 latency and bandwidth varying the number of concurrent reset operations.

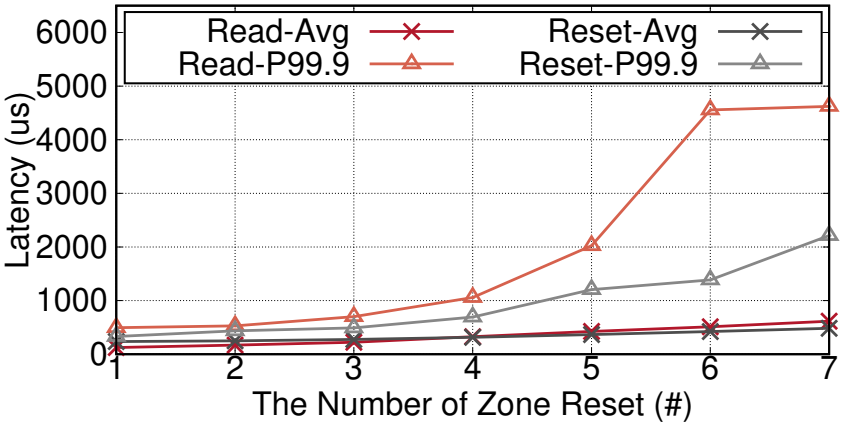


Figure 4.16: Read v.s. reset latency varying the number of reset operations.

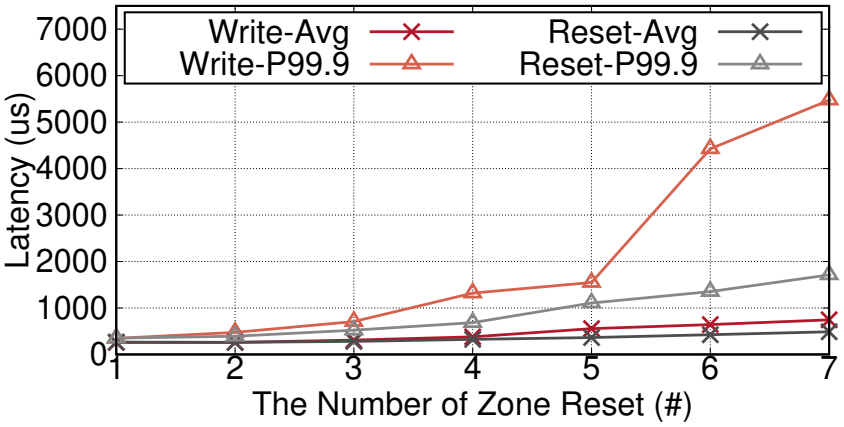


Figure 4.17: Write v.s. reset latency varying the number of reset operations.

ZNS SSDs delegate NAND block lifespan management responsibility to host applications via the RESET command. It is a unique command in ZNS and OC SSDs, which a conventional one lacks. The conventional SSD provides a DEALLOCATE command instead, but it only invalidates logical pages without triggering the block erase activity explicitly. ZNS SSDs further optimize a reset latency using the zone mapping layer while providing the same execution logic on block erase as OC SSDs. When processing this request, a zone will transition to the EMPTY state, where its write pointer is redirected to the start LBA. All associated NAND blocks are erased, and previously written data becomes inaccessible. This allows applications to develop application-specific GC mechanisms, whose goals should be minimizing the impact on concurrent read/write traffic as well as reclaiming stale data timely.

We find that the zone reset is essentially a lightweight command whose execution latency is

much lower than the block erase operation. This is mainly because the firmware sends back the reset completion acknowledgement as soon as the zone mapping is cleared and performs the actual erasure operation lazily. We gradually increase the number of concurrent reset operations and measure the latency and bandwidth of resetting a physical zone. As shown in Table 4.3, its unloaded latency is around 204us, whereas erasing a block takes around 3.5ms [56]. Also, reset is a high-bandwidth operation, achieving 1478GB/s at max. This is reasonable as the per-die erase bandwidth of modern high-density NAND devices is higher than 10GB/s.

Unsurprisingly, read and write I/Os interfere with resets severely. Although the reset bandwidth is extremely high, each operation takes significantly longer than blocking read and write. We use the experiment setup as described above and co-locate it with a 4KB random read (QD=32) or a 128KB write (QD=1), respectively. In terms of the read v.s. reset interference scenario (Figure 4.16), the read average and p99.9 latencies rise by 5.0× and 9.4× when there are 7 concurrent resets. This also causes a significant bandwidth degradation (from 1492MB/s to 202MB/s). Regarding the write vs. reset (Figure 4.17), under the most interfering case, we observe that the write average/p99.9 latencies increase by 2.9×/15.8×, along with a 2.8× bandwidth drop. Thus, zone reset is another interference factor that jeopardizes the I/O predictability. For example, an irresponsible tenant could continuously submit reset operations and break the GC-free illusion of other victim users when they share the same NAND die. Therefore, zone reset should be coordinated globally across co-located tenants.

Observations: The zone reset should be viewed as another type of user I/O command (in addition to read/write) for scheduling with sufficient bandwidth. It is not necessary to process a reset immediately as the bandwidth is hundreds of times higher than write. Instead, one can invalidate a zone immediately and complete the user request, then coordinate zone reset in a global and batched fashion across co-located tenants to minimize the chance of collision with read and write.

4.2.7 Write Cache

Write cache is an indispensable component of NAND-based SSDs for two reasons: (1) There is a mismatch between the LBA size and NAND page size, and the programming is not an atomic operation; (2) SSDs should follow strict timing rules [22] (such as the timing among upper/middle/lower

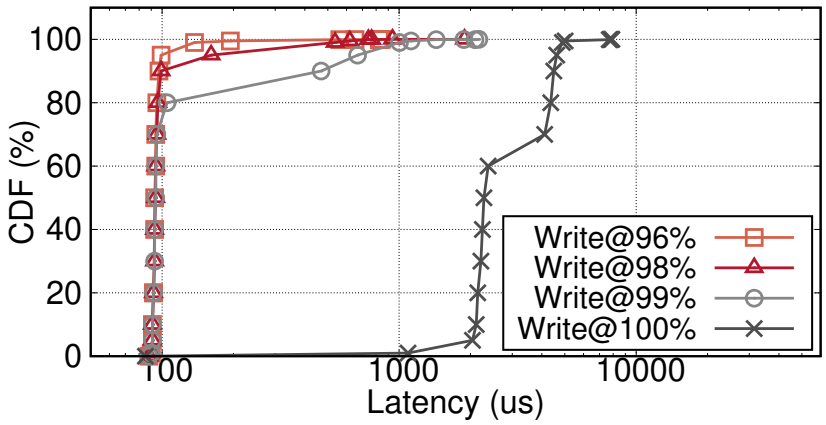


Figure 4.18: CDF of write I/O latency for different traffic loads. x-axis is log scale.

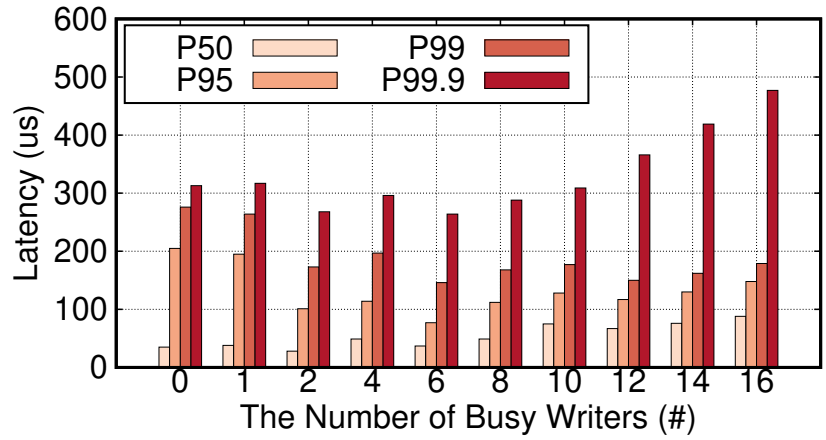


Figure 4.19: Latency varying with the number of busy writers on a conventional SSD.

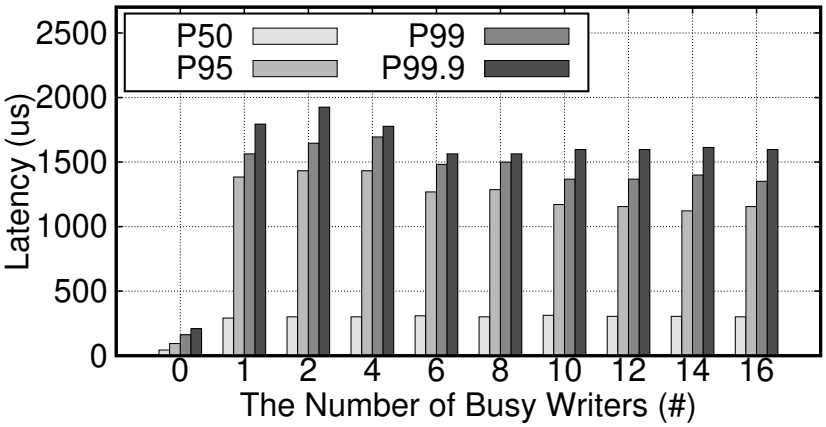


Figure 4.20: Latency varying with the number of busy writers on a ZNS SSD.

pages) to ensure data integrity. Consequently, writes are buffered temporarily in the write cache until the accumulated buffer size is sufficient to complete a programming sequence. Hence, host

applications will observe prompt write I/O completions when they are hitting in the cache until over-committing write I/Os.

ZNS SSDs also benefit from the write cache under the modest write I/O traffic. We set up an experiment that writes to a single physical die at different rates. We configure four cases where the first three issue 16KB write I/Os at a rate lower than the maximum die serving bandwidth (Table 4.1) and the last one maxes out. Figure 4.18 reports the CDF of the I/O latency. When the write traffic is less than the maximum, the write cache can absorb the majority of the I/Os and provide fast write completion (i.e., p50 is 93us). However, when the write runs at the maximum, we observe a significant latency increase because it has to wait until the SSD flushes data and reclaims cache space. The p50 latency rises to 2278us, which is even worse than the p99.9 latency of the first three scenarios (i.e., 562us, 742us, 1434us).

A busy writer can adversely impact all concurrent writer flows, leading to elevated tail latency. This issue primarily arises from the head-of-line-blocking effect induced within the SSD controller. Unlike conventional SSDs, which mitigate this issue by employing striping that spans all dies, ZNS SSDs face unique challenges. In this context, conventional SSDs reclaim cache space at the maximum write bandwidth, and write latency remains stable unless the overall write demand surpasses the device's capability. However, ZNS SSDs exhibit a narrower write bandwidth per zone and can experience significant delays in the cache when an excessive number of write I/Os are overcommitted to a single zone.

To demonstrate this, we set up 32 writers issuing 16KB (QD1) write I/O with different manners: (1) an unloaded writer that submits at a fixed rate lower than the maximum die bandwidth (20MB/s), and (2) a busy writer that runs without capping bandwidth. Then, we measure the 16KB write I/O latency of an unloaded tenant as increasing the number of co-located busy writers. Figure 4.20 demonstrates the high tail latency in the ZNS SSD. Adding a single busy writer will cause the p50/p95/p99/p99.9 latency to increase by 5.6×/13.7×/8.7×/7.6×, respectively. More writers will not further jeopardize the case. Instead, we find that the tail latency will reduce slightly as the write cache is flushed in higher bandwidth as more dies are busy, e.g., 18.4% drop of p99.9 under 16 writers. We conduct the same experiment on a conventional SSD in Figure 4.19. Since the cache is shared among all writes and reclaimed at the maximum bandwidth, we observe that p50/p95/p99 latencies stay quite stable, while the p99.9 latency only increases by 50.5%.

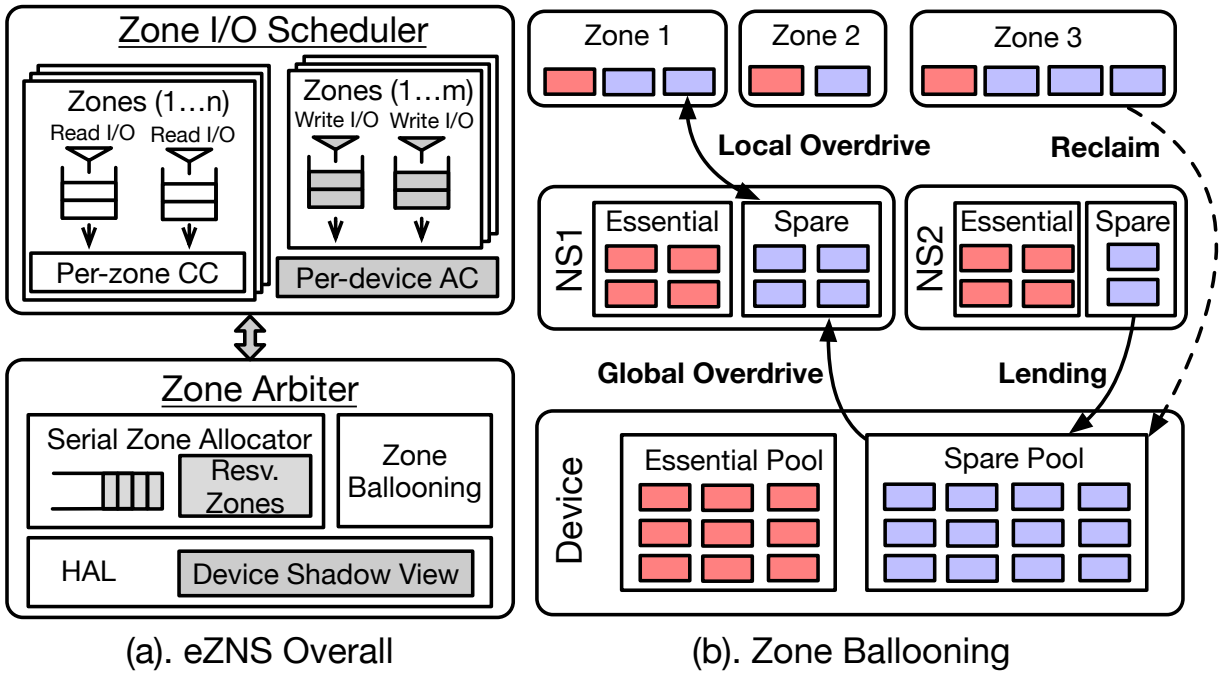


Figure 4.21: eZNS System Architecture.

Observations: Write cache is a hidden performance domain in ZNS SSDs. Unlike the conventional SSD, where the write bandwidth is shared among every write, the ZNS SSD shares the cache among all zones. By carefully pacing the I/O rate from each one, one can mitigate the tail latency and achieve a fair write cache share. This could be realized via a software-managed I/O scheduler using a monitoring mechanism for cache occupancy.

4.3 eZNS: Enabling an Adaptive Zoned NS

This section describes the design and implementation of eZNS that realizes a new and elastic zoned interface. We use the gathered insights from our characterization experiments and address the aforementioned issues.

4.3.1 eZNS Overview

eZNS stays atop the NVMe driver and provides raw block accesses. eZNS exposes the *v-zone* interface that offers runtime hardware adaptiveness, application elasticity, and tenant awareness. We carefully design eZNS and spread its functionalities across the control plane and data plane. As shown in Figure 4.21, it mainly consists of two components. The first is the zone arbiter that (1)

maintains the device shadow view in a hardware abstraction layer (HAL) and provides the basis for other components, (2) performs serialized zone allocation avoiding overlapped placement, and (3) dynamically scales the zone hardware resources and I/O configurations via a harvesting mechanism. The second is a tenant-cognizant I/O scheduler, orchestrating read requests using a delay-based congestion control mechanism and regulating writes through a token-based admission control. In sum, eZNS addresses the three issues discussed in §4.2.

4.3.2 Hardware Contract and HAL

We develop eZNS based on the following hardware contract, which are met by recent ZNS SSDs with small zones: (1) a physical zone consists of one or more erasure blocks on a single die; (2) the maximum number of active physical zones is a multiple of the number of dies, and all dies hold the same number of active zones when they are fully populated (i.e., the ZNS SSD evenly distributes physical zones over dies); (3) the zone allocation mechanism follows the wear-leveling requirements, indicating that consecutive allocated zones will not overlap on a physical die until all the dies have been traversed. We need to caveat that the last contract may not always be followed in allocations if the device firmware enforces a specific policy other than round-robin across dies. However, considering the large number of chips and the wear-leveling constraint, such cases are rare. Our mechanism doesn't require being cognizant of the two-dimensional geometric physical view of SSD NAND dies and channels or maintaining an exact zone-die mapping.

eZNS maintains a shadow device view, exposing the approximate data locality for zone allocation and I/O scheduling. Our mechanism (or HAL layer) only hinges on three hardware parameters from device specifications. The first one is the *maximum number of active zones* (or MAR, maximum active resources). This is based on an observation that the MAR is generally in proportion to or a multiple of the number of physical dies on the SSD. One could estimate the number of active zones that a die could hold by deliberately controlling the zone allocation order in an offline calibration experiment (§4.2.4). The second parameter required is the *NAND page size* used for striping configuration. For example, 16KB is a de facto standard for most TLC NVMe drives and is well-known for system developers. The SSD shows the best efficiency when the stripe size is aligned with it (§4.2.3), and thereby, we choose the stripe size as a multiple or factor of the NAND page size that is closest to avoid inefficient stripe reads for sequential workloads. These two parameters reflect the

device's capabilities. The third one is the *physical zone size*, deciding how a logical zone and strip groups are constructed. With such information, HAL provides a shadow view having a consistent MAR (e.g., 16) and the size of a zone (e.g., 2GB) regardless of the underlying device.

4.3.3 Serial Zone Allocator

eZNS develops a simple zone allocator that provides three guarantees: (1) it ensures that each stripe group comprises a list of consecutive and serial opened physical zones, following the firmware-enforced internal order; (2) there is no die collision within a stripe group; (3) across stripe groups, die collision could happen for writes only if available active physical zones are fully populated across all the dies. Given the above device model, the number of stripe groups colliding on a die is $\frac{Maximum\ \#\ of\ active\ zones}{Die\ \#}$ at most. Channel collision would not be an issue because its bandwidth is usually higher than the aggregated program bandwidth across dies.

Our allocator works as follows. It has a per-device request queue, buffering OPEN commands (including implicit ones followed by writes) from all logical zones. Our allocator serves each logical zone request atomically. Since the completion of a zone OPEN command does not guarantee that the zone is actually allocated on a physical die, we implement a zone reservation mechanism during zone opens–flushing one data block that enforces binding a die to the zone. Writes complete immediately as the write cache of the device absorbs a single block even in high load. To expedite this process, we proactively maintain a certain amount of reserved zones in serial order and provision them to an upcoming stripe group. Upon completion of the allocation, we then update the allocation history and write it into a reserved persistent region (metadata block) following the block for reservation. Hence, we preclude interleaved allocations from concurrently opened logical zones to prevent channel-overlapped placement and facilitate allocation reordering to mitigate die overlaps (§4.2.4).

4.3.4 Zone Ballooning

v-zone, a specialized logical zone, can automatically scale its I/O striping configuration and hardware resources to match changing application requirements in a lightweight fashion. Figure 4.22 illustrates an example of a *v-zone* structure. Similar to a static logical zone, a *v-zone* contains a fixed number of physical zones. However, unlike a static logical zone, it divides physical zones into one or more stripe groups. When *v-zone* is first opened or reaches the end of a previous stripe group, it allocates

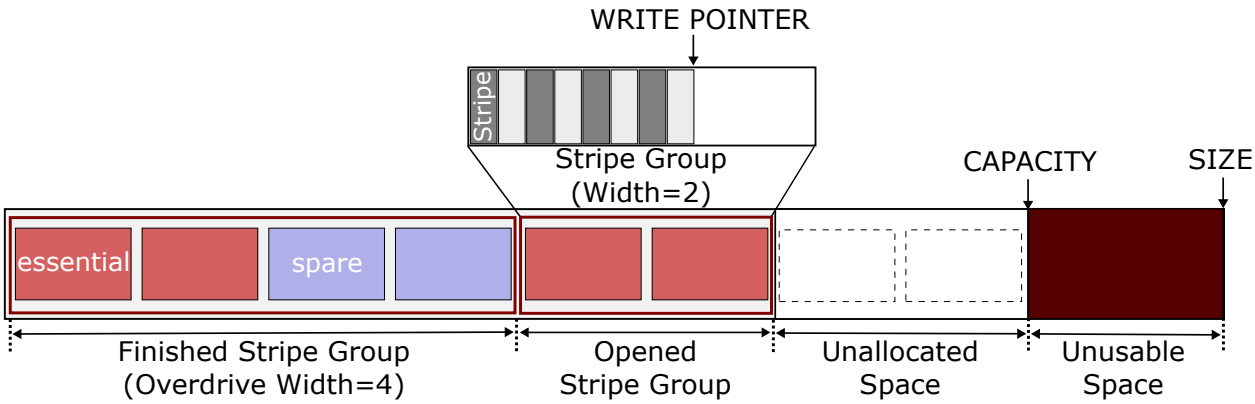


Figure 4.22: Example of eZNS *v-zone* structure.

a new stripe group. All physical zones in the previous stripe group must be finished when the write pointer reaches the end of the stripe group, allowing an active *v-zone* to take active resources for only one stripe group. The number of physical zones in a stripe group is determined at the time of allocation according to the *local overdrive* mechanism, which enables flexible zone striping. To comply with the standard zone interface, *v-zone* has a size that is a power of 2, and its capacity is the sum of user-available bytes in physical zones.

Similar to the virtualization memory ballooning technique [132, 13, 111], zone ballooning allows a *v-zone* to (1) expand its stripe width by leasing spares from others when other namespaces are under low active resource usage; (2) return them when it finishes the stripe group either by writing to the end of the stripe group or explicitly issuing FINISH/RESET commands from the application.

Initial Resource Provisioning

eZNS divides all the available and opened physical zones on the ZNS SSD into two groups: *essential* and *spare*. The *essential* group contains a minimal number of active physical zones that can max out the SSD write bandwidth ($N_{essential}$), whilst the rest belong to the *spare* group (N_{spare}). Our initial resource allocation follows the equal bandwidth partition principle. We choose the write I/O bandwidth as the minimum guarantee because writing resources (or active physical zones) of a ZNS SSD are scarce. Assuming the number of namespaces that a ZNS SSD holds is N_{ns} and the maximum number of active *v-zone*s per namespace is $MAR_{logical}$. A namespace takes $\frac{N_{essential}}{N_{ns}}$ exclusive active physical zones; when a *v-zone* in the namespace opens a new stripe group, it receives $\frac{N_{essential}}{N_{ns} \times MAR_{logical}}$ assured essential ones which is also the minimum stripe width. In terms of *spare* zones, similarly, eZNS equally distributes them to a namespace ($\frac{N_{spare}}{N_{ns}}$) during initialization. Both a *v-zone* and a

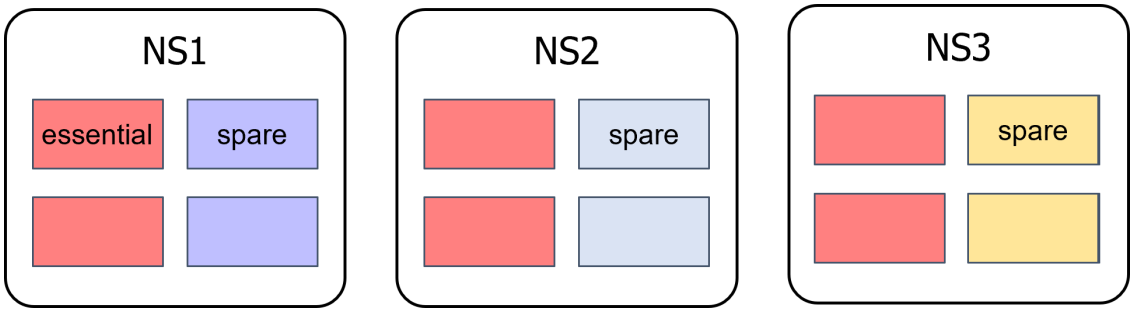
namespace will expand/shrink their capacity to adapt to workload demands.

Local Overdrive: Zone Expanding

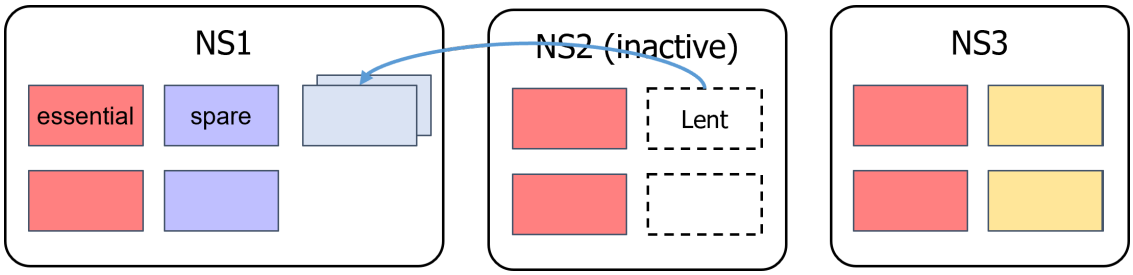
eZNS provisions available spares from the *spare* group of its namespace to boost its write I/O capability. We realize this via an internal *local overdrive* operation while opening a new stripe group. The mechanism works as follows. First, it estimates the resource usage of the namespace by analyzing its previously opened *v-zone*s, quantified as the exponentially weighted moving average over the number of active *v-zone*s ($N_{ActiveZoneHistory}$). Second, it checks the remaining spares from the spare group ($N_{RemainingSpare}$) and reaps additional spares based on $\frac{N_{TotalSpare}}{N_{ActiveZoneHistory}}$. Essentially, a *v-zone* will receive more (fewer) spares if it embodies writing activities but the namespace only opens fewer (more) *v-zone*s. Third, the *v-zone* conflates the harvested spares with assured essential ones for it to open the new stripe group, and the stripe width is rounded down to the nearest power of two for efficient resource management. Note that the *local overdrive* operates in a serial and best-effort fashion. Lastly, eZNS sets the baseline stripe size to 32KB at the minimum width for the optimal I/O efficiency of the device. It then reduces the stripe size for an overdriven zone according to the stripe width, down to the minimum block size of the device. For example, if the width gets two times wider, the stripe size is reduced by half. We determine the range of stripe sizes to optimize the performance as aforementioned in §4.2.3. The reduced stripe size further contributes to the I/O scheduler ensuring fairness (§4.3.5).

Global Overdrive: Namespace Expanding

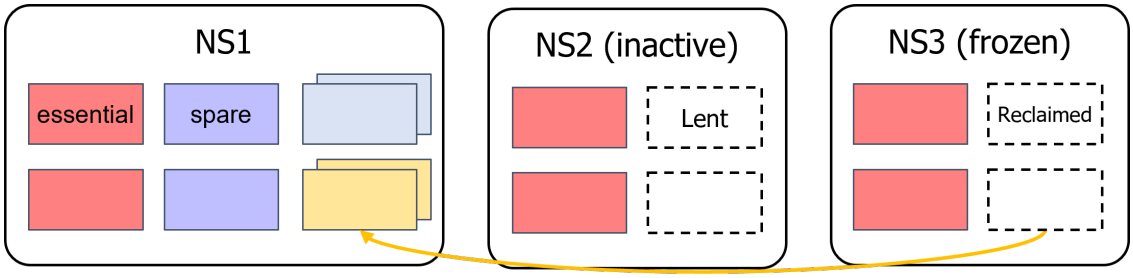
Across the whole device, our zone ballooning mechanism further reallocates spares across namespaces based on their latest write activity. We realize this via another internal *global overdrive* operation– lend spares from the spare group to each other. Unlike *local overdrive*, global overdrive is triggered based on the write intensity across the entire drive. Specifically, our arbiter monitors the past $N_{essential}$ opened physical zones across all active namespaces, computes their zone utilization, and redistributes the remaining spares from inactive namespaces to active ones. Figure 4.23 illustrates an example case of *global overdrive*. Initially, in Figure 4.23a, we evenly distribute essentials and spares across three namespaces. Subsequently, NS1 and NS3 open two *v-zone*s each. While NS1 continues to actively write data, NS3 becomes inactive after opening its allocated *v-zone*s. Simultaneously, NS2 remains devoid of write activity. Once the arbiter identifies NS2 as an *inactive* namespace,



(a) Initial configuration



(b) Lent unused spare zones from inactive NS2 to NS1



(c) Spares are reclaimed from frozen NS3 and lent to NS1

Figure 4.23: Three simple examples of the global overdrive mechanism

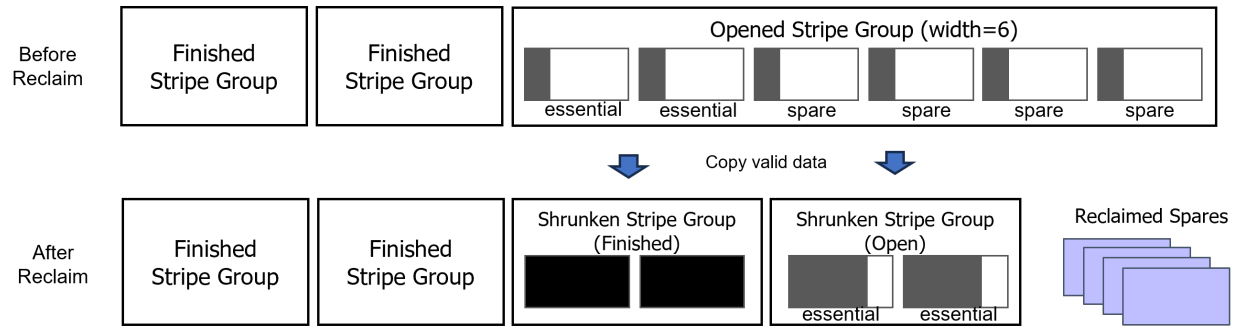


Figure 4.24: Example of the zone reclaim operation. The original stripe group has width of 6 with 4 spares. The reclaim operation shrinks it to the minimum width of 2 using only essentials. When the amount of valid data exceeds the size of a minimum width stripe group, it finishes the stripe group and creates another one until copies all valid data. Then it returns spares in the original stripe group to the pool.

it triggers the redistribution of unused spare zones from NS2 to the active namespace, NS1, with a focus on boosting the performance of NS1 as depicted in Figure 4.23b. In the event that NS2 starts writing data, we ensure the minimum number of active zones and adequate bandwidth using essentials.

In the current design, we determine an *inactive* namespace as a namespace that has no allocation history in the last $N_{essential}$ physical zone allocations of the device, and lent spares are equally distributed across active namespaces. When an *inactive* namespace becomes active again, eZNS marks the leased spares as recall spares, and namespaces release them to the global pool as soon as they FINISH/RESET the stripe group in *v-zone*s. eZNS then returns them to the original namespace at the next *global overdrive* operation.

Reclaim: Zone/Namespace Compaction

Generally, an overdriven *v-zone* after entering the FINISH state will return spare zones. Therefore, spare zones circulate as long as namespaces continue to write to *v-zone*s. However, when a namespace overdrives *v-zone*s, which becomes *inactive* without releasing them, the arbiter has to use a *reclaim* operation to take back the spares to prevent resource leakage. To ensure no slowdown on the performance path, we employ an asynchronous window-based monitoring scheme, where the arbiter bookkeeps the status of each *inactive* namespace and continuously counts how long its status is in the read-only state. If a namespace presents no write I/Os for a certain amount of time, $T_{ReadOnly}$, the arbiter determines the namespace as *frozen* and triggers the reclaim procedure to

proactively collect the spare zones. The execution cost of *reclaim* depends on the configuration within the opened stripe group. If there are committed writes on the zone, *reclaim* will trigger a zone compaction and perform a sequence of I/O reads/writes, i.e., finishing existing zones, opening a new stripe group with shrunk width, and copying data to the new one. Once the migration is done, the spare zones can be returned to the global spare pool. Figure 4.24 presents an example of the zone reclaim process. When the arbiter identifies NS3 as *frozen*, it initiates the reclamation process (i.e., zone compaction) within the current stripe groups. These reclaimed spares are subsequently transferred to NS1.

The zone reclaiming indeed brings GC-like overheads back to the system. Thus, it is crucial that the system does not trigger the operation in normal conditions. In eZNS, zone reclaiming is only performed when namespaces have no write activity for two cycles of global overdrive. This is likely to happen infrequently, such as when an application undergoes a significant change in its running state. Moreover, reclaiming is triggered in a lazy fashion, executed in the background, and regulated by the scheduler to limit its performance impact. As a result, eZNS can avoid triggering zone reclaiming in normal conditions, maintaining high performance and efficiency.

4.3.5 Zone I/O Scheduler

eZNS mindfully orchestrates I/O reads/writes with the goal of providing equal read/write bandwidth shares among contending *v-zone*s, maximizing the overall device utilization, and mitigating superfluous head-of-line blocking when different types of requests interleave. Our zone I/O scheduler comprises two components: congestion-avoiding read scheduler and cache-aware write admission control.

Congestion-avoid Read Scheduler

Our design is based on the observations that (1) ZNS SSDs have no internal housekeeping operations; (2) write I/Os are sequential and synchronous. Hence, the read latency is stable and low until the die becomes congested, and it is thus possible to detect congestion directly via latency measurements.

eZNS introduces a hierarchical design that performs weighted round-robin scheduling firstly across active namespaces and then delay-based congestion control across each intra-namespace *v-zone*s. By conforming to the NVMe architecture, we create per-namespace NVMe queue pairs and offload the round-robin scheduling to the device. Then, we employ a Swift-like [67] congestion control

mechanism to decide the bandwidth allocation for each stripe group in the *v-zone*, where the delay is the device I/O command execution latency. As shown in Algorithm 5, during the congestion-free phase, upon a read I/O completion, we additively increase (AI) the congestion window until it approaches the maximum size (line 6). Since the congestion window (*cwnd*) is shared in the stripe group, when set to the stripe width, it indicates that there is one outstanding I/O per die in the sequential case. The SSD can max out its per-die bandwidth with a few outstanding I/Os. Thus, when the *cwnd* starts with the stripe width, it quickly ramps up to the device bandwidth capacity. Further, we limit the maximum congestion window (*cwnd*) to $4 \times strip\_width$ to minimize the software overheads when handling excess concurrent I/Os and avoid a meaningless rapid growth of *cwnd* that would imperil the efficiency of the MD phase. When congestion happens, we reduce the congestion window multiplicatively (line 4), whose ratio depends on the latency degradation degree. All the physical zones within a stripe group share the same congestion status. It is reasonable because sequential read bandwidth will be capped by the most congested physical zone. Random reads usually will not trigger frequent *cwnd* decrements because the minimum window size is large enough to absorb them. Our congestion control works cooperatively with the reduced stripe size of the overdrive and ensures a fair share of bandwidth regardless of the width of the stripe group.

Cache-aware Write Admission Control

Due to the non-linear write latency and the shared architecture, it is inappropriate to implement a local mechanism to mitigate the problem. Unlike the read congestion case, write congestion happens globally across all zones from all namespaces (§4.2.5). Therefore, eZNS monitors the global write latency and regulates writes using a token-based admission control scheme. We generate tokens periodically (ALG 5 lines 14–16) and admit write I/Os in a batch for each active *v-zone* to ensure overflow rarely happens. This requires a latency monitor to analyze the write cache eviction activity (ALG 5 lines 8–12). Here, we profile the block admission rate (defined as the minimum delay between two consecutive write blocks) and adjust the token generation rate based on its normalized average latency. This is based on an empirical observation that the latency of the write projects its capacity share in the write cache. Hence, we equalize the latency for all write zones and calculate available tokens using the average value. Additionally, we update the available tokens based on the elapsed time from the last token refill upon a write submission. By doing so, we expect that writes are

Algorithm 5 Zone I/O Scheduler

procedure READ SUBMISSION()
  $read\_io \leftarrow head(pending\_queue)$
  if $cwnd \geq io\_count + size(read\_io)$ **then**
    submit($read\_io$)
procedure READ COMPLETION()
  $lat\_thresh \leftarrow 500us$
  if $io\_lat > lat\_thresh$ **then**
    $cwnd = \max(1, cwnd \times \frac{lat\_threash}{2 \times io\_lat})$
  else ▷ α = additive factor
    $cwnd = \min(stripe\_width \times 4, cwnd + \alpha \times \frac{io\_count}{cwnd})$
procedure WRITE SUBMISSION()
  $write\_io \leftarrow head(pending\_queue)$
  if $tokens \geq batch\_size(write\_io)$ **then**
    submit($write\_io$)
procedure WRITE COMPLETION()
  $per\_block\_lat = per\_block\_lat + \frac{io\_lat}{num\_blocks}$
  num_ios += 1
procedure WRITE LATENCY MONITOR()
  On t every 10ms
  $total\_lat = \sum_{active\_zone} per\_block\_lat$
  $total\_ios = \sum_{active\_zone} num\_ios$
  $avg\_lat(t) = \frac{total\_lat}{total\_ios}$
  $block\_admission\_rate = \frac{avg\_lat(t-1)+avg\_lat(t)}{2}$
procedure WRITE TOKEN GENERATOR()
  On every 1ms
  for pending write zones **do**
    $token$ += $\frac{now-last\_refill}{block\_admission\_rate} \times stripe\_width$

self-clocked in the congestion-less condition.

Note that (1) when read and write I/Os mix on a physical die, the total aggregate bandwidth will drop due to the NAND interference effect. However, our read scheduler and write admission control require little coordination because both modules only use the latency (gradient) as a signal to infer the current bandwidth capacity; (2) we coalesce stripes for the same physical zone within a user I/O and submit one write I/O to the device in a batch, and thus, a small stripe size does not degrade the write bandwidth.

4.4 Evaluation

We add a thin layer in the SPDK framework [124] to implement eZNS and realize the *v-zone* concept. The primary reason for choosing the SPDK approach was its ease of implementation and

integration into the software stack of a storage server accessible by remote clients. Moreover, the SPDK-based design can also be used in a local system to serve virtual machines through the SPDK vhost extension. This approach allows the storage server to provide efficient and high-performance I/O operations, while remaining compatible with existing software stacks. We use the same test environment as in §4.2.1. Non-SPDK applications require a standard ZNS block device exposed via the kernel NVMe driver; thus, we set up eZNS as a disaggregated storage device over RDMA (NVMe-over-RDMA) and connect to it using the kernel NVMe driver.

Micro-benchmarks: We use FIO [37] to generate synthetic workloads and allocate a separate thread for each worker when the workload writes to multiple namespaces or zones. For read workloads, we first precondition the namespace by performing sequential writes for the entire range of read I/O. Additionally, we perform a pre-calibration step to determine the die allocations in case the evaluation requires a die-level collision.

Ported Applications: We use RocksDB as a real-world ZNS application, to evaluate the performance of eZNS We run RocksDB over ZenFS [15] to enable the ZNS support. As eZNS complies with the standard NVMe ZNS specification, no modification is required for the application and ZenFS. We initialize the DB instance with 500M entities of 20-byte keys and 1,000-byte values.

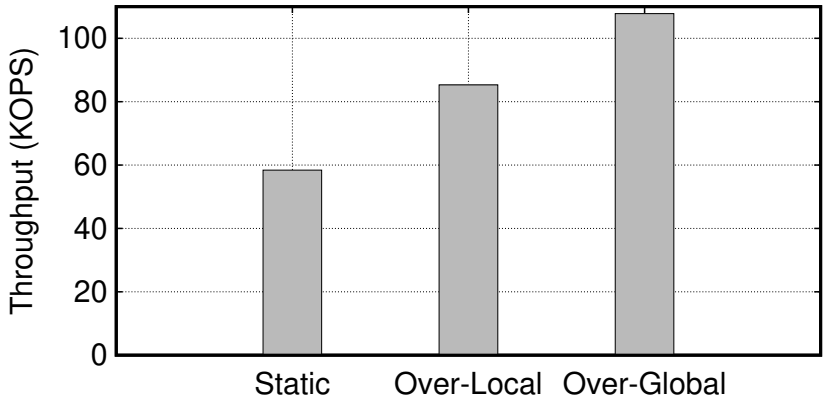


Figure 4.25: Throughput of *overwrite* workload on a single namespace without other tenants. (Over: Overdrive)

Default *v-zone* Configuration: By default, eZNS creates four namespaces (NS1–4), each of which is allocated 32 essential and 32 spare resources. Since each namespace provides a maximum

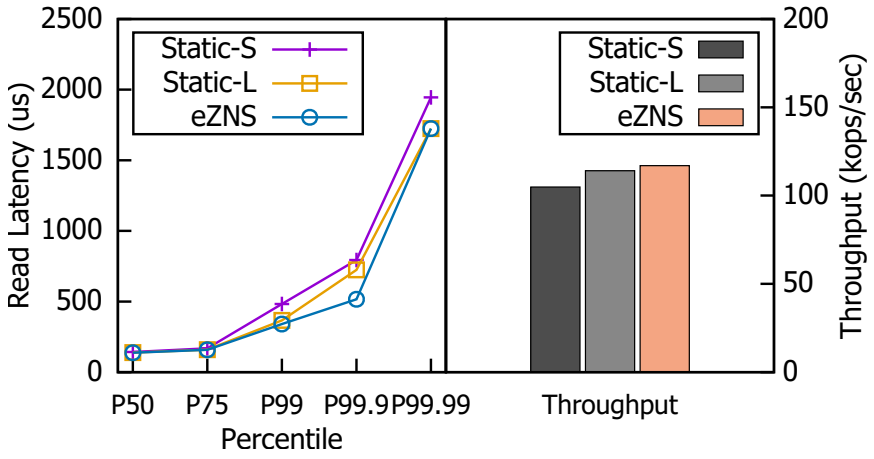


Figure 4.26: ***readwhilewriting*** **workload on a single tenant configurations. Static has stripe width of 16. (S: 4KB stripe, L: 16KB stripe)**

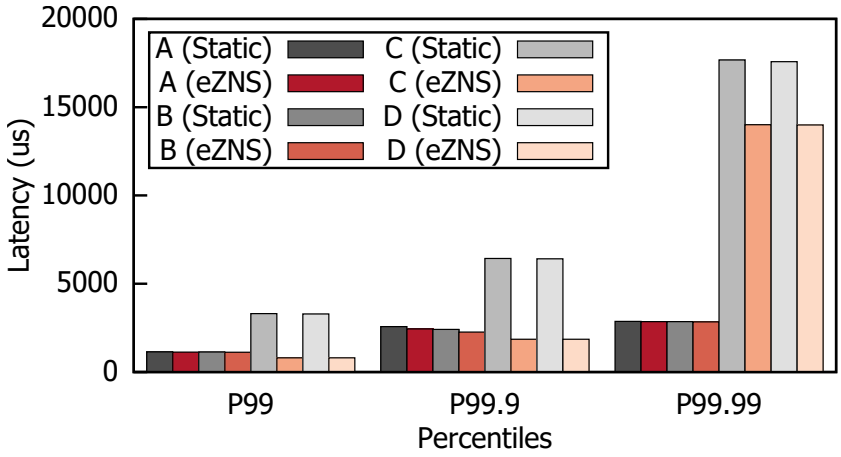


Figure 4.27: Latency of db_bench workloads (2 overwrite, 2 randomread) on different namespaces over eZNS and static zone.

of 16 active zones, the minimum stripe width for *v-zone* is 2 with a stripe size of 32KB. However, eZNS can overdrive the width up to 16 with a stripe size of 4KB. For a fair comparison, we prepare a static logical zone configured with stripe width and size of 4 and 16KB, respectively; hence, it also accesses full device capability when the application populates enough active logical zones. Both a *v-zone* and a static logical zone comprise 16 physical zones. Different configurations are used for single-tenant evaluation (single namespace) and YCSB benchmark (six namespaces), as specified in Section 4.4.3.

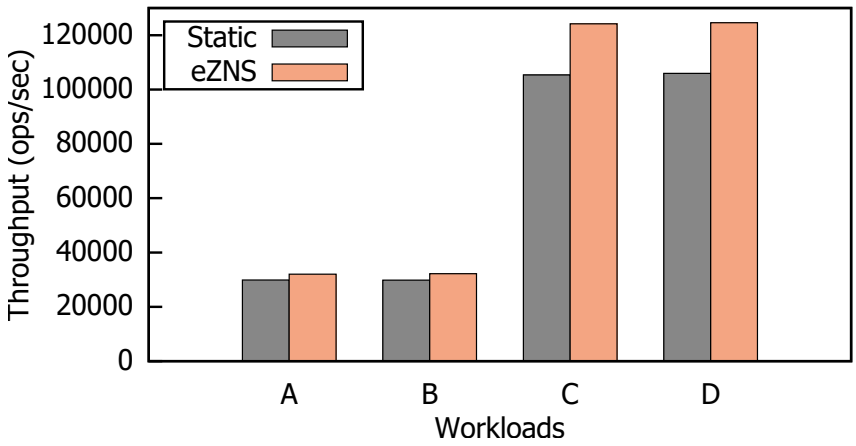


Figure 4.28: Throughput of db_bench workloads (2 overwrite, 2 randomread) on different namespaces over eZNS and static zone.

4.4.1 Zone Ballooning

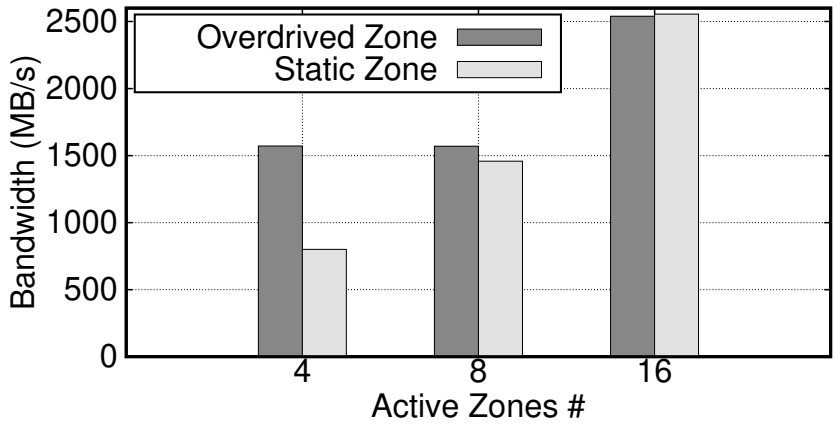


Figure 4.29: B/W comparison between an overdrived and three statically configured zones.

We demonstrate the efficiency of zone ballooning when handling large writes (i.e., 512KB I/O with a queue depth of one). First, within a namespace, we compare the performance between a *v-zone* and a static logical zone, where the number of writers is configured to 4, 8, and 16, respectively. Each writer submits a write I/O to different zones. Our *local overdrive* operation can reap more spare zones and lead to better throughput. As shown in Figure 4.29, the *v-zone* outperforms the static one by 2.0× under the 4-writer case as 4 static logical zones enable only 16 physical zones while 4 *v-zone* overdrive the width to 8 and expand to 32 physical zones. In the 8-writer and 16-writer cases,

v-zone reduces the overdrive width accordingly and utilizes the same number of physical zones (32 and 64, respectively) with the static logical zone.

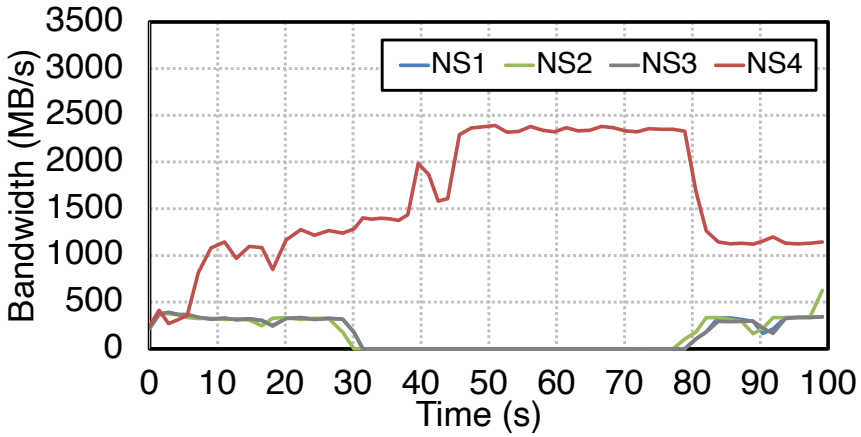


Figure 4.30: Performance variation of four namespaces with global overdrive under 100s.

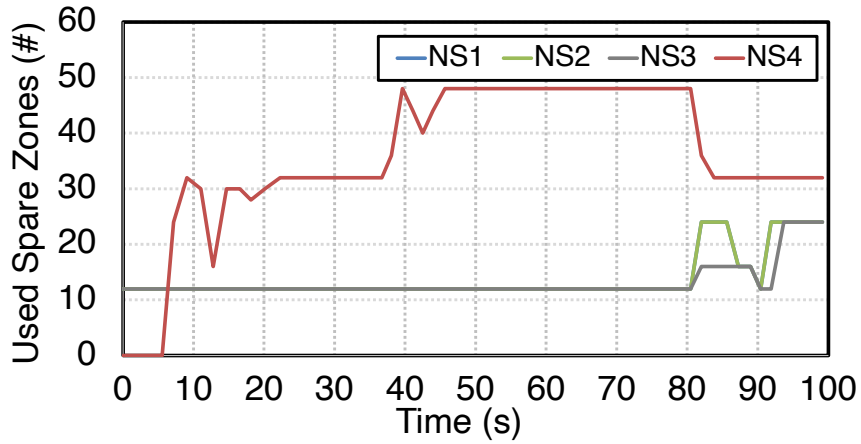


Figure 4.31: The number of used spare zones of four namespaces under 100s.

To evaluate eZNS's adaptiveness under dynamic workloads, we set up overdriven zones from different namespaces. The first three namespaces (NS1, NS2, and NS3) run two writers, while the fourth namespace (NS4) runs eight. NS1, NS2, and NS3 stop issuing writes at t=30s and resume the writing activity at t=80s. We measure the throughput and spare zone usage of four zones for a 100s profiling window (Figures 4.30 and 4.31). When the other three zones become idle, the *v-zone* from

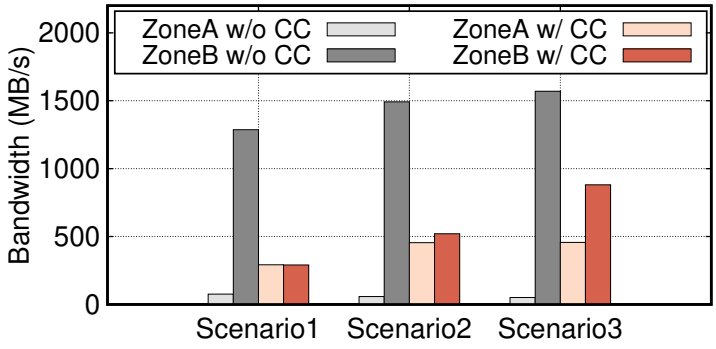
NS4 takes up to 3× more spare zones from other namespaces using the *global overdrive* primitive and maxes out its write bandwidth (∼2.3GB/s). It can then quickly release the harvest zones when other zones start issuing writes again.

4.4.2 Zone I/O Fairness

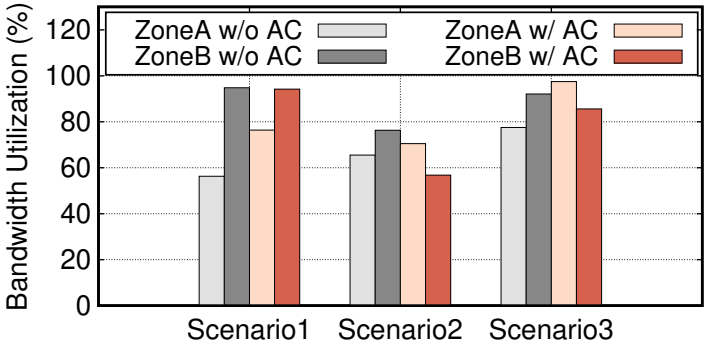
We evaluate our I/O scheduler in various synthetic congestion scenarios by placing competing zones in the same physical die group. We compare the performance of all co-located zones when enabling and disabling our mechanism. The zone ballooning mechanism is turned off for all cases. We report per-thread bandwidth in Figure 4.32.

Read-Read Fairness. We run a sequential read of 128KB I/O size at two types of zones on co-located dies. To equally load the physical dies, we populate more threads for lower-width zones. For example, a zone with a width of 2 runs four threads on each stripe group, while a zone with a width 8 has only one thread. As shown in Figure 4.32-a, in scenario 1, when disabling our congestion control mechanism, Zone A (configured with stripe width 2 and stripe size 32KB, QD-1) and Zone B (configured with stripe width 8 and stripe size 8KB, QD-32), even holding the same sized full stripe, achieve 76MB/s and 1287MB/s, respectively. This is because the zone with the higher QD dominates on the competing die. Our scheme effectively controls the per-zone window size and ensures that each zone submits the same amount of outstanding bytes. Hence, both Zone A and Zone B sustain 290MB/s. In scenarios 2 and 3, we change the Zone A stripe configuration to <stripe width 4, stripe size 16KB, QD-1> and <stripe width 8, stripe size 8KB, QD-1), and observe similar behavior when turning off the read congestion logic. In scenario 3, the congestion level on the die gets lowered as Zone A only submits one 128KB I/O (which was 4 and 2 in scenarios 1 and 2, respectively). Hence, the read latency also becomes below the threshold, and the I/O scheduler chooses to max out the bandwidth.

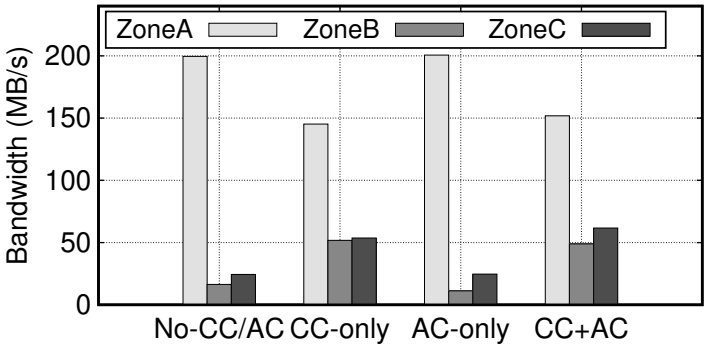
Write-Write Fairness. We carefully create different write congestion scenarios and see how our admission control operates. The workload used is a sequential write of 512KB size. In the first scenario, we co-locate 16 regular write zones (Zone A, where each has a striping width of 8 with 8KB stripe size and submits write I/Os at 5ms intervals, sustaining 95MB/s maximum throughput) with a busy writer (Zone B, that has width 2 and 32KB stripe size, submits I/O without interval delays, achieving 85MB/s at most). Figure 4.32-b reports the bandwidth utilization of one regular



(a) Read-Read Fairness. (128KB Read. Zone A with QD-1, and Zone B with QD-32)



(b) Write-Write Fairness. (Zone A for regular writers, and Zone B for the busy writer)



(c) Read-Write Fairness. (Zone A for readers, Zone B for the busy writer, and Zone C for regular writers)

Figure 4.32: Efficiency of eZNS on handling read-read, write-write, and read-write congestion. (CC=Congestion Control, AC=Admission Control)

zone (Zone A) and the busy writer (Zone B). Our admission control mechanism limits the write issuing rate of Zone B and gives more room at the write cache to the regular zone (Zone A), leading to 35.7% bandwidth improvement per thread. Next, we set up a highly-congested case by changing

16 regular zones to busy writers (scenario 2). Without the admission control, Zone B runs at 64.9 MB/s, which is 32.5 MB/s per physical zone or 76.3% of the physical zone bandwidth, while Zone A receives only 16.4 MB/s per physical zone or 38.4% of the physical zone bandwidth. As described in §4.3.5, our scheme equally distributes the write bandwidth share across competing zones, and Zone B receives 56.8% of the total bandwidth of 2 physical zones, increasing the bandwidth of Zone A by 7.6%. As a result, it improves the overall bandwidth of the device from 2160.9 MB/s to 2304.3 MB/s, or by 6.6%. The last scenario is a collision-less one at the die level where we eliminate the overlapping region among all the write zones by populating active physical zones lesser than the number of dies. (i.e., reducing the number of regular zones to 15.) Similarly, when enabling the admission control, the bandwidth allocated for Zone B slightly decreases (∼7.2%) to avoid cache congestion, and the overall device bandwidth is increased by 24.7% (from 2403.3 MB/s to 2997.7 MB/s).

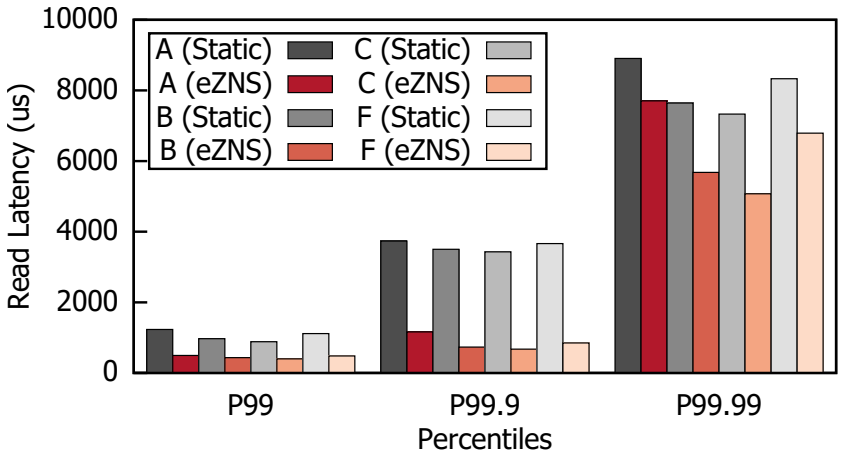


Figure 4.33: Read latency of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone.

Read-Write Fairness. We examine how our congestion control mechanism coordinates with the admission control when handling read/write mixed workloads. In this experiment, we set up three types of zones: (1) ×16 regular readers (Zone A), where each has a striping width of 2 and 32KB stripe size, performing 128KB random read at queue depth 32, across all physical dies; (2) 1 busy writer (Zone B), whose striping width is 2 with 32KB stripe size; (3) ×16 regular writers (Zone

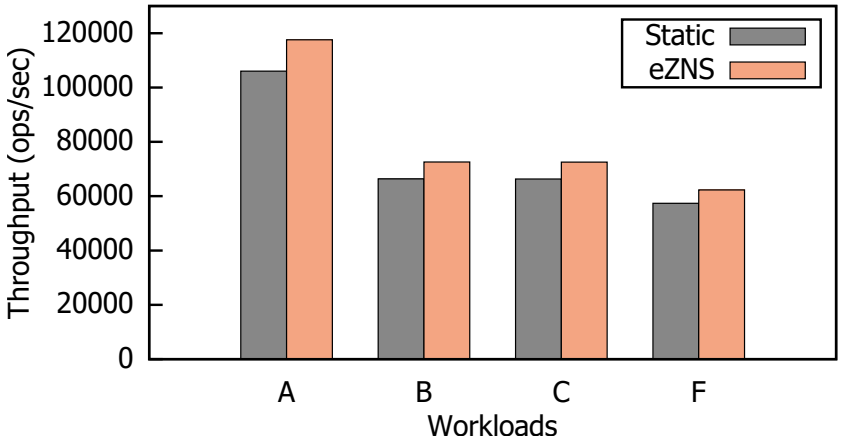


Figure 4.34: Throughput of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone.

C), which has a striping width of 8 and 32KB stripe size each, submitting I/Os under 5ms interval. Both B and C issue 512KB large writes. Figure 4.32-c reports their per-thread bandwidth. When disabling our scheduler, each reader achieves 199.6MB/s but writes are jeopardized significantly, as Zone B and Zone C can only achieve 19.3% and 27.3% of their maximum bandwidth. As we gradually turn on our mechanisms, the congestion control shrinks the window size such that more bandwidth is allocated to the writes. Further, the admission control then equally partitions bandwidth among competing writing zones. As shown in the CC+AC case, zones A, B, and C can sustain 71.6%, 57.5%, and 70.1% of their maximum bandwidth capacity, respectively.

4.4.3 Application: RocksDB

To evaluate eZNS in a real-world scenario, we use RocksDB [103] over the ZenFS storage backend. In addition to the built-in utility in the RocksDB *db_bench* tool, we port YCSB workload generators [12] for the mixed workload evaluation.

Single-tenant performance. First, we run the *overwrite* profile of the *db_bench* to evaluate the write performance of eZNS. Figure 4.25 demonstrates that eZNS improves the throughput by 46.1% and 84.5% with *local* and *global* overdrive, respectively. The ZenFS opens all available zones regardless of actual usage; hence, our *local overdrive* has minimal impact, and the stripe width, 4, becomes the same as with static zones. However, our I/O scheduler mitigates intra-namespace interference, and each zone receives a fair share of the bandwidth, eliminating unnecessary application

delays due to zone interference. When *global overdrive* comes in, zones further harness more active resources and attain higher bandwidth.

Next, we evaluate the performance of a single tenant using the *readwhilewriting* profile of the *db_ bench*, which runs one writer and multiple readers. This workload profile demonstrates a read/write mixed scenario. In the case of a single-tenant configuration, eZNS creates a single namespace on the device and allocates 128 essential and 128 spare resources to it. Since only two stripe widths, 8 and 16, are possible in this configuration, eZNS sets the stripe size to 16KB for the width of 8 to avoid the namespace running only on large stripe sizes. We compare the performance of eZNS over two static configurations, both with a stripe width of 16 but with different stripe sizes of 4KB and 16KB. Since there is only one namespace on the device, eZNS always overdrives *v-zone*s to the width of 16, which is identical to the static configurations. Therefore, both the static namespace and eZNS can exploit all available bandwidth on the device. However, the I/O scheduler of eZNS helps mitigate interferences between zones and improves overall application performance. Figure 4.26 shows that eZNS improves the P99.9 and P99.99 read latency by 28.7% and 11.3% over the static configurations with a stripe size of 16KB and 4KB, respectively. Additionally, eZNS also improves the throughput by 11.5% and 2.5% with a stripe size of 4KB and 16KB.

Multi-tenant Performance. Next, we set up instances of *db_ bench* on four namespaces (A, B, C, and D), each with a different workload profile. A and B perform the *overwrite* profile, while C and D execute *randomread* concurrently. We run the benchmark for 1,800sec and report the latency and the throughput. Figure 4.27 shows that our I/O scheduler significantly reduces P99.9 and P99.99 read (C/D) latency by 71.1% and 20.5%, respectively. In terms of throughput, eZNS improves write (A/B) and read (C/D) throughput by 7.5% and 17.7%, respectively. Furthermore, while the read latency and throughput are improved, the write latency is either maintained at the same level or decreased compared to the static configuration because eZNS moves the spare bandwidth from read-only namespaces (C/D) to write-heavy ones (A/B).

Mixed YCSB Workloads with four namespaces. YCSB [32] is widely used to benchmark realistic workloads. In our experiments, we run YCSB workload profiles A, B, C, and F on each of the six namespaces. We exclude YCSB workload profiles D and E because they increase the number of entities in the DB instance during the benchmark. As YCSB-C (read-only) does not submit any write I/Os during the benchmark, eZNS triggers *global overdrive* and rebalances the bandwidth

to the write-most namespaces (A and F). Figure 4.33 shows that the I/O scheduler improves the P99.9 read latency of read-intensive workloads (YCSB B and C) and also the read-modify-write one (YCSB F) by 79.1%, 80.3%, and 76.8%, respectively. The throughput improvement from global overdrive is up to 10.9% for the write-most workload A in Figure 4.34.

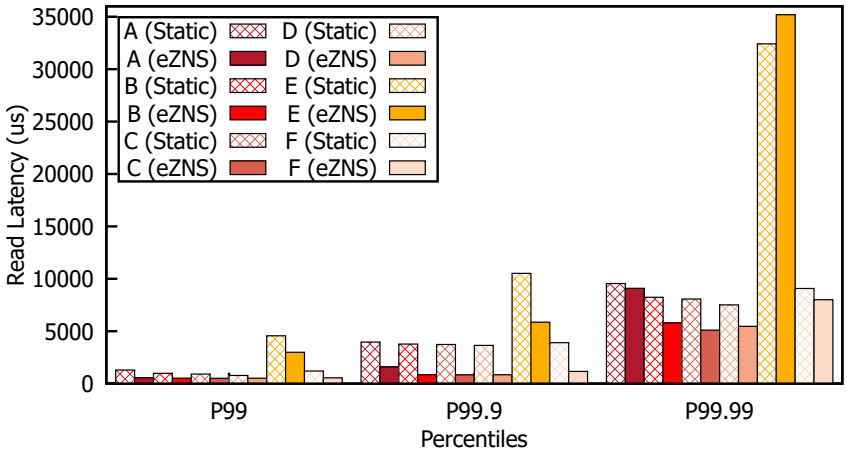


Figure 4.35: Read latency of YCSB workloads (A/B/C/D/E/F) on different namespaces over eZNS and static zone.

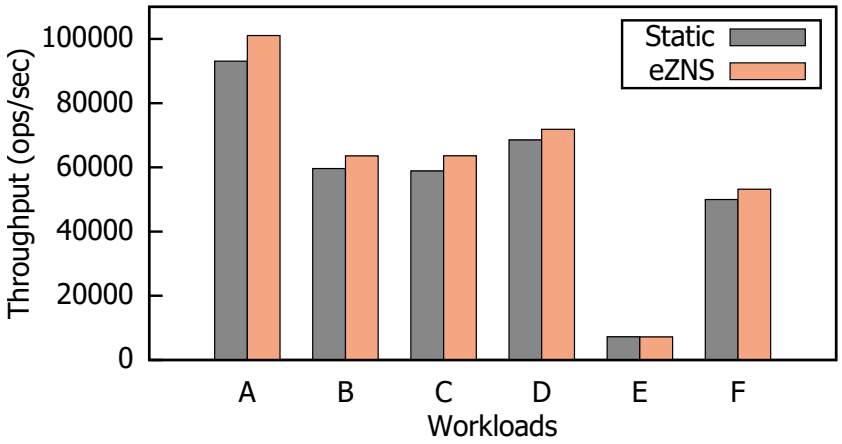


Figure 4.36: Throughput of YCSB workloads (A/B/C/D/E/F) on different namespaces over eZNS and static zone.

Mixed YCSB Workloads with six namespaces. We also conducted evaluations using all

six workload profiles of YCSB (A-F) with a configuration involving six namespaces. To support six namespaces, we reduced the maximum active zones to 11, allocating 22 essentials and 20 spares for each namespace during the initialization process. The remaining 4 physical zones are designated as part of the global spare pool.

Figures 4.35 and 4.36 present the tail latencies and throughput comparisons between eZNS and the static zone configuration. For YCSB A-C and F, our results closely resemble those observed in the four-namespace scenario. Notably, YCSB A demonstrates the most significant improvement in throughput, with an increase of 8.6%, while the read-heavy workloads (YCSB B, C, and F) exhibit remarkable reductions in P99.9 read latency, with improvements of up to 77.6%. YCSB D, a read-intensive profile focusing on the latest data, also showcases notable improvements, with P99.9 latency reduced by 76.9% and a throughput increase of 4.8%. In contrast, YCSB E, which represents a range-scan workload, demonstrates the least improvement among the six workload profiles. Although the P99.9 read latency for YCSB E is reduced by 44.3%, its throughput remains slightly below that of the static zone at 99.0%. In addition, the P99.99 read latency is worse than the static zone configuration. This is primarily due to YCSB E's higher per-operation cost and increased bandwidth of other tenants. The scan operation of YCSB E generates a large number of read I/Os per operation, having more congested I/Os per operation than other workload profiles. At the same time, the increased device bandwidth further raises the chance of congestion on accessing dies. As a result, the worst case latency could be higher than the static configuration. If we keep the throughput of tenants same as the static configuration, P99.99 will be dramatically decreased as well.

eZNS on File System (F2FS) To evaluate the performance of eZNS on a general file system, we replicated the scenario with four namespaces using RocksDB over F2FS [72] instead of ZenFS, while maintaining an identical zone configuration to that of ZenFS. The read-intensive workloads (YCSB B, C, and F) demonstrate improvements in both P99.9 read latencies and throughput, as illustrated in Figures 4.37 and 4.38.

However, YCSB A does not benefit from eZNS and even performs worse than the static zone configuration, achieving a throughput of only 95.3%. This can be attributed to the lower zone utilization of F2FS. We observed that F2FS opens up to three zones but allocates only one zone for writing user data, resulting in the lower write bandwidth for both eZNS and the static zone.

Additionally, since we have tuned the maximum active zones to 16 and the striping size cannot be sized smaller than 4KB, eZNS cannot increase the striping width beyond 8. Consequently, the global overdrive mechanism does not operate effectively in this scenario, forcing eZNS to make a trade-off, sacrificing a small amount of throughput from YCSB A to ensure a fair distribution of read bandwidth across all namespaces.

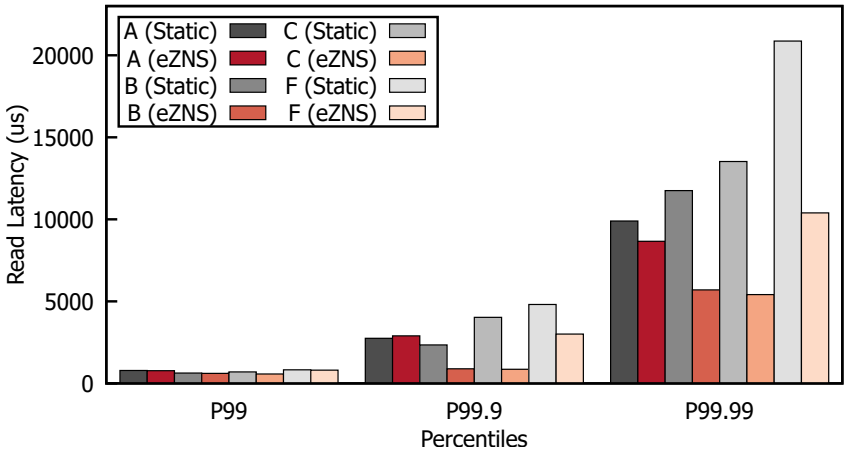


Figure 4.37: Read latency of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone running on F2FS.

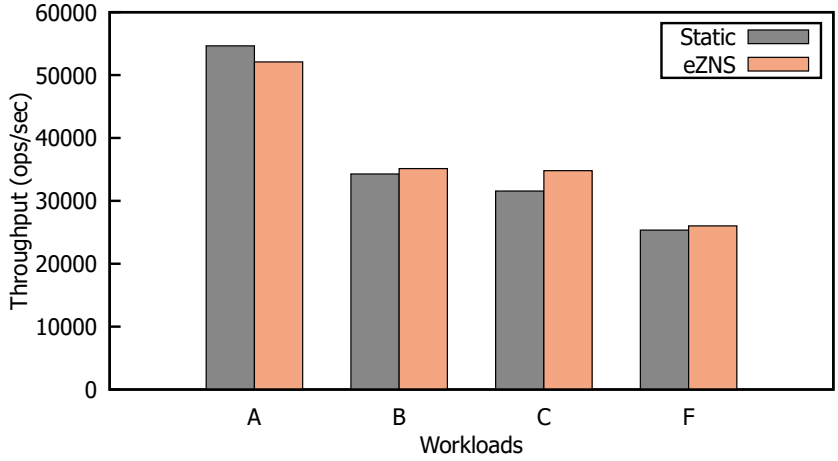


Figure 4.38: Throughput of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone running on F2FS.

4.4.4 Overhead analysis

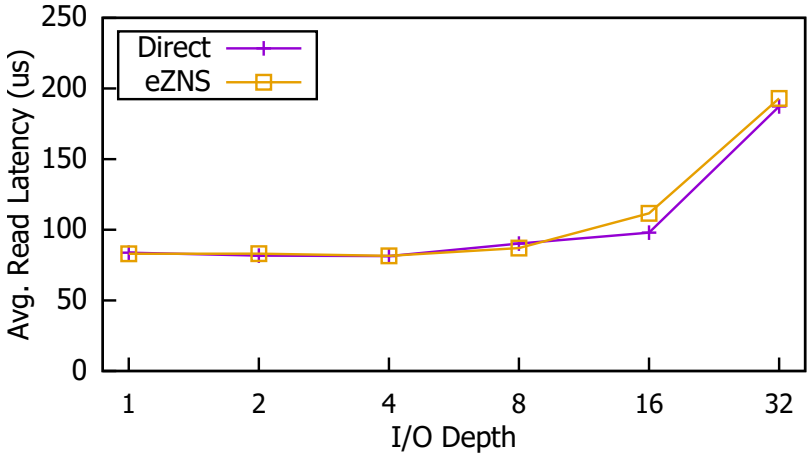


Figure 4.39: Comparison of Avg. Read Latency for 4KB I/Os at various depths between the host-managed zone access and eZNS.

End-to-end read latency overhead. Since eZNS serves as an orchestration layer between the physical ZNS device and the NVMe-over-Fabrics target, there may be some overhead when the I/O load is very low. To measure this overhead, we conducted a quantitative analysis using 4KB random read I/Os and compared it with host-managed zone access, where the host directly accesses the physical device without eZNS. Figure 4.39 demonstrates that eZNS does not add a noticeable latency overhead for I/O depths up to 8. As the I/O depth goes over 16, up to 14.0% overhead is observed due to the I/O scheduler delaying the I/O submission. However, the scheduler provides significant advantages in real-world scenarios, as shown in previous experiments.

Memory footprint. eZNS relies on in-memory data structures for managing *v-zone* metadata, including the logical-to-physical mapping and scheduling statistics. Additionally, it maintains a copy of the physical zone information to reduce unnecessary queries to the device, enabling faster zone allocation and deallocation. In our current implementation, the size of *v-zone* metadata is less than 1KB, and the size of physical zone information is smaller than 64 bytes. For our testbed SSD with four namespaces, each with 1TB of capacity, *v-zone* metadata and physical zone information require 2MB and 2.5MB of memory, respectively. Compared to the memory requirements of the page-mapping in conventional SSDs, the memory usage of eZNS is negligible.

4.5 Related Work

Early ZNS Exploration. Researchers have made initial efforts to understand the ZNS interface and integrate it into the host storage stack. Theano Stavrinos *et al.* [125] argue for a shift in research to the zone interface and discuss future directions (e.g., applying application-level information for zone management and I/O scheduling). Hojin Shin *et al.* [118] develop a performance analysis tool for a ZNS SSD and profile its parallelism, isolation, and predictability properties. Compared with our study, they didn't investigate the underlying device's internal mechanisms when realizing the zoned namespace interface and, thereby, are unable to correlate the observed performance with the ZNS SSD characteristics. ZNS+ [45] enhances the existing interface with two new architectural primitives to optimize LFS file systems. With such support, the authors then propose copy-back-aware block allocation and hybrid segment recycling techniques. Hanyeoreum Bae *et al.* [11] prioritize I/O requests for less congested zones using an interference map, whilst updates incur significant overheads. Although revising the ZNS interface and exposing the physical allocation of zones could potentially eliminate this overhead, it may not be feasible for existing devices due to vendors' resistance to disclosing internal architecture and policies. eZNS uses a delay to determine congestion and doesn't require an allocation map. Furthermore, eZNS addresses such as read and write differences, zone striping, and bandwidth provisioning issues that were not discussed in their work. Minwoo Im *et al.* [50] improved ZenFS on small-zone SSDs by introducing read/write parallelism with a multi-threaded I/O engine and lifetime-based zone management at the application level. However, it requires adjusting the RocksDB parameters to match the device capability instead of the workload-optimized parameters. This can increase the complexity of parameter configuration, resulting in sub-optimal settings for the workload. eZNS maximizes parallelism within the thin layer, regardless of the underlying device and the application profile. It exploits the device's parallel I/O processing capability that can be executed on a single thread.

Addressing Inefficiencies of Conventional SSDs. Early SSD researches [3, 48, 25, 96] focused on internal parallelism and tradeoffs between concurrency, locality, bandwidth, capacity, performance, and lifetime. Modern SSDs handle random write patterns with page mapping FTL, write-cache, and superblock concepts [134] that group blocks together. It benefits from high parallelism that transforms writes into sequential NAND programming. However, multi-tenancy workloads cause

interference and high write amplification factor (WAF). ZNS SSDs eliminate garbage collection and fix WAF to one but require careful parallelism management across zones to avoid degraded device utilization. In addition, future QLC-based ZNS SSDs may have fewer active zones due to a multi-pass programming algorithm [60]. eZNS addresses these challenges by adjusting the parallelism of each logical zone based on the number of namespace flows, providing fully dynamic parallelism and maximizing device capability while presenting an identical logical view to applications.

IODA [76] is an I/O deterministic flash array that uses the I/O determinism feature and exploits data redundancy for a strong latency predictability contract. SSDs can fail an I/O to allow predictable I/Os through proactive data reconstruction. We target the ZNS SSD, where there are no random I/Os, and GCs are user-controlled. This opens up a different design space. Although techniques addressing GC-related interference are not beneficial to GC-free ZNS SSDs, others such as Engurance Group(EG) and NVM Set can be useful to ensure physically isolated zone allocation. eZNS can take advantage of the geometry hints via EG (or even finer-grained NVM Sets). Unfortunately, there is no currently available SSD that supports both ZNS and EG, but it will be an interesting direction for future work.

Open-Channel SSDs. These drives have no mapping layer in the controller and directly expose a set of physically contiguous blocks to applications, and leave the data placement/wear-leveling responsibilities to the host. Researchers have built several domain-specific solutions using them. For example, SDF [95] employs a hardware-software co-designed approach that exposes flash channel details and delegates I/O control-plane and data-plane tasks to host applications. LOCS [133] further improves the throughput of an LSM-tree-based KV store by optimizing the scheduling and dispatching policies, considering the characteristics of access patterns of the LevelDB. RAIL [80] designs a horizontal hot-cold separation mechanism and divides dies into two groups, where user and GC writes are scheduled to different dies, and the hot/cold ratio is dynamically adjusted based on runtime monitoring. By having full control over the device, one can implement a deterministic *v-zone* using eZNS. Despite the potential architecture, it imposes too many responsibilities on the software handling tasks that are offloadable to the device with no cost, for example, wear-leveling, physical zone-to-die mapping, etc. Another challenge arises when the system consists of heterogeneous devices resulting in the overhead of managing different H/W architectures (NAND chip capacity, channel/die configuration, etc.).

eZNS as a firmware. One may implement eZNS solely in the SSD using the controller and firmware. This approach can exploit internal knowledge such as NAND specification, Channel/Die structure, queue length on a die, etc. Thus, it may control the interference better and outperform the software-based implementation. However, completing eZNS in one device is not future-proof, given the disaggregated systems architecture in data centers. The software-based solution can build an eZNS-based system spanning multiple devices enabling elastic capacity scaling, load-aware allocation, high availability, and more. At the opposite end is the Open-Channel SSD. By having full control over the device, one can implement a deterministic *v-zone* using eZNS. Despite the potential architecture, it imposes too many responsibilities on the software handling tasks that are offloadable to the device with no cost, for example, wear-leveling, physical zone-to-die mapping, etc. Another challenge arises when the system consists of heterogeneous devices resulting in the overhead of managing different H/W architectures (NAND chip capacity, channel/die configuration, etc.).

ZNS SSD architecture. We believe the ZNS interface is an appropriate compromise between H/W and S/W ends. However, as one might expect, there certainly are H/W details and architectural assurances that improve system performance and predictability without undermining ZNS's promising abstractions. For example, exposing channel/die structure as the number of parallel units (PU) further optimizes the active zone management, isolating per-die write cache eliminates inter-zone write interference, and so on. These are the issues to be considered as we evolve the ZNS interface.

Filesystems: BtrFS, F2FS. BtrFS [104] and F2FS [72] are the filesystems that currently support the ZNS SSD. However, they are still in a preliminary stage and lack the features that are needed to max out the ZNS SSD capability. For example, they write user data to only one zone at a time, limiting the bandwidth to a single zone, leading to sub-optimal performance.

4.6 Conclusion

This paper presents an in-depth study on understanding the characteristics of a commodity ZNS SSD. Then, we propose eZNS, realizing an elastic zoned view via *v-zone*, providing a flexible zone scaling interface transparent to the application that maxes out the device capability, and ensuring a fair bandwidth share between zones. We demonstrate significant performance and fairness improvements using eZNS over various scenarios.

Chapter 5

NVMf Router: Interposable Transport Protocol for Enhancing End-to-End Performance in Storage Disaggregation

5.1 Motivation

Storage disaggregation has become a foundational architecture for modern data centers, offering independent scaling of compute and storage resources, improved infrastructure sharing, and overall resource efficiency. However, this shift places unprecedented demands on the network: remote storage must now be accessed with performance comparable to locally attached devices, requiring ultra-low latency, high throughput, and predictable Quality of Service (QoS).

Existing storage transport protocols—including NVMe-over-Fabrics (NVMe-oF), key-value stores, and distributed object stores—have adapted to operate over high-speed network fabrics. Despite their logical differences, these protocols face common operational challenges: dynamic request routing, replication, load balancing, and failure handling. Today, these critical functions are almost universally handled at the host level, resulting in substantial CPU overhead, additional latency, limited scalability under load, and QoS unpredictability.

Meanwhile, data center networks have evolved dramatically. Networks now routinely offer microsecond-level latencies and near-lossless delivery, shifting the performance bottleneck increasingly toward host processing rather than the network itself. New transport protocols, such as RDMA and Homa [90], have emerged, specifically designed for these high-performance environments, illustrating that traditional TCP/UDP models are no longer sufficient.

Recognizing this shift, there have been efforts to offload storage-specific network functions to programmable hardware such as SmartNICs and DPUs. Systems like Xenic [113], Solar [86], and LineFS [62] demonstrate the performance benefits of such offloading. However, existing approaches are narrowly specialized and vertically integrated, tightly coupling hardware platforms, transport protocols, and application-specific optimizations. As a result, they lack portability and generality, making it difficult to extend these designs to other storage stacks or heterogeneous hardware environments.

While systems like IODA [76] have demonstrated the use of fast-failing mechanisms for handling performance degradation in RAID arrays, such techniques are typically effective only over longer timescales (seconds or more) and assume tightly coupled, centrally orchestrated environments. In a disaggregated storage system, where multiple independent hosts and applications concurrently access shared storage devices without centralized control, such orchestration becomes impractical. Moreover, storage congestion events in disaggregated architectures may last only tens of milliseconds but still cause significant and sustained performance degradation if not addressed promptly, as demonstrated by Gimbal [87]. Existing fast-failing or request re-issuance techniques cannot react quickly enough to transient congestion, nor can they operate efficiently across independently managed hosts.

These limitations highlight the urgent need for a fast-reacting, host-independent transport-layer solution that can dynamically redirect storage requests at fine timescales without relying on centralized orchestration or re-issuing requests from the host. Storage systems should not have to choose between flexibility and performance. Instead, transport intelligence must move into the network layer itself.

Motivated by these challenges, we introduce the Interposable Transport Protocol (ITP). ITP enables transparent, inline interposition at the packet level, allowing programmable SmartNICs to dynamically adapt storage request flows based on real-time network and storage conditions. By decoupling storage control logic from the host and eliminating rigid, vertically integrated designs, ITP reduces host CPU load, lowers access latency, improves system scalability, and brings truly general-purpose, network-layer intelligence to disaggregated storage architectures.

5.2 Requirements for Network Protocols in Storage Disaggregation

By analyzing the characteristics of storage networks and reviewing existing solutions, we identify essential requirements that a new, storage-oriented network protocol must meet to effectively address the needs of modern disaggregated environments:

- **DMA Offloading Capability:** Recent SmartNICs are equipped with DMA engines programmable through user-defined logic. The new protocol must support DMA offloading similar to RDMA, leveraging these DMA engines to efficiently transfer data directly between network interfaces and host memory. This capability is particularly critical for storage applications, where the typical data transfer size matches the network's maximum transfer unit (MTU), often 4KB. Given the need for high throughput and maximizing IOPS, eliminating memory copies during data transfer becomes a critical optimization for achieving low latency and high efficiency. Previous offloading systems have successfully demonstrated these capabilities using FPGA- or NIC-based DMA engines.

- **Connection-less and Stateless Packet Design:** The protocol must avoid embedding stateful information within packets. In other words, it should adopt a connection-less, message-based model similar to UDP. Connection-oriented protocols like TCP require significant state management, which imposes substantial overhead on SmartNICs and complicates state synchronization with the host. Recent research [4] demonstrated that the connection-less design can handle network communications, even remote memory access, efficiently. A stateless design allows SmartNICs to efficiently handle large numbers of concurrent requests, enabling scalable, low-latency processing without constant host coordination.

- **Transparent Packet Interposition and Routing:** Each packet must be capable of being transparently forwarded, redirected, split, or replicated at the network layer without host involvement. This capability is crucial for storage applications, which often rely on replication strategies to improve reliability and performance. For example, when a node experiences congestion or a device failure, the system must seamlessly redirect requests to an alternate replica containing the needed data. Offloading such operations is key to reducing latency and

improving scalability. To achieve this, the protocol must inherently verify and interpret incoming packets, enabling SmartNICs to make real-time routing decisions and manipulate packets without relying on host-side intervention.

These requirements collectively establish a foundation for a high-performance, scalable, and flexible network protocol tailored specifically for storage disaggregation scenarios. In the following section, we describe our proposed **Interposable Transport Protocol (ITP)**, explicitly designed to fulfill these essential characteristics.

5.3 Design: Interposable Transport Protocol

We propose a new Interposable Transport Protocol (ITP) Layer that meets the above requirements. ITP is designed to support general-purpose storage applications, with a particular focus on enabling SmartNICs attached to initiator and target hosts to interpose and flexibly manage storage traffic for acceleration and offloading. The ITP layer consists of three main components: (1) Connection-free Network, (2) Gateway Service, and (3) Remote Memory Access. In ITP, user messages are transmitted over a Connection-free Network. To manage storage traffic, functions such as congestion control, flow control, and timeout are implemented through a host-to-host communication channel inspired by the Bundler [23] architecture. The Gateway Service is considered a trusted agent, ensuring that the information transmitted through it is valid across different hosts, enabling ITP to support the splitting and replication of storage requests. Additionally, ITP introduces a method to access remote memory via the SmartNIC's DMA engine, further enhancing the flexibility and performance of storage applications.

The following sections will provide detailed descriptions of each component.

5.3.1 Connection-free Network

Previous research efforts, such as FlexTOE [116], have explored offloading connection-based protocols like TCP to SmartNICs. However, managing connection state—especially packet sequencing—introduces significant complexity and often limits the performance gains from offloading. Moreover, streaming protocols impose unnecessary processing overhead for storage applications, where transaction boundaries are well-defined and messages are fixed-sized.

In typical storage protocols, commands (requests) and completions are handled as distinct trans-

actions, and data transfers may occur asynchronously. Because storage operations can be processed out of order, the in-order delivery guarantees provided by streaming protocols like TCP become unnecessary burdens.

ITP addresses these inefficiencies by adopting a **message-based, connection-free network design**. Storage traffic is decomposed into three independent, self-descriptive packet types:

- **Request packets**, containing identifiers (e.g., keys or block addresses) and sizes,
- **Completion packets**, referencing requests by their IDs, and
- **Data packets**, providing buffer addresses or memory keys, particularly for Remote DMA transfers.

Like UDP, ITP ensures that each packet is self-contained and does not rely on maintaining a connection. This design simplifies packet handling and allows SmartNICs to efficiently interpose on storage traffic without needing detailed knowledge of higher-level storage semantics.

Our primary focus in ITP is enabling dynamic request redirection (forwarding) at the network layer. Upon detecting congestion, failures, or policy triggers, a SmartNIC can modify a packet's destination address and forward it to an alternate node without host intervention. This approach builds on a key similarity between storage operations and existing network packet forwarding applications. In both cases, the core task involves examining packet headers, applying policy-based decisions, and dynamically rewriting routing metadata before forwarding the packet. Since these operations are already highly optimized in modern programmable network hardware, offloading storage-specific redirection tasks to the transport layer is both natural and efficient. Redirection significantly improves system resilience and performance by allowing storage systems to quickly react to dynamic network or storage conditions without incurring host-side delays or CPU overhead.

While our implementation prototypes and evaluates redirection, ITP's design naturally extends to support **replication** and **request splitting** as well. Replication involves duplicating a request and forwarding copies to multiple nodes, splitting would generate separate sub-requests targeting different nodes based on the request's characteristics. Although these advanced capabilities are not evaluated in our prototype, ITP's connection-free message structure and transport-layer interposition make them feasible within the same design framework.

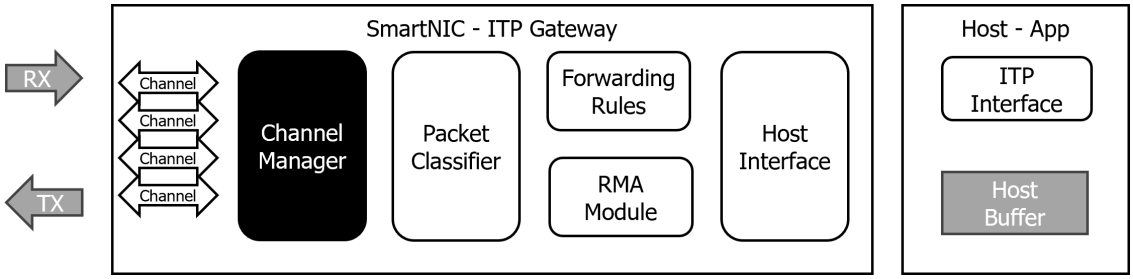


Figure 5.1: Structure of ITP Gateway

However, adopting a simple UDP-like design also introduces challenges, particularly in implementing congestion control, timeout detection, and secure memory access. These challenges are addressed through the **Gateway Service**, described next.

5.3.2 Gateway Service

The Gateway Service 5.1 plays a central role in managing and accelerating ITP traffic on SmartNICs. It introduces the concept of a Channel, a lightweight, trusted communication abstraction established between two hosts. Each Channel provides a reliable, congestion-controlled communication context between two hosts. Unlike traditional protocols such as TCP or RDMA, which require maintaining complex connection state for each flow, Channels offer a lightweight association that still guarantees reliable delivery and predictable performance. Any necessary stateful information, such as congestion control or timeout tracking, is encapsulated within the Channel itself and remains transparent to the applications. As a result, the application-facing interface can be completely stateless and message-based, allowing for simple and efficient communication models while the Gateway and SmartNICs handle reliability and flow management internally. Beyond providing a communication context, the Gateway Service also enables programmability: storage applications can configure timeout thresholds, congestion control policies, and traffic management strategies per Channel to suit workload-specific requirements. This flexibility enhances the system's ability to maintain end-to-end QoS across diverse operating conditions.

By relying on Channels, SmartNICs can manage, forward, or replicate storage traffic efficiently without incurring the overhead of maintaining full connection state for every communication flow. Channels also provide essential control hooks—such as delay measurements and timeout detection—that enable congestion management and failure recovery.

Within a host, multiple applications communicating with the same remote host can share a

single Channel, reducing management overhead while preserving application isolation through logical port mappings. Channels serve multiple critical purposes in ITP. First, they provide the necessary context to implement dynamic operations such as forwarding, replication, and storage-aware traffic shaping at the transport layer, enabling SmartNICs to manage traffic efficiently based on storage-specific requirements. For example, similar to the approach proposed in Gimbal [87], ITP's Channel architecture allows fair scheduling across competing traffic streams, while optimizing for storage-specific characteristics such as operation type (read/write) or I/O size.

Second, Channels establish a trust boundary that simplifies host processing and ensures secure traffic handling across forwarding operations. In traditional protocols, forwarding a request between hosts would typically require the host to terminate the request, verify its contents at the storage stack, and generate a new outbound request. This re-validation and regeneration introduce significant latency and CPU overhead. In contrast, ITP's Channel model guarantees that packets transmitted through the Gateway Service are trusted, even when their destination is dynamically changed. As a result, SmartNICs and remote hosts can accept and process forwarded or redirected packets without additional validation, streamlining the control path and allowing storage operations to proceed with minimal overhead. This design is crucial for maintaining high throughput and low latency in dynamic, large-scale storage systems where frequent redirection and replica selection are required.

Third, Channels carry essential control information that enables the Gateway Service to dynamically adapt transmission behavior based on real-time network conditions. Each Channel collects metadata such as round-trip time (RTT) measurements, congestion states, and timeout events, which are critical for maintaining predictable Quality of Service (QoS) in disaggregated storage systems. By continuously monitoring these metrics, the Gateway Service performs fine-grained congestion control and fair scheduling across competing Channels, avoiding bandwidth contention and maximizing the utilization of the NIC's network interface bandwidth, even under dynamic and bursty traffic conditions. Timeout tracking further enables rapid failure detection and recovery, minimizing the impact of node or network failures on ongoing storage operations. Through this control plane integration, Channels provide the feedback necessary to optimize end-to-end performance, maintain service fairness, and improve system resilience under varying workloads and network conditions.

Channels can be configured as either plain or secure, depending on deployment needs. In trusted

data center environments, plain Channels maximize performance by eliminating encryption overhead. In contrast, hybrid approaches can be used where only sensitive packets—such as commands or completions—are transmitted over secure Channels, balancing security and efficiency.

In addition to providing control and scheduling through Channels, the Gateway Service is responsible for the early classification and dispatching of incoming ITP packets into the appropriate SmartNIC processing pipelines. Based on packet type and metadata, the Gateway directs each packet to specialized modules such as remote memory access (RMA), dynamic redirection using policy tables, or application port mapping. Early classification enables SmartNICs to fully utilize their programmable datapath capabilities, maximizing parallelism and minimizing processing bottlenecks. Moreover, this design can leverage existing packet classification offload features provided by modern SmartNICs. For example, NVIDIA's DOCA Flow architecture offers hardware-accelerated packet parsing and flow classification, allowing applications to steer packets directly into custom processing pipelines with minimal software overhead. By integrating with such capabilities, ITP's Gateway Service achieves true line-rate handling of storage traffic while further reducing host CPU intervention, enabling efficient, scalable deployment on today's SmartNIC platforms. Furthermore, the Gateway Service acts as a bridge between generic network protocols (e.g., Ethernet, IP) and the proprietary internal architectures of SmartNICs. While generic message-based interfaces such as UDP sockets are widely used, they are often inadequate for meeting the stringent throughput and latency requirements of disaggregated storage systems. Their kernel-based processing paths introduce significant overheads, including data copying, system call latencies, and limited concurrency control, which collectively become bottlenecks for high-performance storage operations. To overcome these challenges, the ITP host interface must be capable of asynchronous, zero-copy communication, minimizing context switches and memory operations along the data path. Additionally, providing a user-level networking interface is critical to further reduce access latency and maximize the performance benefits of SmartNIC-based offloading. By bridging the gap between general-purpose network infrastructure and specialized, efficient SmartNIC processing pipelines, the Gateway Service enables ITP to fully exploit modern hardware capabilities while maintaining broad compatibility and portability.

5.3.3 Remote Memory Access

Remote DMA to the host memory across the network is a fundamental technique for enabling high-throughput, low-latency access to remote data. By allowing direct memory operations over the network without host CPU involvement, Remote DMA significantly reduces access latency, minimizes data copying overhead, and improves the efficiency of systems. These properties are especially critical in disaggregated storage architectures, where maintaining fast, predictable access to remote data is essential for achieving performance comparable to local storage devices.

Conventional RDMA protocols, such as those based on queue pairs and memory region registration (e.g., Verbs API), tightly bind access permissions to specific communication endpoints. Each queue pair manages its own set of memory keys, and remote memory operations are allowed only within a connection context that has been explicitly established between two hosts. This model fundamentally limits flexibility in disaggregated storage environments, where dynamic request routing, replica reads, and failure handling require more fluid access to remote memory across many nodes. Moreover, forwarding or relaying RDMA requests at the network layer is practically infeasible under traditional designs. Some recent proposals, such as RedN [102] and Hyperloop [61], attempt to interconnect RDMA operations by chaining lightweight memory accesses across nodes. However, these approaches still require setting up an RDMA command chain in advance and lack the flexibility needed to support arbitrary, dynamic storage operations that may be redirected or replicated at runtime.

To overcome these limitations, ITP introduces a lightweight, flexible Remote Memory Access model integrated into the transport layer. In ITP, memory access is authorized based on Channel-level trust rather than queue pair connections. When a remote memory request is received through a trusted Channel, the associated memory key is considered valid, and the target host grants direct access to the specified memory region. This model decouples memory protection from static connection management and allows SmartNICs to handle remote memory operations dynamically based on transport-layer policies.

This Channel-based RMA design enables important optimizations in storage disaggregation. For example, when a replica read is required, a SmartNIC can forward the original request packet—which includes the memory key and target address—to an alternate replica node with only minimal mod-

ifications to the network header, such as updating the destination address and transport metadata. Because ITP embeds all required access information within the self-descriptive message, the replica host can immediately validate the request based on Channel-level trust and execute the remote memory operation without additional coordination or permission setup. This streamlined forwarding process reduces end-to-end latency, avoids extra memory copies at intermediate hosts, and simplifies the data path. Furthermore, direct remote memory operations free host CPUs from handling unnecessary data forwarding tasks, improving overall system efficiency and scalability.

Remote Memory Access operations in ITP are initiated through the same message-based transport interface used for normal storage requests, preserving a unified communication model. To initiate an RMA operation, the host application sends an RMA request message to the remote host, containing the necessary metadata such as memory keys and buffer addresses. Upon arrival, the SmartNIC classifies the packet based on its RMA type and dispatches it to the Remote Memory Access module within the processing pipeline. The RMA module translates the request into network transactions appropriate for direct remote memory access. For RMA write operations, the SmartNIC generates network packets containing both the RMA metadata and the data payload. For RMA read operations, the SmartNIC generates read request packets targeting the specified remote memory region.

At the receiving side, the remote SmartNIC performs local DMA operations directly on the specified memory buffers without host CPU intervention. After the local DMA completes, the SmartNIC generates a response packet. For RMA read operations, this response includes the requested data; for RMA write operations, it contains a completion acknowledgment. Upon receiving the response, the originating SmartNIC's RMA module generates an RMA completion message, which is delivered back to the host application via the asynchronous ITP transport interface. This offloaded completion handling allows the application to receive notification of RMA results without introducing blocking or context switch overhead.

By embedding this fully offloaded RMA workflow into the transport layer, ITP ensures low latency, high throughput remote memory access with minimal host involvement, providing a practical foundation for efficient, scalable disaggregated storage systems.

With the connection-free transport, the Gateway Service, and the integrated Remote Memory Access model, ITP provides a complete framework for building flexible, high-performance transport

layers optimized for disaggregated storage systems. These design elements enable SmartNICs to offload critical storage operations at the network layer, reducing host CPU involvement, improving latency, and supporting scalable, resilient system architectures. In the next section, we describe the prototype implementation of ITP on a SmartNIC platform and demonstrate how these design concepts are realized in practice.

5.4 Implementation

5.4.1 SmartNIC Platform

The prototype implementation of ITP was developed on the NVIDIA BlueField-3 (BF3) SmartNIC platform. BF3 combines high-performance hardware acceleration engines with a programmable software stack, making it well-suited for offloading transport-layer functionality traditionally handled by host CPUs.

At the hardware level, BF3 integrates multiple Arm Cortex-A78 cores, a programmable packet processing pipeline, a set of high-performance DMA engines, and a PCIe Gen4 interface for host connectivity. For network communication, BF3 provides dual 200 GbE Ethernet ports with built-in support for RDMA over Converged Ethernet (RoCE) and other advanced transport protocols.

To support flexible packet handling and SmartNIC-based offloading, BF3 offers several critical hardware features: a programmable parsing engine capable of analyzing packet headers and extracting metadata for flow steering; a flow classification engine, allowing SmartNIC software to efficiently dispatch packets to appropriate processing modules based on transport-layer policies; and high-throughput DMA engines capable of performing local and remote memory operations with minimal latency and CPU overhead.

On the software side, the prototype leveraged the NVIDIA DOCA SDK, which provides APIs for configuring flow tables, managing memory regions, controlling DMA transactions, and interacting with hardware-accelerated networking features. Key components of the prototype, including packet classification, Gateway Service processing, and Remote Memory Access modules, were developed as user-space applications running on the SmartNIC's Arm cores, interacting directly with the DOCA libraries and hardware interfaces.

In the host-side software stack, the prototype leverages the Storage Platform Development Kit (SPDK) to implement the storage application. SPDK provides a user-space storage framework

optimized for high-throughput and low-latency operation. It is widely adopted in cutting-edge disaggregated storage systems, particularly as a backend for NVMe-oF targets, due to its minimal feature set and focus on performance-critical paths. In the prototype, SPDK enables direct, user-level construction and transmission of ITP messages to the SmartNIC without relying on heavy kernel intervention. Using SPDK as the storage application baseline allows for a fair and realistic performance comparison with state-of-the-art disaggregated storage solutions, ensuring that the evaluation reflects the efficiency of ITP itself rather than limitations of the host software.

5.4.2 SmartNIC-Side Implementation

The Gateway Service and SmartNIC processing pipeline for ITP were developed on top of the NVIDIA DOCA software platform. To enable efficient packet classification and dispatching, the prototype leverages DOCA Flow APIs, which provide a hardware-accelerated interface for managing flow tables and steering incoming packets into programmable software pipelines.

ITP packets are encapsulated inside UDP frames to fully utilize the existing packet parsing and hardware offloading capabilities of the BlueField-3 (BF3) SmartNIC. While BF3 does not support direct hardware parsing of user-defined payload fields, it provides full hardware support for parsing and manipulating Ethernet, IP, and UDP headers. By wrapping ITP messages within UDP packets and assigning a dedicated UDP port number, the system ensures that ITP traffic can be reliably identified and steered entirely using hardware flow classification.

There are two types of messages in the prototype: ITP network packets, which are transferred between hosts across Channels, and ITP local messages, which are exchanged between the SmartNIC and the host system via the PCIe interface.

Each ITP network packet includes a source address field within its transport header, which records the original sender's information. This design is critical for preserving request context during transport-level redirection. When the Gateway Service forwards a request or reconstructs a local message for the host, it uses this source address field to correctly assemble the new ITP message header.

Upon reception, the SmartNIC hardware first classifies incoming packets based on the UDP destination port using DOCA Flow. Packets identified as ITP traffic are redirected to the Gateway Service software pipeline, where further classification and processing occur. Depending on the

operation type encoded in the ITP header, packets are dispatched to the appropriate module for forwarding, replication, Remote Memory Access, or completion generation.

This integration between hardware-accelerated flow classification and software-driven message handling provides the foundation for efficient, line-rate processing of ITP traffic on the SmartNIC, while maintaining the flexibility necessary for supporting dynamic storage operations.

Forwarding Module

The forwarding module is responsible for dynamically redirecting incoming ITP packets at the transport layer based on runtime forwarding policies. When a request arrives, the Gateway Service examines policy tables associated with the corresponding Channel to determine whether redirection is necessary. Policy decisions can be triggered by various conditions, such as load balancing needs, node failures, or congestion control strategies.

ITP supports flexible forwarding policy management, allowing both the SmartNIC and the host software stack to program forwarding entries. In the prototype, only host-driven management is implemented. The host application can register or deregister forwarding policies by sending a control message through the same message-based ITP interface used for normal requests. Upon receiving a control message, the Gateway Service processes it and updates the forwarding table accordingly. While hardware-based flow steering typically provides the best performance, NVIDIA BF3 SmartNICs exhibit noticeable delay when programming new hardware flow entries. To mitigate this, the prototype maintains a lightweight software-managed forwarding table as a fast transitional solution, ensuring immediate policy enforcement until the hardware rules take effect.

If a packet requires redirection, the forwarding module modifies its network-layer headers, including the destination IP address and UDP port, to point to the selected new target node. The key design principle is to ensure that the transport-layer context carried within the ITP header—particularly the original source address—is preserved during forwarding. This allows the target replica, even after redirection, to generate its response and send it directly back to the original requester, without requiring intermediate relays or additional forwarding paths. Maintaining this end-to-end context enables efficient and low-latency completion handling across dynamic redirection events.

After header modification, the forwarding module reinjects the updated packet into the Smart-

NIC's transmit pipeline for delivery to the new destination. Because redirection is performed entirely within the SmartNIC datapath, forwarding decisions can be made at line rate without host CPU involvement. This fast and adaptive request routing capability is critical for maintaining low latency, predictable response times, and consistent QoS in disaggregated storage environments.

Remote Memory Access Module

Before executing any Remote Memory Access operations, the RMA module relies on a memory registration (MR) process. Similar to memory registration in conventional RDMA systems, the host application registers memory regions that may be the target of remote accesses. Memory registration ensures that the SmartNIC is prepared to access the designated memory safely according to SmartNIC-specific framework requirements, such as those enforced by NVIDIA DOCA. Additionally, memory registration enables the RMA module to validate incoming RMA memory keys, ensuring that only authorized accesses are permitted.

The RMA module is responsible for executing remote memory operations initiated either by ITP network packets or by ITP local messages from the host. When the Gateway Service classifies an incoming request as an RMA operation, it dispatches it to the RMA module for further processing. Depending on the source of the request, the processing path differs slightly. When an ITP message from the local host triggers an RMA operation, the SmartNIC interprets the request and determines whether to initiate a network transaction or a local DMA. For RMA read operations, the SmartNIC generates new ITP network packets containing remote read requests targeted at the remote address provided by the host application. For RMA write operations, the SmartNIC performs a local DMA transfer from the host memory into the SmartNIC's memory space. This design ensures that the host memory is accessed exactly once, regardless of retransmissions or network-induced delays such as congestion. While this approach introduces a one-time memory copy between the host and the SmartNIC, its impact can be effectively hidden by pipelining multiple DMA operations and network transmissions. The high memory bandwidth available on modern SmartNICs typically exceeds network bandwidth by a factor of two or more, enabling efficient overlap of memory and network operations.

Similarly, when an RMA request is received as an incoming ITP packet from a remote node, the RMA module parses the transport header to extract the remote memory key, target buffer address,

access size, and operation type. For incoming write requests, the SmartNIC performs a local DMA write operation, placing the incoming data directly into the host memory region specified by the target address. For incoming read requests, the SmartNIC performs a local DMA read operation from the specified memory region and generates an ITP response packet containing the retrieved payload.

To handle large RMA operations that exceed the network's MTU size, the RMA module segments the data into multiple network packets. It maintains internal state to track outstanding segments associated with each active RMA operation. For RMA read operations, the module tracks the completion of each read segment as responses arrive from the remote node. For RMA write operations, the module tracks acknowledgment of each transmitted data chunk. The RMA module generates a local completion message to the host application only after all associated segments of an RMA operation have been successfully completed. All local DMA operations and network transactions are performed without host CPU involvement, ensuring low-latency handling of remote memory operations. Completion events are delivered asynchronously via the ITP interface, allowing the host application to process completions efficiently without blocking or data staging.

5.4.3 Host ITP Interface

In the prototype, communication between the host application and the SmartNIC is built on top of the DOCA Communication Channel abstraction. The DOCA Communication Channel provides a high-bandwidth, asynchronous messaging interface that enables efficient communication over the PCIe bus without consuming external network bandwidth. This design allows the host to interact with the SmartNIC Gateway Service with minimal latency and without interfering with normal network traffic handled by the SmartNIC's Ethernet ports.

The host application submits ITP messages, including standard storage operations and Remote Memory Access (RMA) operations, by enqueueing them into a FIFO-style communication queue. Each ITP message contains a transport-level header and may optionally include application-specific payloads, such as NVMe-oF commands or completions. The SmartNIC Gateway Service polls the communication queue periodically to detect and process incoming messages asynchronously, without relying on interrupts or explicit signaling mechanisms.

Each communication channel in the host ITP interface binds to an application thread, allowing

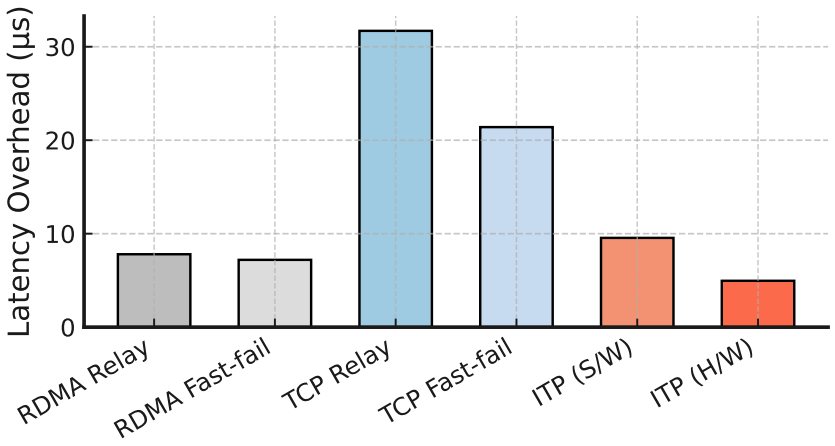


Figure 5.2: Storage redirection latency overhead comparison across different approaches. ITP (H/W) achieves the lowest overhead among all methods, significantly outperforming software-based relaying and fast-fail approaches.

ITP messages and completions to be delivered directly without requiring software steering at the host. Applications retrieve completion events by polling dedicated completion queues associated with each channel, avoiding interrupt-based overhead and enabling highly concurrent, non-blocking operation. The ITP host interface is integrated into the SPDK transport layer framework at the same abstraction level as existing RDMA and TCP transports. As a result, SPDK-based applications can flexibly attach various types of backend storage devices, such as DRAM or SSDs, or build higher-level abstractions like BLOB stores or key-value stores, while leveraging the efficient host-to-SmartNIC communication path provided by ITP.

5.5 Evaluation

5.5.1 Storage Redirection

To evaluate the efficiency of storage redirection using ITP, we measure the additional latency introduced by different redirection mechanisms. As shown in Figure 5.2, traditional approaches such as RDMA relay and TCP relay incur significant overhead, with TCP relay introducing over 30us of additional latency. Fast-fail techniques improve performance slightly but still suffer from noticeable delays. In contrast, ITP achieves much lower overhead, especially when leveraging hardware-based forwarding, reducing redirection latency to under 5us. This result highlights the benefit of offloading dynamic request routing directly into the transport layer, minimizing the impact on end-to-end

Table 5.1: Programming Forward Table Latency

Method	Insertion Latency (μs)
S/W Forward Table	7.0
H/W Flow Rule	68.1

system latency.

5.5.2 Forwarding Latency

We measure the latency of programming new forwarding rules in both the software-managed and hardware-accelerated forwarding paths. As shown in Table 5.1, inserting an entry into the software forwarding table incurs minimal latency, around 7us, while programming a hardware flow rule takes approximately 68us. To understand the impact of these latencies, we note that a typical small block I/O (e.g., 4KB) in a disaggregated storage system takes between 50 and 100us to complete. Thus, if only a single I/O request is in flight, a hardware flow rule can usually be installed in time before subsequent requests arrive. However, under highly parallelized workloads with many outstanding I/Os, multiple requests could pass through before the hardware rule installation completes. The software forwarding table, with its single-digit microsecond installation latency, effectively prevents such a case, ensuring that incoming requests are redirected according to updated policies without delay. Even in the worst case, where some I/Os are misdirected to congested storage during the transition, these operations are still correctly processed with slightly higher latency, preserving system correctness and progress.

5.5.3 RMA Latency

To evaluate the latency of RMA operations, we measured the cumulative latency at each key processing step inside both the initiator-side and target-side SmartNICs. Figure 5.3 shows the initiator-side results. The DMA submission occurs within approximately 1.1us after receiving the request, and local DMA completes within 2.9us. Sending the RMA completion back to the target adds minimal additional delay, resulting in a total cumulative latency of around 3.7us at the initiator side. These results demonstrate that RMA tasks can be completed within a single-digit microsecond timeframe at the initiator side, highlighting the efficiency of high-bandwidth, low-latency DMA engines available on modern SmartNICs. Compared to conventional RDMA solutions, where a 4KB I/O operation targeting a NULL device typically completes in around 7.6us on average (capturing

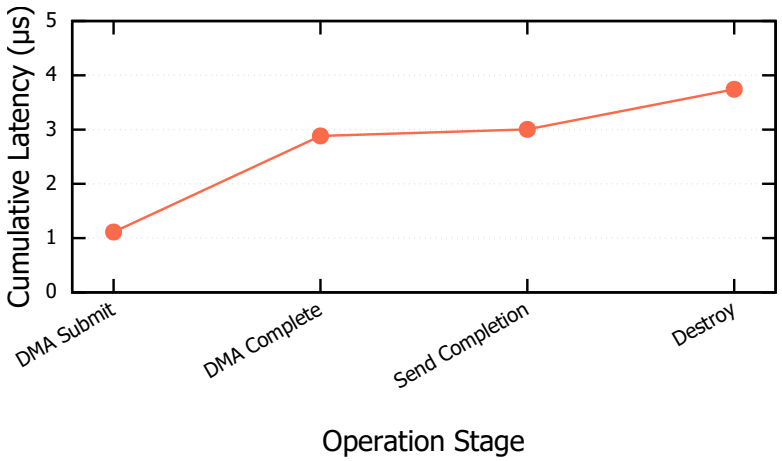


Figure 5.3: Cumulative latency breakdown of RMA write operations at the initiator-side SmartNIC.

only network and protocol stack overhead), the ITP RMA processing on the SmartNIC achieves sufficiently low latency . This result confirms that high-performance remote access can be realized through SmartNIC-based processing.

Figure 5.4 presents the target-side results. The local DMA submission and completion occur similarly quickly, within approximately 3us. However, a significant delay is observed between sending the RMA request and receiving the corresponding completion, caused by network transmission and processing delays on the NVIDIA BF3 SmartNIC platform. This network-related delay dominates the cumulative latency at the target side, as visible in the broken y-axis figure. We defer a detailed discussion of this observed behavior to Section 5.5.4.

5.5.4 Limitations of the Prototype

The ITP prototype is built on the NVIDIA BlueField-3 (BF3) SmartNIC platform, utilizing its DOCA SDK and SmartNIC ARM cores for programmability. While this platform offers flexibility and accessibility, it also introduces several practical limitations that affect the end-to-end performance of the prototype.

First, we observe significant latency in the network packet processing path, particularly at the target-side SmartNIC. As shown in the RMA latency breakdown, there is a noticeable delay between sending an RMA request and receiving the corresponding completion, which dominates the overall latency on the target side. This delay appears to stem from the software stack and packet scheduling

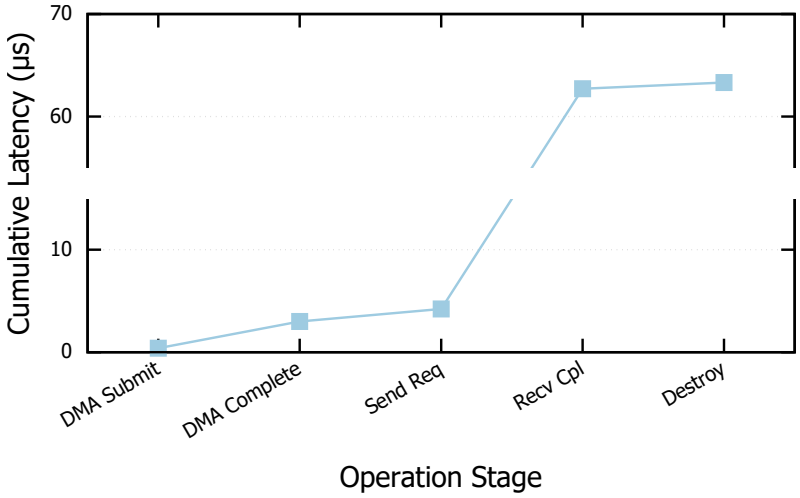


Figure 5.4: Cumulative latency breakdown of RMA write operations at the target-side SmartNIC. A broken y-axis is used for readability.

behavior within the BF3 platform, rather than any fundamental hardware limitation or firmware overhead. With further exploration of the platform software, we may be able to better coordinate packet generation and completion signaling to reduce this delay without requiring architectural changes.

Second, programming hardware flow rules in the SmartNIC forwarding engine introduces substantial latency. As measured in Section 5.5.2, installing a hardware rule takes approximately 68us, compared to only 7us for a software forwarding table update. To mitigate this, the prototype maintains a fast software-based forwarding table for immediate redirection, while hardware rule installation occurs opportunistically in the background. Future platforms with faster hardware rule programming or hybrid approaches could further optimize redirection performance.

Third, the communication channel between the SmartNIC and the host, provided by the BF3 DOCA framework, is designed primarily for high throughput rather than low latency. While effective for bulk data transfers, it introduces non-negligible latency when handling the small, frequent messages typical in ITP, such as storage requests and completions. A future platform with a latency-centric channel design, such as lightweight notification queues or direct MMIO signaling, would further enhance ITP's responsiveness for storage applications.

Despite these practical limitations, the evaluation results validate the fundamental design principles of ITP and demonstrate the feasibility of SmartNIC-based transport-layer offloading for dis-

aggregated storage systems.

5.6 Conclusion

This work presented the design, implementation, and evaluation of the Interposable Transport Protocol (ITP), a general-purpose transport-layer abstraction for disaggregated storage systems. ITP enables SmartNICs to interpose on storage traffic at the network layer, allowing critical operations such as redirection, replication, and remote memory access to be performed efficiently without host involvement. We identified key requirements for transport protocols in storage networks, including support for connectionless messaging, hardware-assisted DMA, and in-network programmability.

The ITP design was implemented on a programmable SmartNIC platform, leveraging the BlueField-3 architecture and DOCA software framework. The prototype demonstrates that forwarding and RMA operations can be performed with single-digit microsecond latency, achieving performance comparable to conventional RDMA solutions. Through microbenchmark evaluations, we showed that ITP supports efficient redirection, fast software-managed flow control, and low-latency RMA operations. While the current prototype exposes several limitations related to software stack behavior and hardware programming interfaces, these are not fundamental and can be addressed through improved platform integration.

Beyond the current SmartNIC-based implementation, ITP can also be extended to programmable network switches, as its design adheres to standard packet processing semantics. Deploying ITP on switches would enable even lower-latency operations and broader visibility across the storage fabric. Such a deployment could act as a centralized controller to coordinate traffic redirection and replication at the rack or cluster scale. Alternatively, a collaborative architecture combining SmartNICs and switches could support distributed decision-making, where the switch handles global coordination and the SmartNICs manage local enforcement. This direction opens opportunities for more scalable, fault-tolerant, and adaptive disaggregated storage systems that integrate deeper with the underlying network fabric.

Overall, the ITP prototype demonstrates the practicality and performance benefits of offloading transport-layer logic to SmartNICs in disaggregated storage environments. The results support the feasibility of the transport layer that enables network-level adaptation and coordination across diverse storage workloads.

Chapter 6

Conclusion and Future Work

The increasing adoption of disaggregated architectures in modern datacenters demands scalable and efficient mechanisms to manage shared storage resources across multiple tenants. This dissertation presents a comprehensive exploration of architectural and software techniques to enhance the performance, fairness, and manageability of disaggregated storage systems through hardware innovations.

We began by identifying the limitations of conventional SSDs and NVMe-over-Fabrics deployments in multi-tenant disaggregated environments. We demonstrated that SmartNIC-based disaggregation offers a power-efficient and cost-effective alternative to x86-based storage nodes, while still being capable of achieving high throughput and microsecond-scale latency. However, their limited compute resources and lack of mature multi-tenancy mechanisms motivated the need for more intelligent software-layer solutions.

To address these challenges, we introduced **Gimbal** (Chapter 3), a software storage switch that transforms SmartNIC JBOFs into multi-tenant-aware storage nodes. The novel design—comprising delay-based congestion control, hierarchical I/O scheduling, write cost modeling, and credit-based flow control—enables fair and efficient sharing of SSD resources while staying within the limited compute budget of SmartNICs. Our evaluation demonstrated that Gimbal not only improves fairness across mixed workloads but also achieves higher throughput and reduced tail latency compared to prior solutions.

We further explored architectural limitations of existing NVMe SSD interfaces, particularly in the context of Zoned Namespace SSDs (ZNS). To overcome the inflexibility and interference challenges in conventional SSDs, we proposed **eZNS** (Chapter 4), an Elastic Zoned Namespace abstraction. eZNS

introduces a novel ballooning-based resource management scheme that dynamically adjusts zone configurations according to application needs. It provides global and local overdrive mechanisms, reclaim strategies, and coordinated I/O scheduling to adapt to dynamic workload behavior while preserving QoS and device utilization. The evaluation confirmed that eZNS significantly improves performance isolation and write bandwidth efficiency over static configurations.

Lastly, recognizing the growing role of SmartNICs as programmable dataplane processors, we introduced the **Interposable Transport Protocol (ITP)** (Chapter 5). ITP enables in-network acceleration of storage routing tasks (e.g., redirection, replication, and mapping) that traditionally require host software intervention. Our design decouples protocol semantics from specific endpoints, allowing SmartNICs to act as in-path routers for disaggregated storage operations. In addition, we showed that ITP RMA operations can be completed in sub-10 μs with high bandwidth using modern SmartNIC DMA engines, achieving near-RDMA performance without full RDMA stack dependency. Though the current prototype has limitations, the results suggest the protocol's feasibility for real deployments and underscore the potential of NIC-side storage offload.

In summary, this dissertation demonstrates that by rethinking both hardware interfaces and software mechanisms for SmartNICs and NVMe SSDs, it is possible to construct scalable, fair, and high-performance disaggregated storage systems. The proposed innovations open up new pathways for building composable infrastructure in next-generation datacenters.

Bibliography

[1] Netronome Agilio SmartNIC. `https://www.netronome.com/products/agilio-cx/`, 2020.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, 2008.

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, volume 57. Boston, USA, 2008.

[4] Aditya Akella, Amin Vahdat, Arjun Singhvi, Behnam Montazeri, Dan Gibson, Hassan Wassel, Joel Scherpelz, Milo M. K. Martin, Monica C Wong-Chan, Moray Mclaren, Prashant Chandra, Rob Cauble, Sean Clark, Simon Sabato, and Thomas F. Wenisch. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, page 708–721, New York, NY, USA, 2020.

[5] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: approximating push-in first-out behaviors using strict-priority queues. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020.

[6] Alpha Data ADM-PCIE-9V3 FPGA SmartNIC. `https://www.alpha-data.com/dcp/products.php?product=adm-pcie-9v3`, 2020.

[7] Thomas E Anderson, Susan S Owicki, James B Saxe, and Charles P Thacker. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems (TOCS)*, 11(4):319–352, 1993.

[8] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 93–109, 2020.

[9] AWS Nitro System. `https://aws.amazon.com/ec2/nitro/`, 2020.

[10] Jens Axboe. Linux block io—present and future. In *Ottawa Linux Symp*, 2004.

[11] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. What You Can't Forget: Exploiting Parallelism for Zoned Namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022.

[12] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. {SILK}: Preventing latency spikes in {Log-Structured} merge {Key-Value} stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[14] Matias Bjørling. From open-channel ssds to zoned namespaces. In *Linux Storage and Filesystems Conference (Vault 19)*, volume 1, 2019.

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

[16] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[17] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, 2013.

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

[19] Mellanox BlueField-2 SmartNIC. `https://www.mellanox.com/products/bluefield2-overview`, 2020.

[20] Lawrence S. Brakmo and Larry L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[21] Brocade G620 Switch. `https://www.broadcom.com/products/fibre-channel-networking/switches/g620-switch`, 2021.

[22] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proceedings of the IEEE*, 105(9):1666–1704, 2017.

[23] Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Site-to-site internet traffic control. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 574–589, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Feng Chen, Binbing Hou, and Rubao Lee. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Trans. Storage*, may 2016.

[25] Feng Chen, Binbing Hou, and Rubao Lee. Internal parallelism of flash memory-based solid-state drives. *ACM Transactions on Storage (TOS)*, 12(3):1–39, 2016.

[26] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, 2009.

[27] Feng Chen, David A Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):181–192, 2009.

[28] Feng Chen, Tian Luo, and Xiaodong Zhang. {CAFTL}: A {Content-Aware} flash translation layer enhancing the lifespan of flash memory based solid state drives. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[29] Eric Chung, Andreas Nowatzyk, Tom Rodeheffer, Chuck Thacker, and Fang Yu. An3: A low-cost, circuit-switched datacenter network. *Technical Report MSR-TR-2014-35, Microsoft Research*, 2014.

[30] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5-6):332–343, 2009.

[31] Cisco MDS 9000 Series Multilayer Switches. `https://www.cisco.com/c/en/us/products/storage-networking/mds-9000-series-multilayer-switches/index.html`, 2021.

[32] 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*, pages 143–154, 2010.

[33] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 1989.

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

[35] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

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

[37] Flexible I/O Tester (FIO). `https://github.com/axboe/fio`, 2022.

[38] 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 *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[39] Broadcom FlexSPARX acceleration subsystem. `https://docs.broadcom.com/doc/1211168571391`, 2020.

[40] Fungible FS1600. `https://www.storagereview.com/review/fungible-fs1600-pushes-hyperscale-storage-to-the-data-center`, 2022.

[41] Pawan Goyal, Harrick M Vin, and Haichen Chen. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, 1996.

[42] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*, 2009.

[43] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017.

[44] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[45] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[46] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. Linnos: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[47] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. Multi-queue fair queuing. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*, pages 301–314, 2019.

[48] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Chao Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, 2012.

[49] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[50] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. Accelerating rocksdb for small-zone zns ssds by parallel i/o mechanism. In *Proceedings of the 23rd International Middleware Conference Industrial Track*, Middleware Industrial Track '22, page 15–21, New York, NY, USA, 2022. Association for Computing Machinery.

[51] Industry Outlook: NVMe and NVMe-oF For Storage. `https://www.iol.unh.edu/news/2018/02/08/industry-outlook-nvme-and-nvme-storage`, 2018.

[52] Mellanox Innova-2 Flex Open Programmable SmartNIC. `https://www.mellanox.com/products/smartnics/innova-2-flex`, 2020.

[53] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[54] Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, 2013.

[55] Myoungsoo Jung and Mahmut Kandemir. Revisiting widely held ssd expectations and rethinking system-level implications. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):203–216, 2013.

[56] Dongku Kang, Minsu Kim, Su Chang Jeon, Wontaeck Jung, Jooyong Park, Gyosoo Choo, Dong-kyo Shim, Anil Kavala, Seung-Bum Kim, Kyung-Min Kang, et al. 13.4 a 512gb 3-bit/cell 3d 6 th-generation v-nand flash memory with 82mb/s write throughput and 1.2 gb/s interface. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 216–218. IEEE, 2019.

[57] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for nand flash memory. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, 2006.

[58] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. Durable write cache in flash memory ssd for relational and nosql databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 529–540, 2014.

[59] Elie Kfoury, Samia Choueiri, Ali Mazloum, Ali AlSabeh, Jose Gomez, and Jorge Crichigno. A comprehensive survey on smartnics: Architectures, development models, applications, and research directions, 2024.

[60] Ali Khakifirooz, Sriram Balasubrahmanyam, Richard Fastow, Kristopher H Gaewsky, Chang Wan Ha, Rezaul Haque, Owen W Jungroth, Steven Law, Aliasgar S Madraswala, Binh Ngo, et al. 30.2 a 1tb 4b/cell 144-tier floating-gate 3d-nand flash memory with 40mb/s program throughput and 13.8 gb/mm 2 bit density. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 64, pages 424–426. IEEE, 2021.

[61] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, 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 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 297–312, New York, NY, USA, 2018. Association for Computing Machinery.

[62] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[63] Moosung Kim, Sung Won Yun, Jungjune Park, Hyun Kook Park, Jungyu Lee, Yeong Seon Kim, Daehoon Na, Sara Choi, Youngsun Song, Jonghoon Lee, Hyunjun Yoon, Kangbin Lee, Byunghoon Jeong, Sanglok Kim, Junhong Park, Cheon An Lee, Jaeyun Lee, Jisang Lee, Jin Young Chun, Joonsuc Jang, Younghwi Yang, Seung Hyun Moon, Myunghoon Choi, Wontae Kim, Jungsoo Kim, Seokmin Yoon, Pansuk Kwak, Myunghun Lee, Raehyun Song, Sunghoon Kim, Chiweon Yoon, Dongku Kang, Jin-Yub Lee, and Jaihyuk Song. A 1tb 3b/cell

8th-generation 3d-nand flash memory with 164mb/s write throughput and a 2.4gb/s interface. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pages 136–137, 2022.

[64] 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 *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[65] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[66] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash= local flash. *ACM SIGARCH Computer Architecture News*, 45(1):345–359, 2017.

[67] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2020.

[68] H. T. Kung, Trevor Blackwell, and Alan Chapman. Credit-Based Flow Control for ATM Networks:Credit Update Protocol, Adaptive Credit Allocation, and Statistical Multiplexing. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, 1994.

[69] HT Kung and Alan Chapman. The fcvc (flow-controlled virtual channels) proposal for atm networks: A summary. In *1993 International Conference on Network Protocols*, pages 116–127. IEEE, 1993.

[70] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. {DC-Store}: Eliminating noisy neighbor containers using deterministic {I/O} performance and resource isolation. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 183–191, 2020.

[71] Damien Le Moal and Ting Yao. zonefs: Mapping {POSIX} file system interface to raw zoned block device accesses. In *Linux Storage and Filesystems Conference (Vault)*, Santa Clara, CA, February 2020. USENIX Association.

[72] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. {F2FS}: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[73] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanaël Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale Disaggregated Storage. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*, 2017.

[74] 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 Symposium on Operating Systems Principles*, 2017.

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

[76] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[77] Free Your Flash And Disaggregate. `https://www.lightbitslabs.com/blog/free-your-flash-and-disaggregate/`, 2018.

[78] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[79] Marvell LiquidIO SmartNICs. `https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics.html`, 2020.

[80] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. RAIL: Predictable, Low Tail Latency for NVMe Flash. *ACM Trans. Storage*, 18(1), 2022.

[81] Chun-Yi Liu, Yunju Lee, Myoungsoo Jung, Mahmut Taylan Kandemir, and Wonil Choi. Prolonging 3d nand ssd lifetime via read latency relaxation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 730–742, 2021.

[82] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2019.

[83] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[84] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. vfair: latency-aware fair storage scheduling via per-io cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[85] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2015.

[86] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From luna to solar: the evolutions of the compute-to-storage networks in alibaba cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 753–766, New York, NY, USA, 2022. Association for Computing Machinery.

[87] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling multi-tenant storage disaggregation on smartnic jbofs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 106–122, New York, NY, USA, 2021. Association for Computing Machinery.

[88] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[89] Jeonghoon Mo, Richard J La, Venkat Anantharam, and Jean Walrand. Analysis and comparison of tcp reno and vegas. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 3, pages 1556–1563, 1999.

[90] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[91] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017.

[92] Disaggregated or Hyperconverged, What Storage will Win the Enterprise? `https://www.nextplatform.com/2017/12/04/disaggregated-hyperconverged-storage-will-win-enterprise/`, 2017.

[93] The NVMe Base Specification. https://nvmexpress.org/developers/nvme-specification/, 2022.

[94] NVM Express over Fabrics Specification. https://nvmexpress.org/developers/nvme-of-specification/, 2020.

[95] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[96] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):1–23, 2008.

[97] Stan Park and Kai Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[98] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *FAST*, 2012.

[99] Chris Petersen, Wei Zhang, and Alexei Naberezhnov. Enabling nvme i/o determinism@ scale. *The 12th annual Flash Memory Summit*, 2018.

[100] 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):1–25, 2018.

[101] Radian Memory System RMS ZNS SSDs. https://www.radianmemory.com/zoned_namespaces/, 2022.

[102] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is turing complete, we just did not know it yet! In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 71–85, Renton, WA, April 2022. USENIX Association.

[103] RocksDB. http://rocksdb.org/, 2022.

[104] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[105] Lorenzo Rosa, Luca Foschini, and Antonio Corradi. Empowering cloud computing with network acceleration: A survey. *IEEE Communications Surveys & Tutorials*, 2024.

[106] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[107] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and Flexible Software Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[108] Samsung PM1731a ZNS SSDs. `https://news.samsung.com/global/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan`, 2022.

[109] Samsung PM1731a Review from STH. `https://www.servethehome.com/samsung-pm1731a-ssd-with-zns-support/`, 2022.

[110] Mark R Schibilla and Randy J Reiter. Garbage collection for solid state disks, April 24 2012. US Patent 8,166,233.

[111] Joel H Schopp, Keir Fraser, and Martine J Silbermann. Resizing memory with balloons and hotplug. In *Proceedings of the Linux Symposium*, volume 2, pages 313–319, 2006.

[112] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[113] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[114] The SCSI Protocol. `https://en.wikipedia.org/wiki/SCSI#cite_note-1`, 2022.

[115] Prateek Shantharama, Akhilesh S. Thyagaturu, and Martin Reisslein. Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies. *IEEE Access*, 8:132021–132085, 2020.

[116] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, Renton, WA, April 2022. USENIX Association.

[117] Kai Shen and Stan Park. {FlashFQ}: A fair queueing {I/O} scheduler for {Flash-Based}{SSDs}. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 67–78, 2013.

[118] Hojin Shin, Myounghoon Oh, Gunhee Choi, and Jongmoo Choi. Exploring Performance Characteristics of ZNS SSDs: Observation and Implication. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2020.

[119] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, 2019.

[120] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[121] David Shue and Michael J Freedman. From application requests to virtual iops: Provisioned key-value storage with libra. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[122] Chang Siau, Kwang-Ho Kim, Seungpil Lee, Katsuaki Isobe, Noboru Shibata, Kapil Verma, Takuya Ariki, Jason Li, Jong Yuh, Anirudh Amarnath, et al. 13.5 a 512gb 3-bit/cell 3d flash memory on 128-wordline-layer with 132mb/s write performance featuring circuit-under-array technology. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 218–220. IEEE, 2019.

[123] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[124] The Storage Performance Development Kit (SPDK). `https://spdk.io`, 2022.

[125] Theano Stavrinos, Daniel S. Berger, Ethan Katz-Bassett, and Wyatt Lloyd. Don't Be a Blockhead: Zoned Namespaces Make Work on Conventional SSDs Obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021.

[126] Broadcom Stingray PS1100R SmartNIC. `https://www.broadcom.com/products/storage/ethernet-storage-adapters-ics/ps1100r`, 2020.

[127] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. Mqsim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018.

[128] Nick Tehrany and Animesh Trivedi. Understanding NVMe Zoned Namespace (ZNS) Flash SSD Storage Devices, 2022.

[129] 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 Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[130] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. Understanding the impact of power loss on flash memory. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 35–40. IEEE, 2011.

[131] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance insulation for shared storage servers. In *FAST*, 2007.

[132] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, dec 2003.

[133] 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 Ninth European Conference on Computer Systems*, 2014.

[134] Shunzhuo Wang, Fei Wu, Chengmo Yang, Jiaona Zhou, Changsheng Xie, and Jiguang Wan. Was: Wear aware superblock management for prolonging ssd lifetime. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[135] Watts up? PRO. `http://www.idlboise.com/sites/default/files/WattsUp_Pro_ES.pdf`, 2020.

[136] Western Digital Ultrastar ZNS SSDs. `https://www.westerndigital.com/solutions/zns`, 2022.

[137] What is Composable Disaggregated Infrastructure? `https://blog.westerndigital.com/what-is-composable-disaggregated-infrastructure/`, 2019.

[138] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Manu Awasthi, Tameesh Suri, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance characterization of hyperscale applicationson on nvme ssds. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2015.

[139] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[140] Jinfeng Yang, Bingzhe Li, and David J Lilja. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 5(1):1–28, 2020.

[141] Jinfeng Yang, Bingzhe Li, and David J Lilja. Heuristicdb: a hybrid storage database system using a non-volatile memory block device. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–12, 2021.

[142] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack Your Log On My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014.

[143] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. Garbage collection and wear leveling for flash memory: Past and future. In *2014 International Conference on Smart Computing*, pages 66–73. IEEE, 2014.

[144] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, 2012.

[145] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, page 1, 2012.

[146] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of {SSDs} under power fault. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 271–284, 2013.