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

JAEHONG MIN, University of Washington, Seattle, United States
CHENXINGYU ZHAO, University of Washington, Seattle, United States
MING LIU, University of Wisconsin–Madison, Madison, United States
ARVIND KRISHNAMURTHY, University of Washington, Seattle, United States

Emerging Zoned Namespace (ZNS) SSDs, providing the coarse-grained zone abstraction, hold the potential to significantly enhance the cost efficiency of future storage infrastructure and mitigate performance unpredictability. However, existing ZNS SSDs have a static zoned interface, making them in-adaptable to workload runtime behavior, unscalable to underlying hardware capabilities, and interfering with co-located zones. Applications either under-provision the zone resources yielding unsatisfied throughput, create over-provisioned zones and incur costs, or experience unexpected I/O latencies.

We propose eZNS, an elastic-ZNS interface that exposes an adaptive zone with predictable characteristics. eZNS comprises two major components: a zone arbiter that manages zone allocation and active resources on the control plane, and a hierarchical I/O scheduler with read congestion control and write admission control on the data plane. Together, eZNS enables the transparent use of a ZNS SSD and closes the gap between application requirements and zone interface properties. Our evaluations over RocksDB demonstrate that eZNS outperforms a static zoned interface by 17.7% and 80.3% in throughput and tail latency, respectively, at most.

CCS Concepts: • **Hardware** → **External storage;** • **Computer systems organization** → **Cloud computing**;

Additional Key Words and Phrases: ZNS, SSD, Storage, Disaggregation

**ACM Reference Format:**
Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2024. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage* 20, 3, Article 16 (June 2024), 41 pages. https://doi.org/10.1145/3653716

## 1 INTRODUCTION

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 with users. Some storage systems address this by physically segregating tenants onto different SSD devices,

This work was supported in part by NSF grant CNS-2212193 and ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Authors' Contact Information: Jaehong Min, Computer Science, University of Washington, Seattle, Washington, United States; e-mail: jaehongm@cs.washington.edu; Chenxingyu Zhao, University of Washington, Seattle, Washington, United States; e-mail: cxyzhao@cs.washington.edu; Ming Liu, University of Wisconsin–Madison, Madison, Wisconsin, United States; e-mail: mgliu@cs.wisc.edu; Arvind Krishnamurthy, University of Washington, Seattle, Washington, United States; e-mail: arvind@cs.washington.edu.

ACM 1553-3077/2024/06-ART16
https://doi.org/10.1145/3653716

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 [31, 32, 58], leveraging features such as I/O Determinism [16] and **Open-Channel (OC)** SSD [9], 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 over-provisioning, 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, and (2) what tradeoffs 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 (stand-alone), and applications sharing a device with other tenants (co-located). Using carefully calibrated micro-benchmarks, 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 ZNS 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, whereas 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 the 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. However, write I/Os share a performance domain due to the write cache architecture of the SSD, which causes global

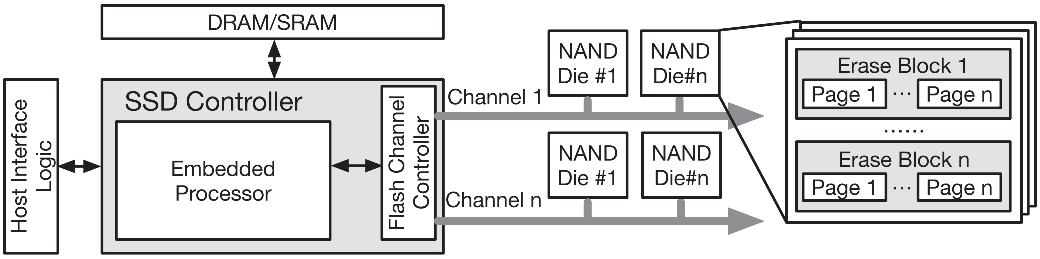


Fig. 1. The architecture of a conventional and ZNS SSD.

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 (Section 5) over micro-benchmarks and RocksDB.

In summary, our study not only empirically demonstrates the benefits of using a coarse-grained ZNS 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 are 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 quality of service without sacrificing the lifetime of devices.

## 2 BACKGROUND AND MOTIVATION

This section reviews the basics of NAND-based SSDs, introduces the ZNS SSD and its features, and discusses problems with the existing zoned interface.

### 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 gigabytes per second. It comprises four main architectural components (Figure 1): (1) a host interface logic that implements the protocol used to communicate with the host, such as SCSI [46] and recent NVMe [34]; (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; and (4) a multi-channel subsystem that connects NAND dies via a high-bandwidth interconnect. As shown in Figure 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, whereas the erase command is performed in units of blocks. NAND flash memory has three unique characteristics [1, 11, 13, 21, 31]: (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; and (3) limited lifetime (endurance)—each cell has a finite number of program/erase cycles [25].

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 [14, 23, 39, 60]: (1) dynamically mapping LBA (logical block addresses) to PPA (physical NAND pages addresses), (2) implementing a GC mechanism to handle the no in-place update issue and asynchronously reclaim invalid pages, and (3) applying a wear-leveling technique to evenly balance the usage (or aging property) of all blocks and prolong the SSD lifespan. However, the FTL brings in considerable overheads. First, the translation table requires a large amount of DRAM to store the mapping entries (e.g., 1 GB for 1TB NAND capacity for 4 KB 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 problem. Third, the FTL does not employ performance isolation mechanisms and incurs significant interference issues under mixed I/O profiles [33, 37].

## 2.2 ZNS SSDs

ZNS SSDs, a successor to OC SSDs [6, 9], have recently been developed to overcome the aforementioned limitations of conventional SSDs. There are several commodity ZNS SSDs from various vendors [40, 43, 44, 57]. A ZNS SSD applies the same architecture as a conventional one (see Figure 1) but exposes the ZNS 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. First, they 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 24 MB block size has at least 6,144× smaller table size compared to the traditional 4 KB page mapping [49]. Second, they eliminate the device-side GC and reclaim NAND blocks via explicit zone resets by host applications, which mitigates the WAF and log-on-log [59] issues and minimizes the over-provisioning overhead. Third, they 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 2. 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 a way that is either explicit (by OPEN management command) or implicit (by write I/O). 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). Note that *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

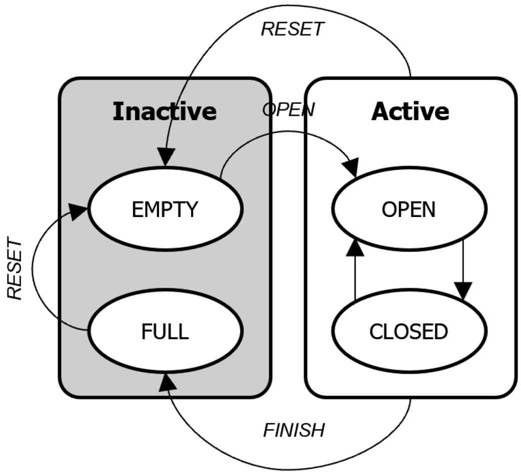


Fig. 2. The zone state diagram of ZNS SSD.

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 [53, 62]. 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 [61] 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, whereas 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 submit 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 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 [7] or file system [29, 30].

### 2.3 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 3 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

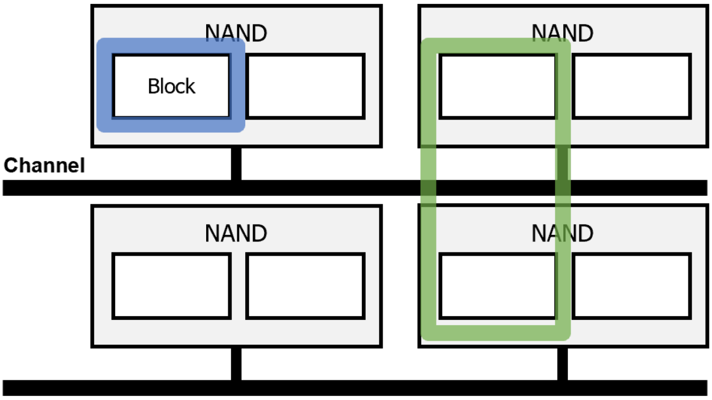


Fig. 3. The examples of physical zone placement in small (blue) and large (green) zone SSDs.

for application allocation (e.g., 14 zones [57]). 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. Additionally, 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 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 GC, especially while managing large numbers of small objects. Recent studies also corroborate some of these points. Specifically, Bae et al. [3] advocate a zone to be as small as possible to reduce the interference caused by high zone reclaiming latencies. ZNS+ [18] also prefers small zones, as it minimizes the latency of COPY operations performed frequently in its F2FS implementation.

## 2.4 Need for an Elastic Interface

The ZNS SSD brings in two key benefits. First, it exposes controllable GC 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 [3, 8, 18, 52] and our characterizations (Section 3) that follow 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 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

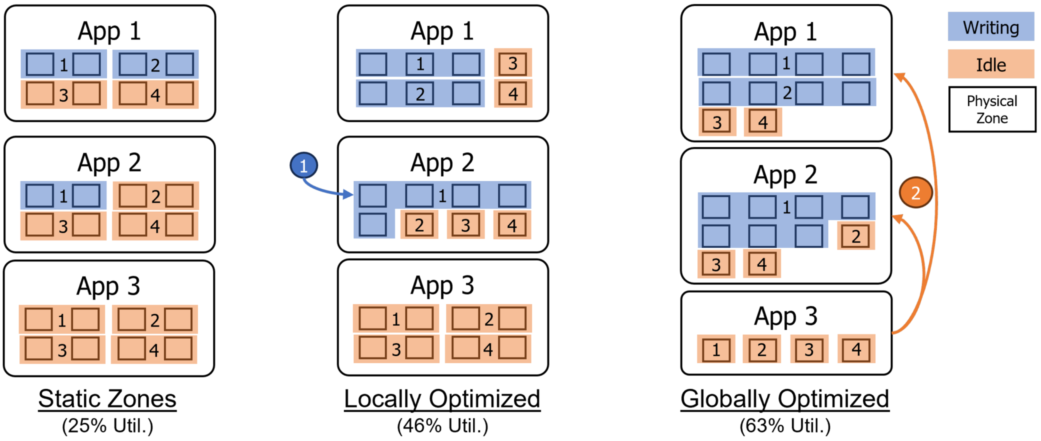


Fig. 4. An example of the static striping configuration and the optimizations.

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 sub-optimal 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. 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. 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 5, it is shown that the RocksDB over ZenFS [7] 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 [42] and F2FS [30] support ZNS SSDs but write user data to only one zone at a time, resulting in sub-optimal utilization of the available 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.

### 2.5 Lack of Performance Isolation in a 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

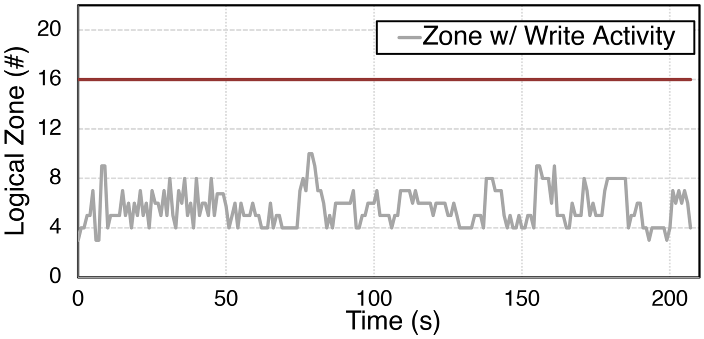


Fig. 5. The number of zone with actual write activity when running the *fill-random* workload over the RocksDB. The storage backend is ZenFS. The maximum number of active zones is 16 (red line).

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.

However, 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.

## 3 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.

### 3.1 Experimental Setup

***ZNS SSD and Testbed.*** We use a commodity ZNS SSD for characterization. Table 1 presents its hardware details. It has 40,704 physical zones, where each 96 MB-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 SSD

Table 1. Commodity ZNS SSD Specification

| Device HW Parameter | 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 |

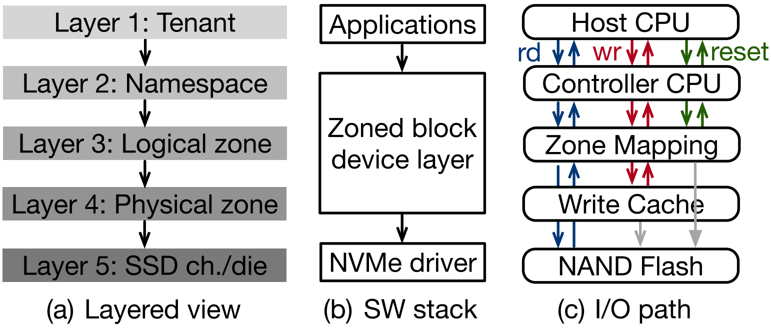


Fig. 6. System model, SW stack, and I/O path of a multi-tenant ZNS SSD deployment. The write cache flushes data to the NAND flash asynchronously. Zone resets are completed after invalidating the mapping layer, where NAND blocks are erased lazily. rd, read; wr, write.

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

***Workloads and Performance Metrics.*** We use the Fio benchmark tool [17] running on the SPDK framework [50] to generate synthetic workloads. We report both per-I/O 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.

### 3.2 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 6(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, whereas 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.

Table 2. Read I/O Average/P99.9 Latency and Bandwidth Varying the Stripe Size on a Physical Zone

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

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 layer (see Figure 6(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, and (3) scheduling a sequence of I/O commands to maximize device utilization and avoid head-of-line blocking. Figure 6(c) shows the I/O path of read/write/reset requests. We carefully configure each layer when designing characterization experiments.

## 3.3 Zone Striping

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

*3.3.1 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 2, the achieved per-die bandwidth increases slowly after the 16 KB stripe size. In terms of latency, the access time reduction is non-linear for sizes smaller than a NAND page (16 KB). When the I/O size is larger than 16 KB, 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 7, when issuing 2 MB 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. However, a wider width that fully uses the stripe size ($stripe\_size \times stripe\_width$) achieves higher bandwidth. For example, the 4 KB stripe size in 8-zones achieves 37.3% higher read bandwidth than the 8 KB 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.

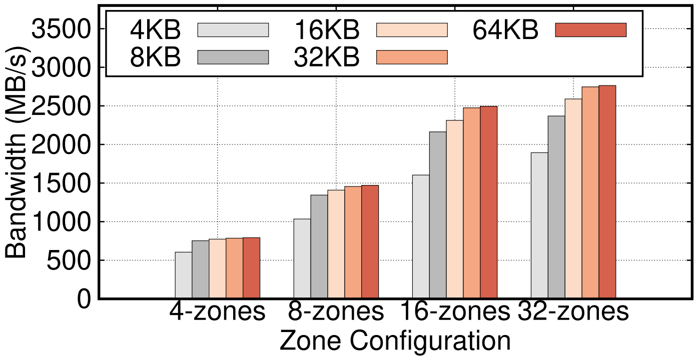


Fig. 7. Read bandwidth varying the stripe size for different types of zones.

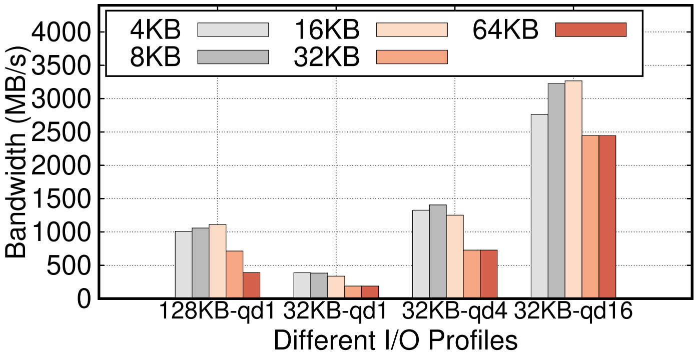


Fig. 8. Read bandwidth varying the stripe size under the stripe width of 16.

*3.3.2 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. However, too small a stripe also hurts the I/O efficiency of the device; a 4 KB stripe with an 8-zone or wider width significantly lags behind 8 KB or larger stripes in Figure 7. 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 8 presents the sustained read bandwidth varying the stripe size from 4 KB to 64 KB for four I/O profiles under the 16-zone configuration. For the 32 KB I/O with a queue depth of 16 (32 KB-QD16) and the 128 KB synchronous I/O (128 KB-QD1), the 16 KB stripe performs the best because NAND operates most efficiently when the stripe size aligns with the NAND page size. However, for the 32 KB with 1 and 4 queue depth cases (32 KB-QD1, 32 KB-QD4), 4 KB/8 KB stripes outperform the 16 KB 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, 32 KB and 64 KB, 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.

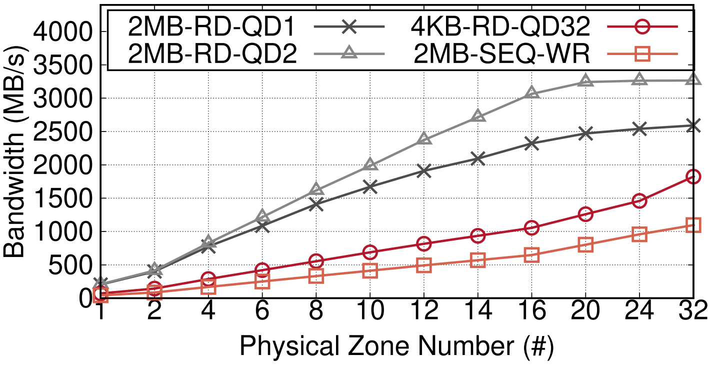


Fig. 9. Read/write bandwidth varying the number of physical zones.

***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 is 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.

## 3.4 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 program/erase cycles based on opaque wear-leveling policies.

*3.4.1 Basic Performance.* 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 16 KB 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 9). The performance of 2 MB reads with queue depths 1 and 2 (i.e., 2 MB-RD-QD1/2 MB-RD-QD2) keeps increasing until the number of physical zones approaches 20. But they max out for different reasons. The QD2 case is bounded by the PCIe bandwidth (i.e., four Gen3 lanes or 3.2 GB/s), whereas the QD1 scenario is simply limited by the application as it cannot issue enough outstanding I/Os at that queue depth. In terms of 4 KB random read with 32 queue depth and 2 MB sequential write, they sustain 80 MB/s read and 40 MB/s program bandwidth per physical die, respectively, requiring much more physical zones (~ 40 and 80) to utilize the channel or PCIe bandwidth fully.

*3.4.2 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:

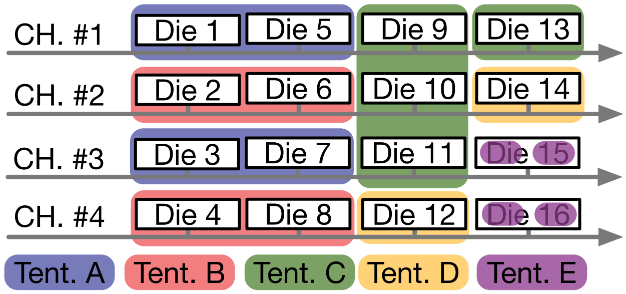


Fig. 10. Channel/die-overlapped zone placement. Assume that the ZNS SSD has 16 dies across four channels and each tenant has its own namespace. In case (1), the SSD serves two allocations from tenants A and B simultaneously (both asking for four physical zones). Due to the overlapped placement, both A and B benefit from two channels. Similarly, in case (2), the SSD serves four requests (which allocate three zones, one zone, one zone, and one zone) from tenants C and D asynchronously. Because of the overlapped placement, tenants C and D obtain three and two channels, respectively. In case (3), tenant E allocates four zones spanning across 2 dies, limited by the die bandwidth.

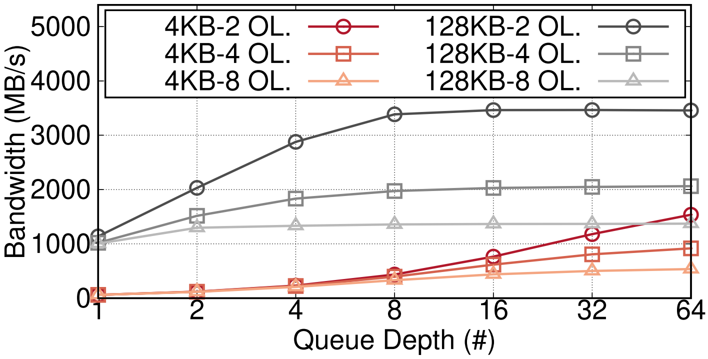


Fig. 11. Read bandwidth under three channel overlapping (OL) allocations.

- *Channel-overlapped placement:* As shown in cases (1) and (2) of Figure 10, 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 11 presents 4 KB and 128 KB 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 (i.e., 4-overlapped/8-overlapped) by 1.7×/2.9× and 1.7×/2.5× for 4 KB and 128 KB cases, respectively.
- *Die-overlapped placement:* Case (3) of Figure 10 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 12 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 1,128 MB/s and 2,051 MB/s along with 317us and 284us p99.9 tail latency for the 4 KB random read and 128 KB sequential read cases, respectively. With full overlap, we observe 47.2%/23.8% bandwidth drop and 87.1%/28.0%

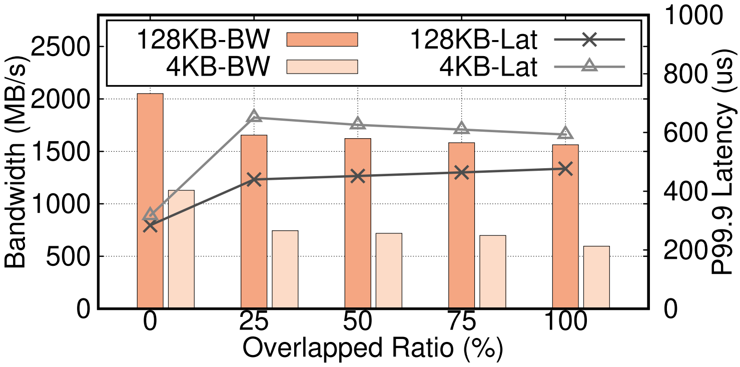


Fig. 12. Bandwidth and tail latency varying with the die overlapping ratio.

tail latency increase. Such performance degradation happens even when the overlapping ratio is lower than 25%, because both types of 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 [3]. 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, and (3) profiles its placement balanced level across different physical channels and dies.

### 3.5 I/O Execution under ZNS SSDs

A ZNS SSD eradicates background GC I/Os, thereby removing one form of performance nondeterminism. 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 with a conventional SSD because ZNS SSDs impose no physical resource partitions, and per die/channel bandwidth is narrow.

*3.5.1 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 channels and dies (i.e., 128-zone configuration with 16 KB stripe size) to match the conventional one. The fragmented conventional SSD is 70% filled and pre-conditioned with 128 KB random writes. Then we run eight read threads—where each issues one 128 KB read I/O to all dies uniformly random—and one write thread that performs sequential write at a fixed rate. Figure 13 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 1,000 MB/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 200 MB/s and 1,000 MB/s write bandwidth, respectively.

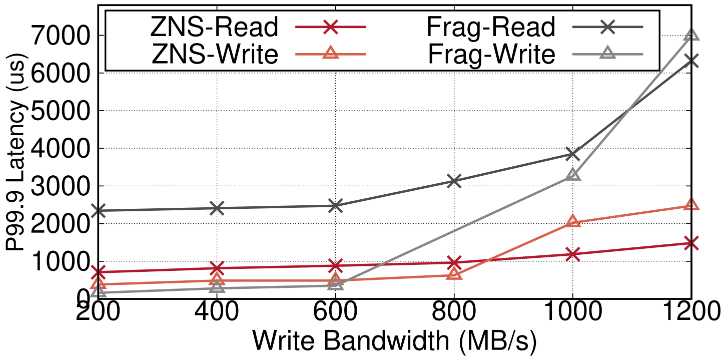


Fig. 13. Read tail latency varying the write bandwidth (ZNS vs. conventional SSD).

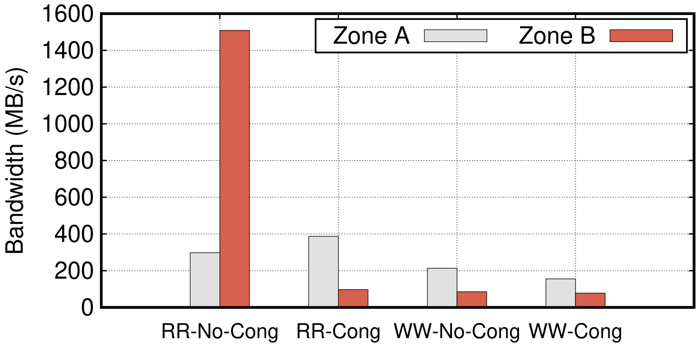


Fig. 14. Bandwidth under RD-RD and WR-WR congestion due to the die collision.

*3.5.2 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 [31] 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.

We set up two readers performing 128 KB 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 14 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 [26, 47]. We

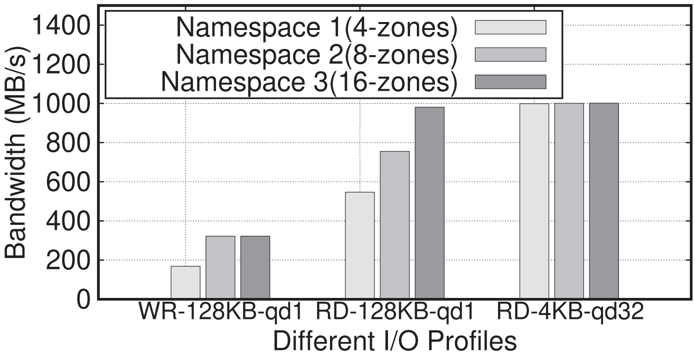


Fig. 15. Performance comparison of three namespaces (with different logical zones) running the same workload.

also demonstrate the write cache congestion in Figure 14. 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 (3.2 GB/s), and a single zone performs at ~213.3 MB/s. In this case, a physical zone in the logical zone receives write at a lower rate than the maximum bandwidth (~26.7 MB/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 toward the narrow zone are equally fetched by the device, but it soon consumes all available cache because of the scarce bandwidth (~85 MB/s) of underlying physical zones. It degrades others' bandwidth by 27.3% or 155 MB/s, and the device even fails to max out the PCIe bandwidth (~2.4 GB/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 channels and dies. Figure 15 shows the achieved bandwidth when these three namespaces run the same type of workload. In terms of the 128 KB 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 16 KB, a 128 KB write generates eight outstanding I/Os that can harness eight dies at best, 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 128 KB sequential reads. However, the three namespaces show the same bandwidth for a small 4 KB 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 16 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 4 KB (QD = 32) in the first two scenarios and 128 KB (QD = 32) in the third case.

*3.5.3 ZNS vs. Conventional SSDs.* There is no performance isolation support on a conventional SSD. Some recent proposals employ I/O determinism and NVM Sets [28, 31, 38] 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

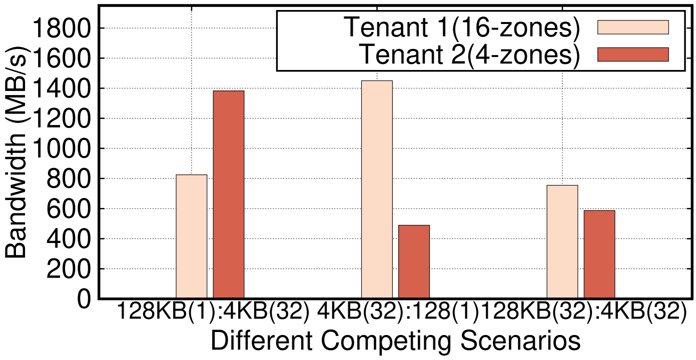


Fig. 16. Performance comparison of two tenants (with different logical zones) running different workloads.

redundant arrays [32, 58], 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 and (2) a per-zone I/O scheduler that orchestrates the read I/O submission based on the congestion level.

### 3.6 Zone Reset

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 3, its unloaded latency is around 204 us, whereas erasing a block takes around 3.5 ms [22]. Additionally, reset is a high-bandwidth operation, achieving 1,478 GB/s at max. This is reasonable as the per-die erase bandwidth of modern high-density NAND devices is higher than 10 GB/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 earlier and co-locate it with a 4 KB random read (QD = 32) or a 128 KB write (QD = 1), respectively. In terms of the read vs. reset interference scenario (Figure 17),

Table 3. Reset Average/P99.9 Latency and Bandwidth Varying the Number of Concurrent Reset Operations

| Reset # | Avg. Lat (us) | P99.9 Lat. (us) | B/W (GB/s) |
|---|---|---|---|
| 1 | 204 | 233 | 455 |
| 2 | 220 | 338 | 841 |
| 3 | 239 | 449 | 1,158 |
| 4 | 275 | 445 | 1,342 |
| 5 | 313 | 889 | 1,478 |
| 6 | 381 | 2,343 | 1,456 |

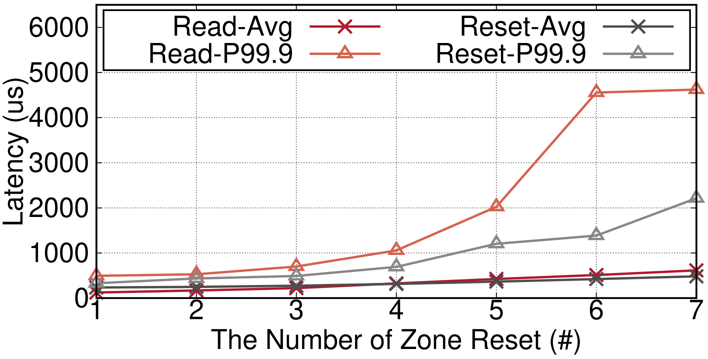


Fig. 17. Read vs. reset latency varying the number of reset operations.

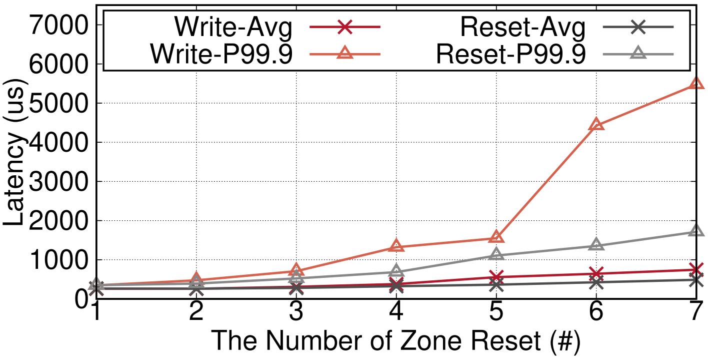


Fig. 18. Write vs. reset latency varying the number of reset operations.

the read average and p99.9 latencies rise by 5.0× and 9.4× when there are seven concurrent resets. This also causes a significant bandwidth degradation (from 1,492 MB/s to 202 MB/s). Regarding the write vs. reset (Figure 18), 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.

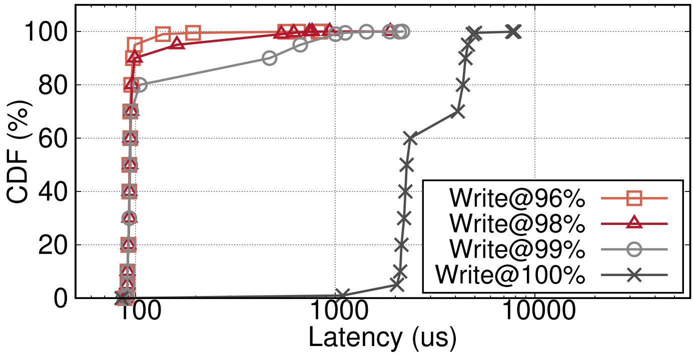


Fig. 19. CDF of write I/O latency for different traffic loads. The *x*-axis is log scale.

***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.

### 3.7 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, and (2) SSDs should follow strict timing rules [10] (e.g., the timing among upper/middle/lower 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 16 KB write I/Os at a rate lower than the maximum die serving bandwidth (see Table 1) and the last one maxes out. Figure 19 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 93 us). 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 2,278 us, which is even worse than the p99.9 latency of the first three scenarios (i.e., 562 us, 742 us, 1,434 us).

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 16 KB (QD = 1) write I/O with different manners: (1) an unloaded writer that submits at a fixed rate lower than the maximum die bandwidth

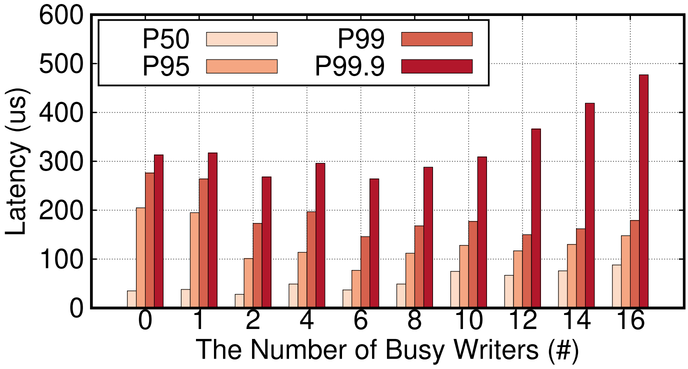


Fig. 20. Latency varying with the number of busy writers on a conventional SSD.

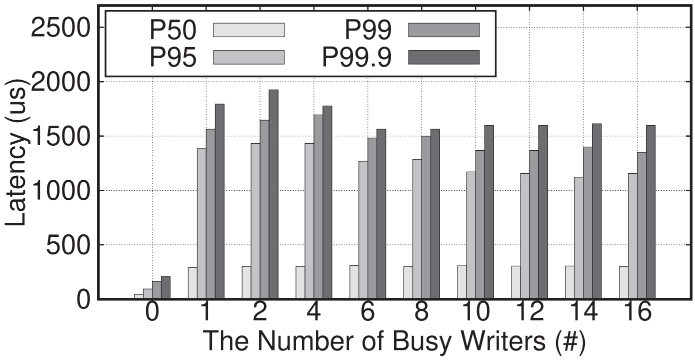


Fig. 21. Latency varying with the number of busy writers on a ZNS SSD.

(20 MB/s), and (2) a busy writer that runs without capping bandwidth. Then, we measure the 16 KB write I/O latency of an unloaded tenant as increasing the number of co-located busy writers. Figure 21 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 20. Since the cache is shared among all writes and reclaimed at the maximum bandwidth, we observe that p50/p95/p99 latencies stay quite stable, whereas the p99.9 latency only increases by 50.5%.

***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 EZNS: ENABLING AN ADAPTIVE ZNS

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.

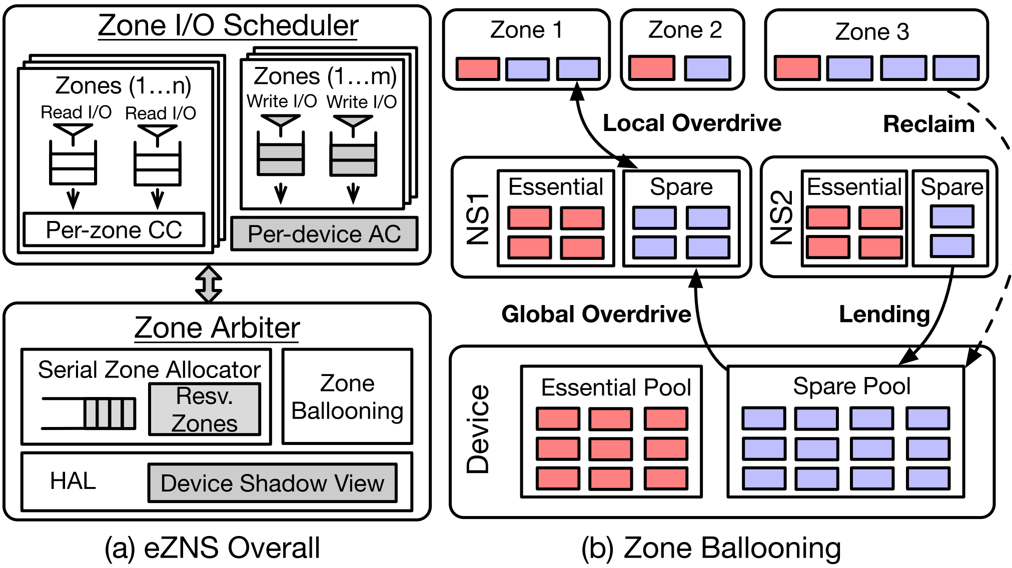


Fig. 22. eZNS system architecture.

### 4.1 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 22, 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 Section 3.

### 4.2 Hardware Contract and HAL

We develop eZNS based on the following hardware contract, which is 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); and (3) the zone allocation mechanism follows the wear-leveling requirements, indicating that consecutive allocated zones will not overlap on a physical die until all 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 does not 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 (Section 3.4). The second parameter required is the *NAND page size* used for striping configuration. For example, 16 KB 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 (Section 3.3), 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., 2 GB) regardless of the underlying device.

## 4.3 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; and (3) across stripe groups, die collision could happen for writes only if available active physical zones are fully populated across all dies. Given the preceding 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 (Section 3.4.2).

## 4.4 Zone Ballooning

The *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 23 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 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 [5, 45, 54], 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, and (2) return them when it finishes the stripe group either by

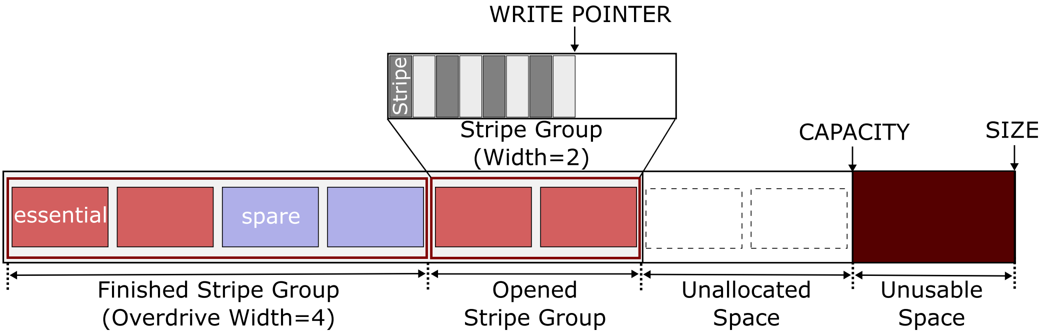


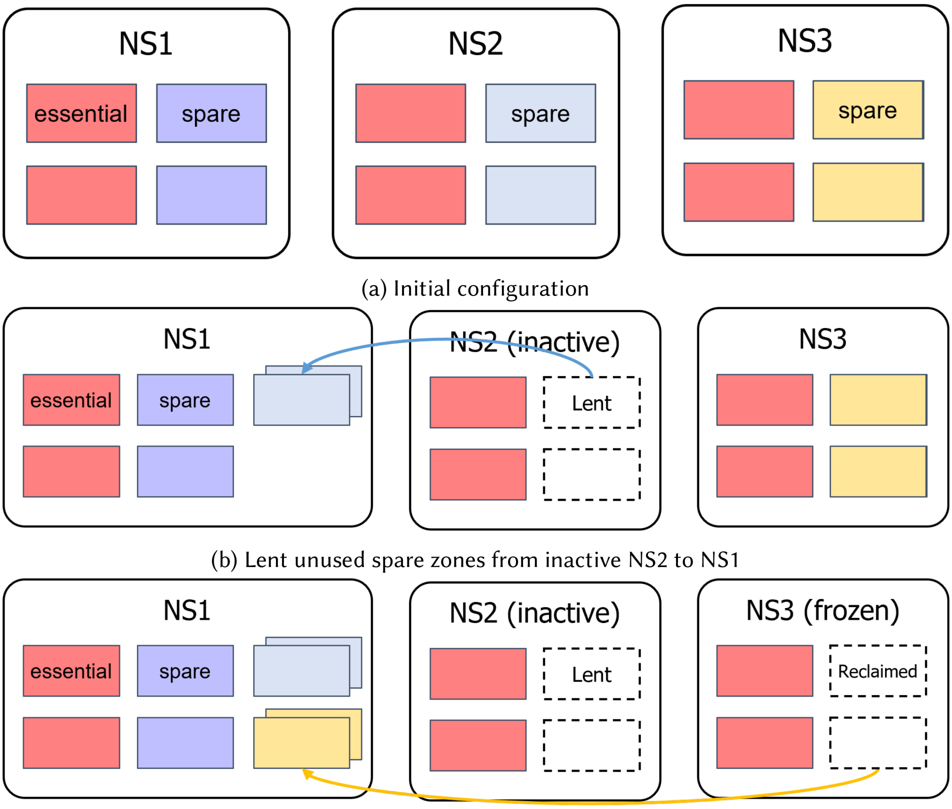
Fig. 23. Example of eZNS *v-zone* structure.

writing to the end of the stripe group or explicitly issuing FINISH/RESET commands from the application.

*4.4.1 Initial Resource Provisioning.* eZNS divides all 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 ($N_{essential}$), whereas 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.

*4.4.2 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-zones* ($N_{ActiveZoneHistory}$). 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 2 for efficient resource management. Note that the *local overdrive* operates in a serial and best-effort fashion. Last, eZNS sets the baseline stripe size to 32 KB 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 2× wider, the stripe size is reduced by half. We determine the range of stripe sizes to optimize the performance as aforementioned in Section 3.3. The reduced stripe size further contributes to the I/O scheduler ensuring fairness (Section 4.5).

*4.4.3 Global Overdrive: Namespace Expanding.* Across the whole device, our zone ballooning mechanism further reallocates spares across namespaces based on their latest write activity. We



(a) Initial configuration

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

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

Fig. 24. Three simple examples of the global overdrive mechanism.

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 24 illustrates an example case of *global overdrive*. Initially, in Figure 24(a), 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, 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 24(b). 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.

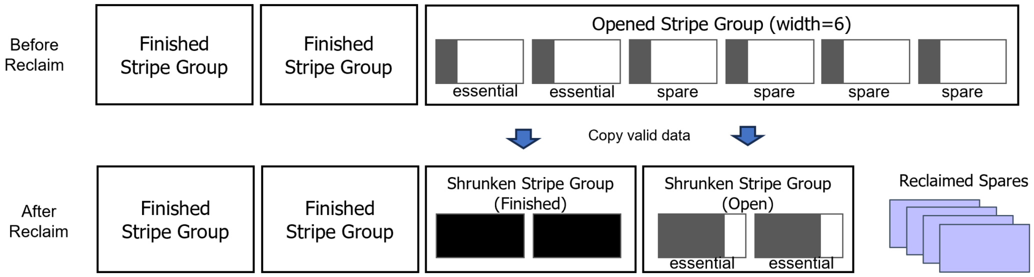


Fig. 25. Example of the zone reclaim operation. The original stripe group has width of 6 with four spares. The reclaim operation shrinks it to the minimum width of 2 using only essentials. When the amount of valid data exceeds the size of a minimum width stripe group, it finishes the stripe group and creates another one until it copies all valid data. Then it returns spares in the original stripe group to the pool.

*4.4.4 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—that is, 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 25 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.

## 4.5 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.

*4.5.1 Congestion-Avoid Read Scheduler.* Our design is based on the observations that (1) ZNS SSDs have no internal housekeeping operations and (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 first 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 [27] 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 1, 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 physical zones within a stripe group share the same congestion status. This 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.

*4.5.2 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 (Section 3.5). Therefore, eZNS monitors the global write latency and regulates writes using a token-based admission control scheme. We generate tokens periodically (Algorithm 1 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 (Algorithm 1 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 self-clocked in the congestion-less condition.

First, 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. Second, 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.

## 5 EVALUATION

We add a thin layer in the SPDK framework [50] to implement eZNS and realize the *v-zone* concept. The primary reason for choosing the SPDK approach was its ease of implementation

**ALGORITHM 1**: 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                                                        ▷ $\alpha$ = 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 10 ms
    $total\_lat = \sum_{active\_zone} per\_block\_lat$
    $total\_ios = \sum_{active\_zone} num\_ios$
    $avg\_lat(t) = \frac{total\_lat}{total\_ios}$
    $block\_admission\_rate = \frac{avg\_lat(t-1) + avg\_lat(t)}{2}$
procedure Write Token Generator()
    On every 1 ms
    for pending write zones do
        $token$ += $\frac{now - last\_refill}{block\_admission\_rate} \times stripe\_width$
```

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 Section 3.1. 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 [17] 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 pre-condition 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 [7] 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.

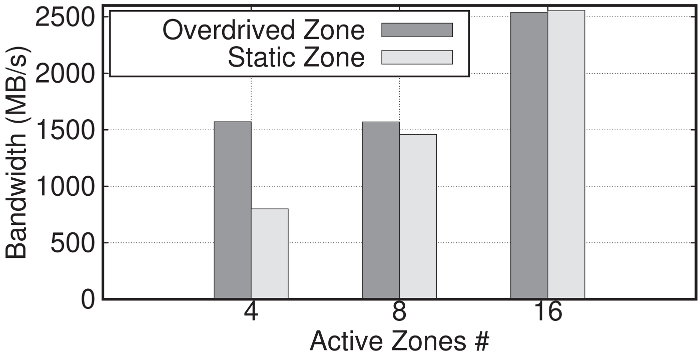


Fig. 26. B/W comparison between an overdrived and three statically configured zones.

***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 of 16 active zones, the minimum stripe width for *v-zone* is 2 with a stripe size of 32 KB. However, eZNS can overdrive the width up to 16 with a stripe size of 4 KB. For a fair comparison, we prepare a static logical zone configured with stripe width and size of 4 KB and 16 KB, 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 the YCSB benchmark (six namespaces), as specified in Section 5.3.

### 5.1 Zone Ballooning

We demonstrate the efficiency of zone ballooning when handling large writes (i.e., 512 KB I/O with a queue depth of 1). 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 26, 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.

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, whereas the fourth namespace (NS4) runs eight. NS1, NS2, and NS3 stop issuing writes at $t = 30$ seconds and resume the writing activity at $t = 80$ seconds. We measure the throughput and spare zone usage of four zones for a 100-second profiling window (Figures 27 and 28). 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 (~2.3 GB/s). It can then quickly release the harvest zones when other zones start issuing writes again.

### 5.2 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 29.

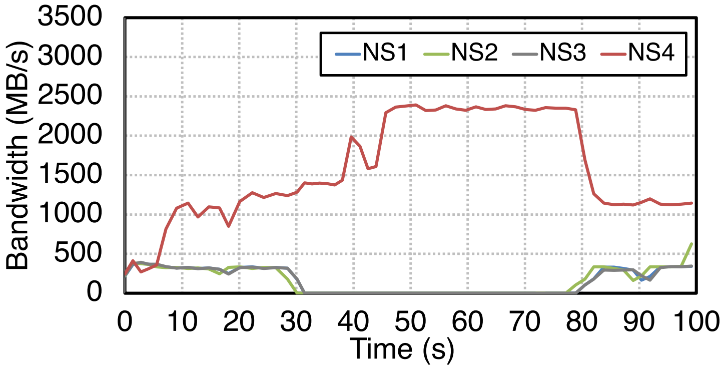


Fig. 27. Performance variation of four namespaces with global overdrive under 100 seconds.

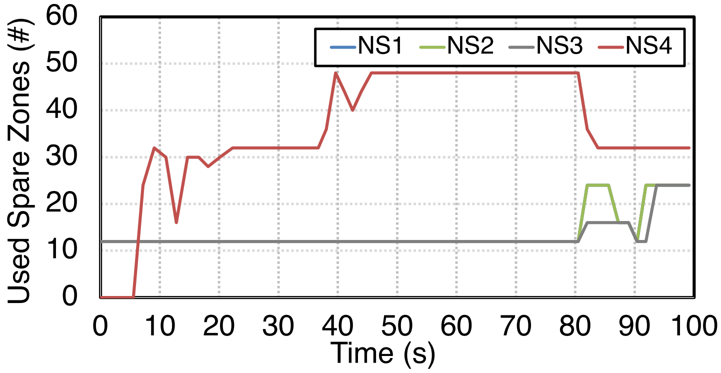
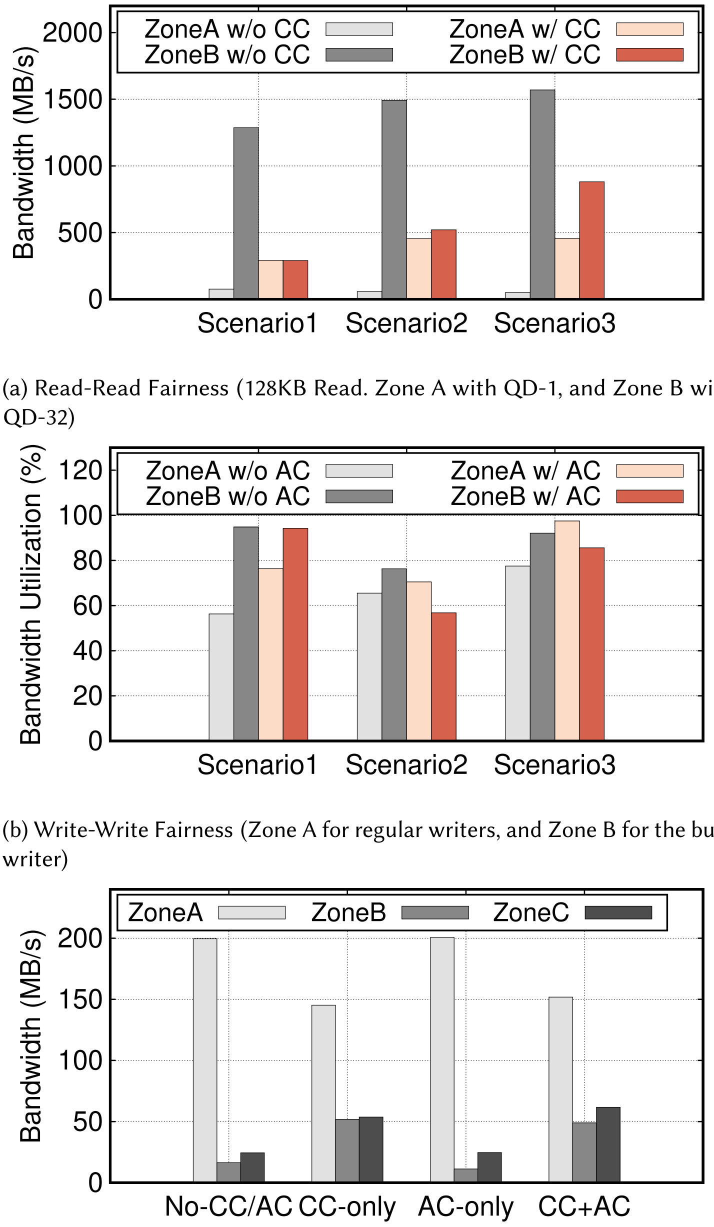


Fig. 28. The number of used spare zones of four namespaces under 100 seconds.

***Read-Read Fairness.*** We run a sequential read of 128 KB 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, whereas a zone with a width 8 has only one thread. As shown in Figure 29(a), in scenario 1, when disabling our congestion control mechanism, Zone A (configured with stripe width 2 and stripe size 32 KB, QD-1) and Zone B (configured with stripe width 8 and stripe size 8 KB, QD-32), even holding the same sized full stripe, achieve 76 MB/s and 1,287 MB/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 290 MB/s. In scenarios 2 and 3, we change the Zone A stripe configuration to <stripe width 4, stripe size 16 KB, QD-1> and <stripe width 8, stripe size 8 KB, 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 128 KB 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 512 KB size. In the first scenario, we co-locate 16 regular write zones (Zone A, where each has a striping width of 8 with 8 KB stripe size and submits write I/Os at 5 ms intervals, sustaining 95 MB/s maximum throughput) with a busy writer (Zone B, which has width 2 and 32 KB stripe size, submits I/O without interval



(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)

Fig. 29. Efficiency of eZNS on handling read-read, write-write, and read-write congestion. CC, congestion control; AC, admission control.

delays, achieving 85 MB/s at most). Figure 29(b) reports the bandwidth utilization of 1 regular 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, whereas Zone A receives only 16.4 MB/s per physical zone or 38.4% of the physical zone bandwidth. As described in Section 4.5.2, 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 2,160.9 MB/s to 2,304.3 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 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 2,403.3 MB/s to 2,997.7 MB/s).

***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 32 KB stripe size, performing 128 KB random read at queue depth 32, across all physical dies; (2) 1 busy writer (Zone B), whose striping width is 2 with 32 KB stripe size; and (3) ×16 regular writers (Zone C), which has a striping width of 8 and 32 KB stripe size each, submitting I/Os under a 5 ms interval. Both B and C issue 512 KB large writes. Figure 29(c) reports their per-thread bandwidth. When disabling our scheduler, each reader achieves 199.6 MB/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.

### 5.3 Application: RocksDB

To evaluate eZNS in a real-world scenario, we use RocksDB [41] over the ZenFS storage backend. In addition to the built-in utility in the RocksDB *db_bench* tool, we port YCSB workload generators [4] 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 30 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 16 KB for the width of 8 to avoid the namespace running only on large stripe sizes. We compare the

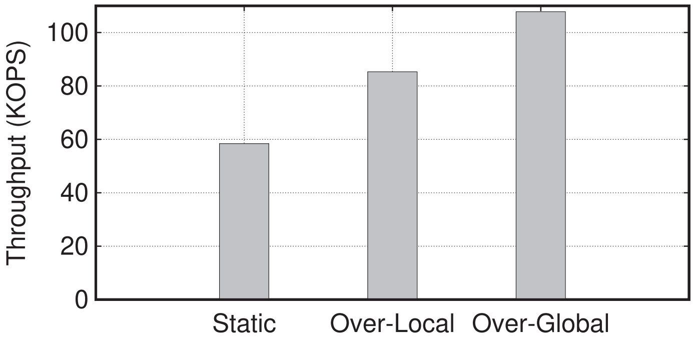


Fig. 30. Throughput of *overwrite* workload on a single namespace without other tenants. Ove, overdrive.

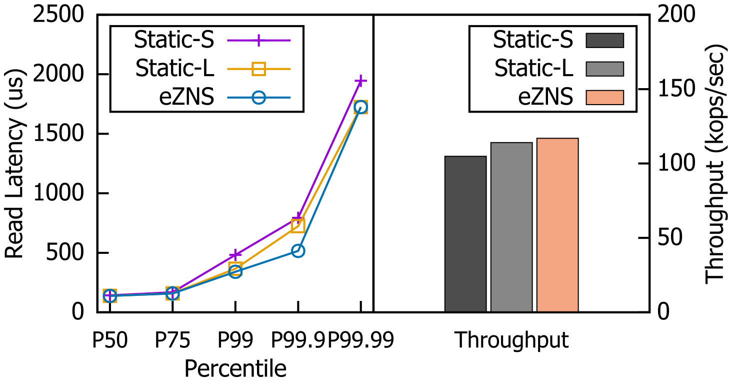


Fig. 31. The *readwhilewriting* workload on single-tenant configurations. Static has stripe width of 16. S, 4 KB stripe; L, 16 KB stripe.

performance of eZNS over two static configurations, both with a stripe width of 16 but with different stripe sizes of 4 KB and 16 KB. 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 31 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 16 KB and 4 KB, respectively. Additionally, eZNS also improves the throughput by 11.5% and 2.5% with a stripe size of 4 KB and 16 KB.

***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, whereas C and D execute *randomread* concurrently. We run the benchmark for 1,800 seconds and report the latency and the throughput. Figure 32 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) (Figure 33).

***Mixed YCSB Workloads with Four Namespaces.*** YCSB [15] 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

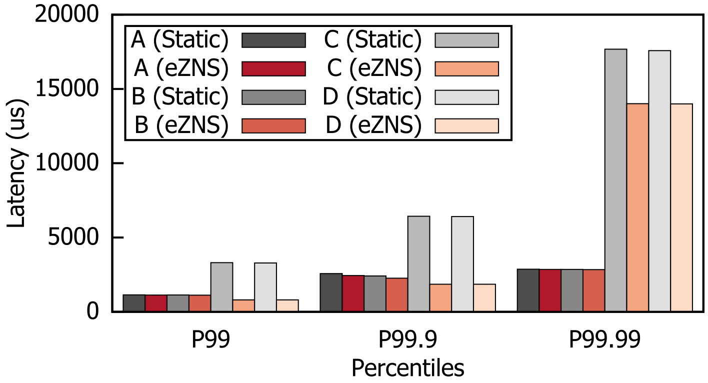


Fig. 32. Latency of db_bench workloads (two overwrite, two randomread) on different namespaces over eZNS and static zone.

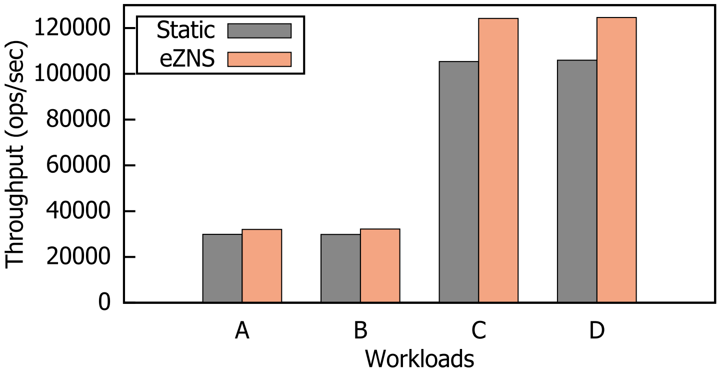


Fig. 33. Throughput of db_bench workloads (two overwrite, two randomread) on different namespaces over eZNS and static zone.

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 34 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 35.

***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 four physical zones are designated as part of the global spare pool.

Figures 36 and 37 present the tail latencies and throughput comparisons between eZNS and the static zone configuration. For YCSB A through 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%, whereas 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

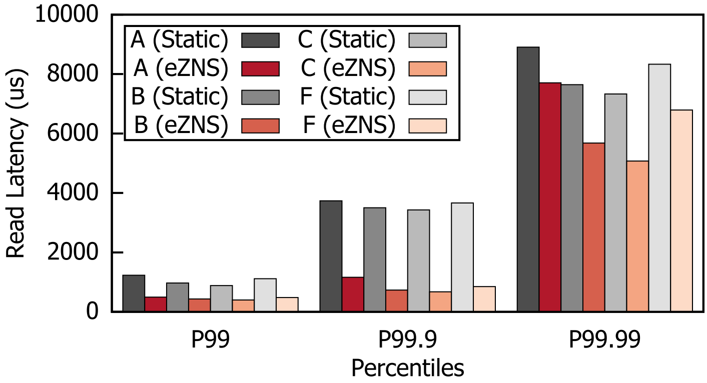


Fig. 34. Read latency of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone.

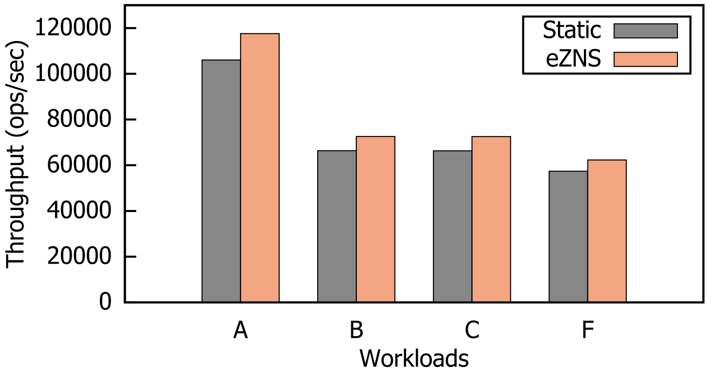


Fig. 35. Throughput of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone.

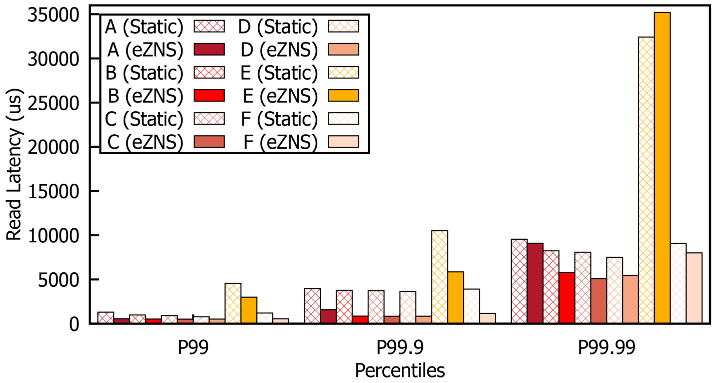


Fig. 36. Read latency of YCSB workloads (A/B/C/D/E/F) on different namespaces over eZNS and static zone.

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

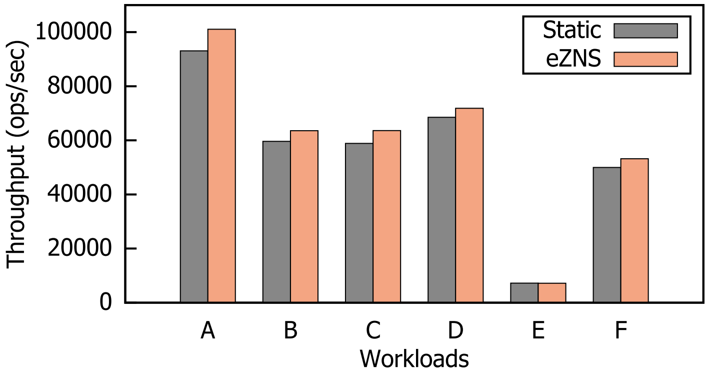


Fig. 37. Throughput of YCSB workloads (A/B/C/D/E/F) on different namespaces over eZNS and static zone.

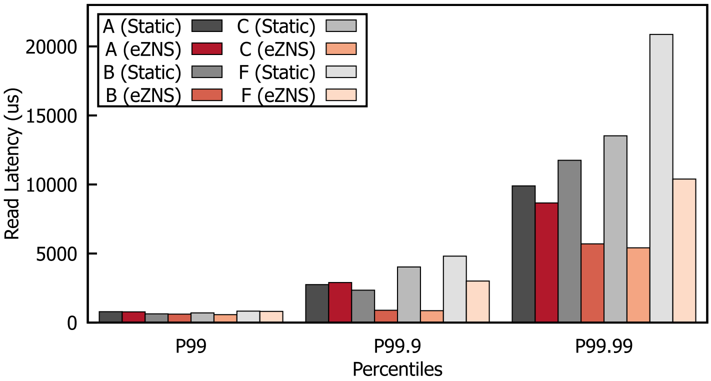


Fig. 38. Read latency of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone running on F2FS.

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 a 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 [30] 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 38 and 39.

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 4 KB, eZNS cannot increase the striping width beyond 8. Consequently, the global overdrive mechanism does not operate effectively in this scenario, forcing eZNS to make

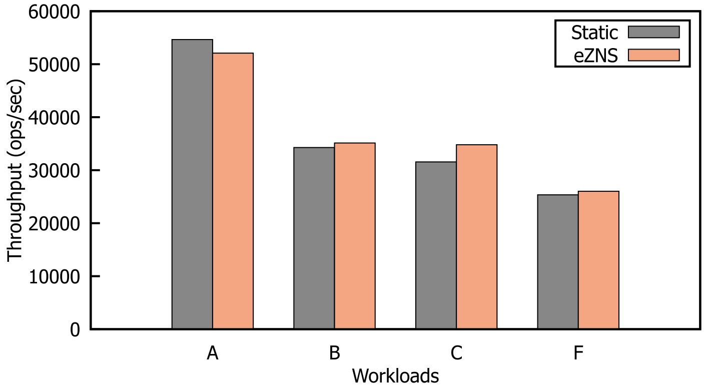


Fig. 39. Throughput of YCSB workloads (A/B/C/F) on different namespaces over eZNS and static zone running on F2FS.

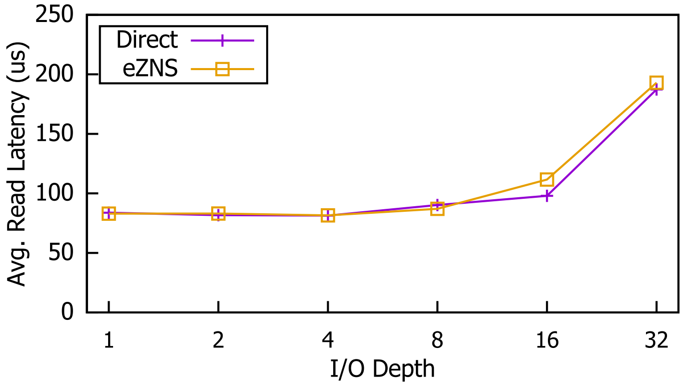


Fig. 40. Comparison of average read latency for 4 KB I/Os at various depths between the host-managed zone access and eZNS.

a tradeoff, sacrificing a small amount of throughput from YCSB A to ensure a fair distribution of read bandwidth across all namespaces.

### 5.4 Overhead Analysis

***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 4 KB random read I/Os and compared it with host-managed zone access, where the host directly accesses the physical device without eZNS. Figure 40 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 1 KB, 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 2 MB and 2.5 MB of memory, respectively. Compared to the memory requirements of the page mapping in conventional SSDs, the memory usage of eZNS is negligible.

## 6 RELATED WORK

***Early ZNS Exploration.*** Researchers have made initial efforts to understand the ZNS interface and integrate it into the host storage stack. Stavrinos et al. [51] 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). Shin et al. [48] develop a performance analysis tool for a ZNS SSD and profile its parallelism, isolation, and predictability properties. Compared with our study, they did not investigate the underlying device's internal mechanisms when realizing the ZNS interface and thereby are unable to correlate the observed performance with the ZNS SSD characteristics. ZNS+ [18] 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. Bae et al. [3] prioritize I/O requests for less-congested zones using an interference map, whereas 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 does not 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. Im et al. [20] 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 research [2, 12, 19, 36] 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 [56] that group blocks together. It benefits from high parallelism that transforms writes into sequential NAND programming. However, multi-tenancy workloads cause interference and a high WAF. ZNS SSDs eliminate GC and fix the WAF to 1 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 [24]. 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 [31] 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 **Endurance 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.

***OC 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 [35] 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 [55] 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 [32] 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, such as wear leveling and physical zone-to-die mapping. Another challenge arises when the system consists of heterogeneous devices resulting in the overhead of managing different hardware 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, and queue length on a die. 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 OC SSD.

***ZNS SSD Architecture.*** We believe that the ZNS interface is an appropriate compromise between hardware and software ends. However, as one might expect, there certainly are hardware 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 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 and F2FS.*** BtrFS [42] and F2FS [30] 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.

## 7 CONCLUSION

This article presented an in-depth study on understanding the characteristics of a commodity ZNS SSD. Then, we proposed 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 demonstrated significant performance and fairness improvements using eZNS over various scenarios.

## REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX 2008 Annual Technical Conference.*

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference*, Vol. 57. Boston, USA.

[3] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. 2022. 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.*

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing latency spikes in log-structured merge key-value stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. 753–766.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*.

[6] Matias Bjørling. 2019. From open-channel SSDs to zoned namespaces. In *Proceedings of the Linux Storage and Filesystems Conference (Vault'19)*, Vol. 1.

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

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

[9] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux open-channel SSD subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 359–374.

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

[11] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal parallelism of flash memory-based solid-state drives. *ACM Transactions on Storage* 12 3, (May 2016), Article 13, 39 pages.

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

[13] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*.

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

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 143–154.

[16] NVM Express. 2019. NVM Express Base Specification, Revision 1.4. Retrieved April 20, 2024 from https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf

[17] GitHub. 2022. Flexible I/O Tester (FIO). Retrieved April 20, 2024 from https://github.com/axboe/fio

[18] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*.

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

[20] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. 2022. 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)*. ACM, New York, NY, USA, 15–21. DOI: https://doi.org/10.1145/3564695.3564774

[21] Myoungsoo Jung and Mahmut Kandemir. 2013. 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*.

[22] 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. 2019. 13.4: A 512Gb 3-bit/cell 3D 6th-generation V-NAND flash memory with 82 MB/s write throughput and 1.2 Gb/s interface. In *Proceedings of the 2019 IEEE International Solid-State Circuits Conference-(ISSCC'19)*. IEEE, 216–218.

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

[24] 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. 2021. 30.2: A 1Tb 4b/cell 144-tier floating-gate 3D-NAND flash memory with 40 MB/s program throughput and 13.8 Gb/mm 2 bit density. In *Proceedings of the 2021 IEEE International Solid-State Circuits Conference (ISSCC'21)*, Vol. 64. IEEE, 424–426.

[25] 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. 2022. A 1Tb 3b/cell 8th-generation 3D-NAND flash memory with 164 MB/s write throughput and a 2.4 Gb/s interface. In *Proceedings of the 2022 IEEE International Solid- State Circuits Conference (ISSCC'22)*, Vol. 65. 136–137. DOI: https://doi.org/10.1109/ISSCC42614.2022.9731640

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

[27] 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. 2020. 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*.

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

[29] Damien Le Moal and Ting Yao. 2020. zonefs: Mapping POSIX file system interface to raw zoned block device accesses. In *Proceedings of the Linux Storage and Filesystems Conference (Vault'20)*.

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

[31] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. 2021. 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*.

[32] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. 2022. RAIL: Predictable, low tail latency for NVMe flash. *ACM Transactions on Storage* 18, 1 (2022), Article 5, 21 pages.

[33] 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 2021 Conference (SIGCOMM'21)*. ACM, New York, NY, USA, 106–122. DOI: https://doi.org/10.1145/3452296.3472940

[34] NVM Express. 2022. The NVMe Base Specification. Retrieved April 20, 2024 from https://nvmexpress.org/developers/nvme-specification/

[35] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. 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*.

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

[37] Stan Park and Kai Shen. 2012. FIOS: A fair, efficient flash I/O scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*.

[38] Chris Petersen, Wei Zhang, and Alexei Naberezhnov. 2018. Enabling NVMe I/O determinism @ scale. In *Proceedings of the 12th Annual Flash Memory Summit*.

[39] Roman Pletka, Ioannis Koltsidas, Nikolas Ioannou, Saša Tomić, Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Aaron Fry, and Tim Fisher. 2018. Management of next-generation NAND flash to achieve enterprise-level endurance and latency targets. *ACM Transactions on Storage* 14, 4 (2018), 1–25.

[40] Radian Memory Systems. 2022. Radian Memory System RMS ZNS SSDs. Retrieved April 20, 2024 from https://www.radianmemory.com/zoned_namespaces/

[41] RocksDB. 2022. RocksDB Home Page. Retrieved April 20, 2024 from http://rocksdb.org/

[42] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage* 9, 3 (2013), 1–32.

[43] Samsung. 2022. Samsung PM1731a ZNS SSDs. Retrieved April 20, 2024 from https://news.samsung.com/global/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan

[44] Samsung. 2022. Samsung PM1731a Review from STH. Retrieved April 20, 2024 from https://www.servethehome.com/samsung-pm1731a-ssd-with-zns-support/

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

[46] Wikipedia. 2022. The SCSI Protocol. https://en.wikipedia.org/wiki/SCSI#cite_note-1

[47] Kai Shen and Stan Park. 2013. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. 67–78.

[48] Hojin Shin, Myounghoon Oh, Gunhee Choi, and Jongmoo Choi. 2020. Exploring performance characteristics of ZNS SSDs: Observation and implication. In *Proceedings of the 2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA'20)*.

[49] Chang Siau, Kwang-Ho Kim, Seungpil Lee, Katsuaki Isobe, Noboru Shibata, Kapil Verma, Takuya Ariki, Jason Li, Jong Yuh, Anirudh Amarnath, et al. 2019. 13.5: A 512Gb 3-bit/cell 3D flash memory on 128-wordline-layer with 132 MB/s write performance featuring circuit-under-array technology. In *Proceedings of the 2019 IEEE International Solid-State Circuits Conference (ISSCC'19)*. IEEE, 218–220.

[50] SPDK. 2022. The Storage Performance Development Kit (SPDK). Retrieved April 20, 2024 from https://spdk.io

[51] Theano Stavrinos, Daniel S. Berger, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Don't be a blockhead: Zoned namespaces make work on conventional SSDs obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems*.

[52] Nick Tehrany and Animesh Trivedi. 2022. Understanding NVMe Zoned Namespace (ZNS) flash SSD storage devices. arXiv:2206.01547 (2022).

[53] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. 2011. Understanding the impact of power loss on flash memory. In *Proceedings of the 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC'11)*. IEEE, 35–40.

[54] Carl A. Waldspurger. 2003. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review* 36, SI (Dec. 2003), 181–194.

[55] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the 9th European Conference on Computer Systems*.

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

[57] Western Digital. 2022. Western Digital Ultrastar ZNS SSDs. Retrieved April 20, 2024 from https://www.westerndigital.com/solutions/zns

[58] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*.

[59] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't stack your log on my log. In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*.

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

[61] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.

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

Received 2 November 2023; revised 7 March 2024; accepted 13 March 2024