FlexZNS: Building High-Performance ZNS SSDs with Size-Flexible and Parity-Protected Zones

Yu Wang[1], You Zhou[2*], Zhonghai Lu[3], Xiaoyi Zhang[4],
Kun Wang[4], Feng Zhu[4], Shu Li[4], Changsheng Xie[1], and Fei Wu[1*]
[1]*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China*
[2]*School of Computer and Technology, Huazhong University of Science and Technology, Wuhan, China*
[3]*School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, Stockholm, Sweden*
[4]*Alibaba Group, Hangzhou, China*
{ywang_wnlo, zhouyou2, cs_xie, wufei}@hust.edu.cn, zhonghai@kth.se,
{shijing.zxy, andrew.wk, f.zhu, s.li}@alibaba-inc.com

***Abstract*—NVMe zoned namespace (ZNS) SSDs present a new class of storage devices with attractive features including low cost, software definability, and stable performance. However, one primary culprit that hinders the adoption of ZNS is the high garbage collection (GC) overhead it brings to host software. The ZNS interface divides the logical address space into size-fixed zones that must be written sequentially. Despite being friendly to flash memory, ZNS requires host software to perform out-of-place updates and GC on individual zones. Current ZNS SSDs typically employ a large zone size (e.g., of GBs) to be conducive to die-level RAID protection on flash memory. This impedes flexible data placement, such as mixing data with different lifetimes in the same zone, and incurs sizable data migrations during zone GC. To address this problem, we propose FlexZNS, a novel ZNS SSD design that provides reliable zoned storage allowing host software to configure the zone size flexibly as well as multiple zone sizes. The size variability of zones poses two interrelated challenges, one for the SSD controller to establish per-zone RAID protection, and the other for host software to manage variable zone capacity loss caused by parity storage. To tackle the challenges, FlexZNS decouples the storage of parity from individual zones on flash memory and hides the zone capacity loss from the host software. We verify FlexZNS on a ZNS-compatible file system F2FS and a popular key-value store RocksDB. Extensive experiments demonstrate that FlexZNS can significantly improve the system performance and reduce GC-induced write amplification, compared with a conventional ZNS SSD with large-sized zones.**

***Index Terms*—Zoned Namespace, Flash Memory, Solid-State Drives, Garbage Collection**

I. Introduction

NAND flash-based *solid-state drives (SSDs)* have been replacing traditional *hard disk drives (HDDs)* and gained increasingly used in modern storage systems [1]. This transformation leads to a fundamental rethink of the storage interface. In the past decades, the block interface has been the standard for storage devices to interact with the host. It presents the storage space as a linear array of fixed-size logical pages that can be accessed randomly. This easy-to-use abstraction is efficient for HDDs as they support random access.

However, flash memory has unique features, for example, pages in a flash block can only be written in a sequential manner after the block is erased. The mismatch between the block interface abstraction and flash access features brings in high taxes on SSDs, including high cost, unpredictable performance, and limited lifetime [2]. To manifest as a standard block device, SSDs internally employ a *flash translation layer (FTL)* to access flash memory respecting its features. The FTL allows random access to *logical page numbers (LPNs)* but performs log-structured writes and out-of-place updates on flash memory. A mapping table is maintained to translate LPNs to flash *physical page numbers (PPNs)*, demanding expensive DRAM (1GB per 1TB storage capacity) for caching the table.

Meanwhile, periodical *garbage collection (GC)* is required to reclaim obsolete, invalid flash pages in the unit of blocks, where valid pages are migrated and then the blocks are erased to be free. SSD GC activities are invisible to and uncontrollable by the host and could degrade system performance significantly and unexpectedly. GC also increases the *write amplification (WA)*, which becomes an increasingly serious concern since high-density flash memory bears poor write endurance/lifetime [3]. Moreover, from a holistic system view, a log-on-log problem emerges and magnifies write pressure on SSDs [4], as log-structured applications and file systems are popular and stack additional log layers over the FTL.

To address the concerns about SSD cost, performance, and lifetime, the NVMe *zoned namespace (ZNS)* interface has recently been introduced [5]. It abstracts the storage space as an array of fixed-size zones, which must be written sequentially and then reset before re-write by the host. This write constraint is consistent with the sequential-write-after-erase feature of flash memory, leading to several advantages.

On the SSD side, GC activities and subsequent performance fluctuation can be eliminated by aligning a logical zone with a *flash block group (FBG)*. A zone reset would invalidate the whole FBG and no data migrations are required to erase the FBG. Also, the hardware cost decreases due to a minimum DRAM requirement for caching the mapping table and zero

This work was supported in part by the National Natural Science Foundation of China under Grants 62372197, U2001203, U22A2071, and 62102155, in part by the Fundamental Research Funds for the Central Universities, HUST under Grant 2021XXJS108, and in part by the Alibaba Group through Alibaba Innovative Research Program.

* Corresponding authors: You Zhou and Fei Wu.

over-provisioning flash space for improving GC efficiency. Modern SSDs are architected with tens of parallel flash dies. In general, they employ an FBG as the unit of space allocation and GC. To exploit parallelism and enhance reliability, each FBG contains flash blocks with the same offset across many dies and constitutes a die-level RAID stripe with parity protection [3], [6]. As current ZNS SSD products usually map a zone to such an FBG, the zone size reaches GBs in size, e.g., 2GB in the Western Digital ZN540 SSD [2] and 1.44GB in the Inspur NS8600 SSD [7]. The FTL only needs to maintain a zone-level mapping table, whose size is a few KBs per 1TB storage capacity.

From a host-side perspective, the ZNS interface shifts the GC responsibility from the SSD FTL to the host software [8]. A zone is the unit of free space allocation and GC. Host software is able to control data placement over zones, where rich host-side semantics can be exploited to reduce zone GC overhead, and also schedule zone GC activities to avoid performance interference. Recent research has demonstrated such software-definable capability can significantly improve storage performance and WA/lifetime in the application scenario of RocksDB (a popular key-value store) [2], [7], [9], [10].

Although ZNS SSDs promise the above advantages, a main culprit that overshadows their full potential is the high zone GC overhead originating from the fixed and large zone size. To reduce the GC overhead, it has become common practice to store data with different lifetimes (or write hotness) in separate GC units. However, a large zone size would impede accurate data separation and cause a large number of valid data migrations during zone GC. We have conducted experiments to verify that this problem leads to significant performance degradation and WA increase, as detailed in Section II.

In addition, a large zone size has two other drawbacks. First, a ZNS SSD can only open a limited number of zones for writing data simultaneously, since the controller must allocate some hardware resources to maintain open (or partially-written) flash blocks [2], [11]. A larger zone size indicates a smaller number of zones that can be opened at the same time. This results in degraded write concurrency and multi-tenant sharing capability (each tenant requires an independent set of open zones). Second, in a multi-tenant scenario, using small zones facilitates performance isolation between tenants [11]. As a small zone is mapped to an FBG spanning a small number of flash dies, zones of different tenants can be placed in separate sets of flash dies for hardware isolation.

To address the drawbacks of ZNS SSDs with a fixed (and usually large) zone size, in this paper, we advocate that the current NVMe ZNS protocol should be extended to support size-configurable zones. This would provide a better software-definable capability and be essential to deliver the benefits of ZNS storage to a wider range of application scenarios. For example, an application can choose small zones to store a small set of write-hot data to reduce GC overhead, while large zones can be used for write-cold data; in some common scenarios where multiple applications share the same ZNS SSD, they can configure different sizes for their respective zones according to their own data access patterns.

However, the practical implementation of this idea is nevertheless challenging because it requires the systematic consideration of several trade-offs between performance, reliability, cost, and storage management complexity. To tackle the trade-offs, we propose *FlexZNS*, an efficient ZNS SSD design that allows applications to manage the storage space flexibly as one or multiple sets of zones, each set with a different size configuration. For storage management simplicity, FlexZNS supports a limited number of available zone sizes aligned with the power of two, such as 512MB, 1GB, and 2GB.

Die-level RAID protection is indispensable to guarantee SSD reliability, such as achieving a bit error rate below 10^{-17} and protecting data against die failures [3], [6]. In conventional block-interfaced or ZNS SSDs, each zone or FBG contains flash blocks across a fixed number of (say *N*) dies and stores one parity chunk along with *N-1* user data chunks. The NVMe ZNS protocol defines a concept named *zone capacity*, which refers to the user-writable capacity of a zone and can be smaller than the *zone size*. A zone can have an easy-to-manage size aligned with the power of two but an abnormal user-writable capacity smaller than the zone size (so as to adapt to the parity storage overhead and possible irregular flash block size). Unlike traditional ZNS SSDs that have fixed-size zones and deterministic parity-related capacity loss, FlexZNS allows applications to configure variable zone sizes. This would present a dilemma for applications to manage storage space, since the user-writable capacities of zones and the whole SSD change over different zone size configurations (as shown in Table I in Section II).

To solve this problem, FlexZNS aligns the user-writable capacity of each zone with its nominal size and offloads the storage management complexity into the SSD. From the view of applications, the available storage capacity is exactly the same as the sum of the sizes of zones allocated to them. Specifically, FlexZNS decouples the storage of parity from individual zones and divides flash storage into a data area and a parity area. We refer to a *superblock* as a group of flash blocks with the same offset across all the dies. Zones of different sizes are mapped to FBGs in separate sets of superblocks in the data area. The parity of each zone is stored in the parity superblocks and in a die non-overlapped with the zone's FBG.

We implement FlexZNS on a popular SSD emulator FEMU [12] and evaluate it in two ZNS-compatible application scenarios, F2FS [13] and RocksDB, from several aspects. First, experimental results show that compared to a conventional ZNS SSD with large zones sized in 2GB, FlexZNS with small zones sized in 512MB has high GC efficiency and thus improves system performance and WA by an average of 27.2% and 51.7%, up to 1.54× and 2.63×, respectively.

Second, there exists a trade-off between the performance and storage cost when the zone size can be configured flexibly. Using small zones can reduce zone GC overhead and improve performance. However, the parity storage cost increases, e.g., from 1.56% of SSD capacity with a 2GB zone size to 6.25% of SSD capacity with a 512MB zone size. We exploit this trade-

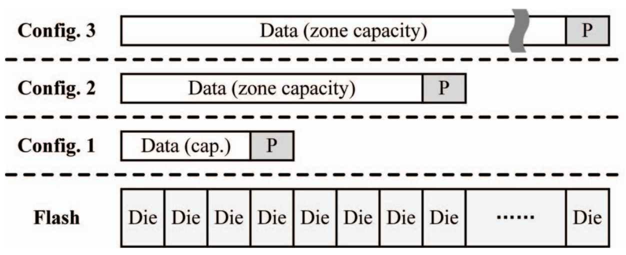

Fig. 1. **Mapping of zones in different size configurations.** Each zone is mapped to a parity-protected FBG spanning a number of flash dies.

TABLE I
STORAGE UTILIZATION IN DIFFERENT ZONE SIZE CONFIGURATIONS

RAID Stripe	Zone size	Zone capacity	Utilization
63+1p	2048 MB	2016 MB	98.44%
31+1p	1024 MB	992 MB	96.88%
15+1p	512 MB	480 MB	93.75%
7+1p	256 MB	224 MB	87.50%
3+1p	128 MB	96 MB	75%

off in a cost-effective manner, where FlexZNS is employed to provide a combination of small zones and large zones. It is found that FlexZNS with only 6.25% of the SSD capacity configured as small zones (parity storage cost is 3.52% of SSD capacity) can achieve comparable performance and WA to FlexZNS fully configured with small zones in RocksDB.

In addition, we verify the efficiency of FlexZNS in a multi-tenant scenario by assigning each tenant a set of small zones for performance isolation. The impact of the number of open zones under different zone sizes on performance is also studied in FlexZNS. The results reveal that a small zone size configuration can utilize fewer hardware resources for maintaining open flash blocks to achieve higher performance than a large zone size configuration.

II. MOTIVATION

We have conducted experiments to study the impact of zone size on GC efficiency in the two most popular ZNS-compatible applications, F2FS and RocksDB. F2FS is a log-structured file system that supports ZNS storage, while RocksDB is an LSM-tree-based key-value store and can run on a ZNS SSD through a file system plugin ZenFS. The ZNS SSD is built on a widely used SSD emulator FEMU [12]. Detailed experimental configurations and workload characteristics are present in Section IV-A. When free storage space is running out, F2FS or ZenFS would trigger zone GC operations. Valid data pages in victim zones are migrated and then the zones are reset to be free.

The ZNS SSD is divided into zones of the same size. A zone is mapped to a group of flash blocks that span a number of (say N) dies and contain die-level RAID parity, as shown in Figure 1. When the zone size or N decreases, the parity storage overhead grows and the storage utilization (i.e., the ratio between zone capacity and zone size) degrades. Table I lists the zone size configurations, ranging from 128MB to 2GB, and corresponding zone capacities used in the experiments.

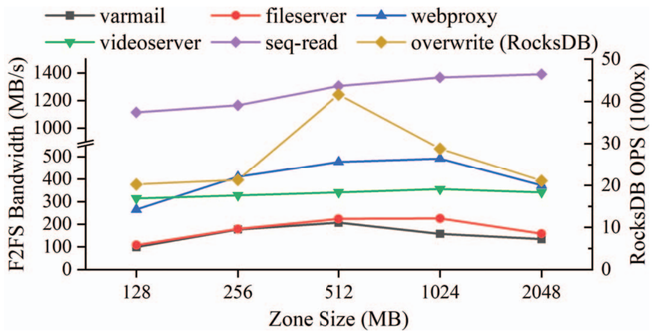

Fig. 2. **Performance under different zone size configurations.** Five workloads are used for F2FS, while the *overwrite* workload runs over RocksDB.

The zone size has two-fold effects on GC efficiency. First, with a large zone size, it is likely to cause data with different lifetimes mixed in the same zone and lead to high data migration overhead during zone GC operations. Second, given the same flash storage capacity, a smaller zone size results in a smaller user-writable capacity, where less invalid data could be accommodated. Also, the GC frequency would increase, which may relocate short-lived data that is temporally valid, as each GC operation releases a smaller amount of free space. Therefore, it would adversely affect the GC efficiency when the zone size is configured too small.

Fig. 2 exhibits the system performance of F2FS and RocksDB under various workloads when the zone size of the ZNS SSD changes from 128MB to 2048MB. The performance is low when the zone size is very small. This is because the zone GC overhead is non-trivial and also the access speed of a zone is slow (due to low intra-zone flash parallelism). As the zone size increases, the performance grows first but then degrades because of high zone GC overhead, except in the *seq-read* workload. Compared to the 2GB zone size configuration, the 512MB zone size configuration results in significantly higher system performance, up to 1.54× in F2FS and 1.97× RocksDB, respectively. In the *seq-read* workload, which contains only sequential reads, no GC operations occur and the optimal zone size configuration is 2GB (because such zones have high flash parallelism).

To verify the GC efficiency, we take the *varmail* workload in F2FS as an example to show the breakdown of *write amplification (WA)* in terms of different types of writes under different zone size configurations in Fig. 3. The WA refers to the ratio between the total number of page writes and the number of page writes in the workload. The results indicate that zone GC is the main contributor to the WA and increases it by 1.1∼2.4. The minimum and maximum cases occur when the zone size is 512MB and 2GB, respectively. The average ratio of valid data in GC zones increases from 26.1% to 33.1% when the zone size grows from 512MB to 2GB.

In summary, the zone size configuration is an essential factor influencing the performance and lifetime of zone storage systems. Taking the diversity of data-intensive applications into consideration, the fixed-size zone abstraction of the ZNS interface becomes a main obstacle to constructing flexible and

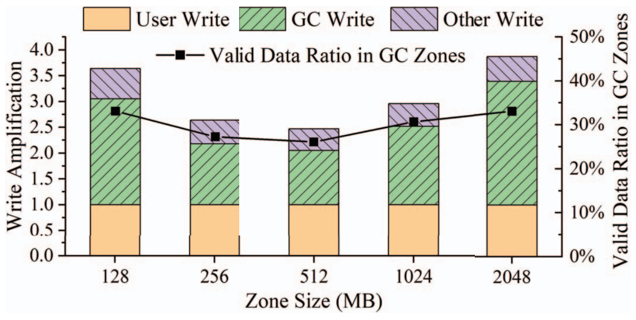

Fig. 3. **Breakdown of write amplification in the *varmail* workload on F2FS under different zone size configurations.** User writes and GC writes refer to the writes from the workload and zone GC operations, respectively. Other writes mainly include metadata and checkpointing writes in F2FS.

efficient software-defined storage systems. Especially, existing ZNS SSDs employ a large zone size in GBs to enable die-level RAID protection at a low cost, which leads to significant GC overhead on the host side. In conclusion, it is highly desirable to extend the ZNS interface and build high-performance and reliable ZNS SSDs with flexible-size zones.

III. FlexZNS

A. Design Overview

We propose a novel ZNS SSD design, called FlexZNS, to allow applications to configure zone sizes flexibly for efficient data placement, as shown in Fig. 4. The logical address space can be abstracted as either an array of zones of a fixed and customized size or multiple sets of zones, each set with a different size. Generally (but not necessarily), the available zone sizes align with the power of two (e.g., 256MB, 512MB, 1GB, and 2GB), as storage software typically manages logical address space or data in a unit aligned with the power of two.

For reliability guarantee, each zone is mapped to a *flash block group (FBG)* spanning a number of dies and protected by die-level RAID parity. The parity storage overhead would lead to variable user-writable capacity under different zone size configurations (see Table I in Section II). For example, the writable capacity of a zone sized at 1GB or 512MB would be 992MB or 480MB, respectively (see Table I in Section II); then, to store 1GB of data with the same lifetime, an application would have to allocate two 1GB-size zones or three 512MB-size zones. This not only complicates the storage space management, but also may cause data with different lifetimes to be mixed in the same zone and thus result in high GC overhead.

Considering diverse zone size configurations and to address the above concerns, FlexZNS decouples parity storage from user data on flash memory and aligns the capacity of each zone to its nominal size. Then, applications can efficiently manage the logical address space and data placement without handling irregular zone capacities. The complexity of storage management is offloaded into the ZNS SSD. Specifically, flash storage is organized in superblocks (a superblock consists of flash blocks with the same offset across all dies in the SSD), and they are divided into a data area and a parity area. In the

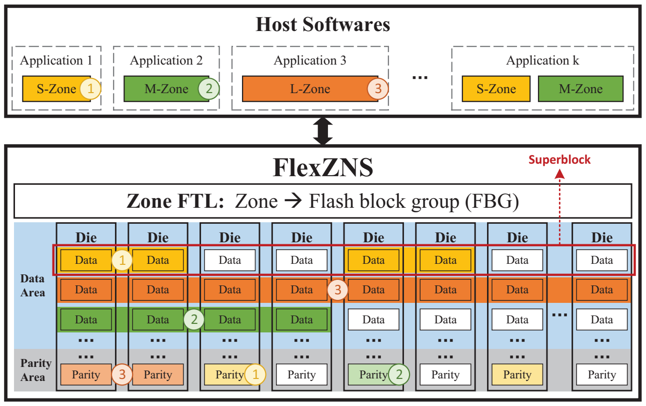

Fig. 4. **Design overview of FlexZNS.** The SSD FTL maps zones of different sizes to flash block groups in separate superblocks, while the die-level RAID parity of each zone is stored separately from user data. *S*: small-size, *M*: medium-size, *L*: large-size.

data area, zones of different sizes are assigned to different sets of superblocks. In the parity area, the die-level RAID parity of each zone is stored in a separate die that does not overlap with the FBG mapped to the zone. Thus, if a die failure occurs, the data could be reconstructed from RAID stripes. In a special case, where the size of a zone is configured as large as the superblock size or a zone is mapped to a superblock, its parity is duplicated and placed in two dies.

B. Zone Capacity Management

FlexZNS provides an API for the host software to configure the zone size through a customized NVMe command, named *zone_format(zone_size_list, zone_num_list)*. Based on the zone size configuration, FlexZNS calculates the amount of flash storage required for accommodating the total parity of zones. Suppose that an SSD is formatted into *M* sets of zones of different sizes, denoted by S_i ($i \in [1, M]$), each with N_i zones. The data area size is $\sum_{i=1}^{M} S_i * N_i$, while the parity area size can be calculated as $S_p * (N_{sb} + \sum_{i=1}^{M} N_i)$, where S_p is the parity chunk size and N_{sb} is the number of superblock-sized zones (that need two parity chunks). If the total size of the data area and parity area exceeds the physical capacity of flash memory, the *zone_format* command would return an error of space shortage and the user should intervene to re-configure the zoned layout and zone sizes.

For instance, suppose that an SSD having a physical capacity of 1TB is formatted into two sets of zones with sizes of 512MB and 2GB. Each set has 64 and 480 zones, respectively. The parity chunk size is 32MB. Then the data area size is $512MB*64+2GB*480 = 992GB$, and the parity area size is $32MB*(480+64+480) = 32GB$. The parity space overhead of this configuration is 3.13% of the total capacity. A naive method to reduce the parity storage overhead is to construct a RAID stripe over multiple zones in a superblock (the parity is shared by the zones). Whenever a zone in a stripe is reset and written with new data, the parity blocks in the superblock would have to be updated on flash memory, leading to high performance and WA overheads. FlexZNS does not adopt this method and we argue that the parity storage overhead could be compensated and the total storage cost can decrease when small zones are configured (see Section III-D).

C. Zone Mapping

To enable flexible zone size configurations and hide the resulting zone capacity loss, FlexZNS proposes a novel zone mapping mechanism to manage data and parity separately. As shown in Fig 4, a zone is mapped to an FBG in the data area, while its parity chunk locates in the parity area. The FTL of FlexZNS maintains a mapping table between zones and FBGs as well as their parity chunks. The table is kept in SSD-internal DRAM protected by capacitors and persisted on flash memory periodically or conditionally (e.g., when the system shuts down). For instance, for a 1TB SSD with all 512MB-sized or all 1GB-sized, or all 2GB-sized zones, the table size is 32KB, 16KB, or 12KB (assuming the superblock size is 2GB and thus two parity chunks are required for 2GB-sized zones), respectively. After the system powers on, the mapping table is read from flash memory to restore the zoned layout and zone sizes.

Data Mapping: FlexZNS employs an efficient data mapping strategy that maps a zone to an FBG on flash memory, which consists of flash blocks spanning multiple dies and is part of a superblock. The zoned layout, including the numbers of zones in different sets each with a specific zone size, is deterministic since the ZNS SSD is formatted. Thus, the number of FBGs mapped to each set of zones is fixed. To simplify flash space allocation and avoid space fragmentation, FlexZNS assigns a certain number of flash superblocks in the data area to accommodate each set of zones and splits each superblock into multiple FBGs of the relevant zone size. When a zone is open for writing, a free FBG with the same size is allocated and referenced by the zone. As zones are mapped to FBGs dynamically, it leaves room for improving load balance between flash dies, for reducing I/O interference between zones according to the access characteristics of zones, and for performing wear leveling between flash superblocks. In the current implementation, we adopt a simple FBG allocation policy that maintains a list of free FBGs for each set of zones of the same size and chooses the FBG located on the dies that contain the least number of open zones/FBGs.

Parity Mapping: Unlike dynamic data mapping, FlexZNS assigns flash blocks in the parity area statically to store the parity chunk for every FBG when the SSD is formatted and the zoned layout is given. Once an FBG is mapped by a zone, the parity of data in this zone is written to the pre-allocated flash blocks. Note that RAID protection is performed at the die level, as a die is the basic unit to execute flash commands independently and may fail unexpectedly. The choice of the die for the parity chunk of an FBG should guarantee that the die does not overlap with the dies where the FBG resides. Regarding a zone mapped to a superblock spanning all flash dies, two duplicate parity chunks are stored in separate dies for the zone. Such static parity mapping eliminates the runtime space allocation overhead and the mapping update overhead.

Algorithm 1 Parity Mapping Algorithm

```
SortFBGs() // $by\ size\ in\ descending\ order$
for $FBG_i$ isn't superblock-sized do
    // $from\ unfilled\ parity\ superblocks$
    $die \leftarrow$ GetDie($FBG_i$)
    if $die$ is not available then
        AllocateNewParitySuperblock()
        $die \leftarrow$ GetDie($FBG_i$)
    end if
    MapParity($FBG_i$, $die$)
end for
for $FBG_{sb}$ is superblock-sized do
    // $from\ unfilled\ or\ new\ parity\ superblocks$
    $dies \leftarrow$ GetDifferentDies(2)
    MapParity($FBG_{sb}$, $dies$)
end for
```

Specifically, the parity mapping algorithm is described in Algorithm 1. To optimize parity space utilization, it maps parity chunks of FBGs of the same size to the same parity superblock whenever possible. First, for FBGs whose size is smaller than a superblock, FlexZNS sorts them by size in descending order. A larger FBG spans a higher number of dies and fewer dies are available for placing its parity chunk, while the mapping of the parity chunk of a smaller FBG has better flexibility. Thus, the parity chunks of large FBGs are located on flash memory first. Then, FlexZNS maps the parity chunks of superblock-sized FBGs, if exist, to two different dies that have free space in parity superblocks.

D. Benefits of FlexZNS

Reducing GC overhead and storage cost. FlexZNS allows applications to configure the zone size flexibly and thus enable efficient and customized data placement. A main advantage is to employ a smaller zone size configuration, as existing ZNS SSDs typically have a large zone size in GBs. This can significantly reduce GC overhead but has higher parity storage overhead, as analyzed in Section II. Note that the reduction in GC-induced WA indicates the improvement of flash storage lifetime and thus the decrease of storage cost. Experimental results in Sections IV-B and IV-C show that, compared with a conventional ZNS SSD with a 2GB zone size configuration (parity storage cost is 1.56%), FlexZNS with a 512MB-size zone configuration (parity storage cost is 6.25%) reduces the WA by 16.7% to 36.4% in F2FS workloads, and FlexZNS with a mixed-size zone configuration (parity storage cost is 3.52%) reduces the WA by 17.7% to 46.3% in RocksDB workloads. In total, FlexZNS is able to reduce both GC overhead and storage cost significantly.

Performance isolation in multi-tenant scenarios. To improve storage utilization, it is common practice to share an SSD among multiple tenants, where performance interference between tenants becomes a concern. By supporting small zone sizes, FlexZNS is able to map the zones of different tenants into separate sets of flash dies for performance isolation. There

TABLE II
EXPERIMENTAL SETUP

OS	Ubuntu 22.04.1 LTS (Linux-5.15)	
CPU	16 * Intel(R) Xeon(R) P-8124 CPU @ 3.00GHz	
DRAM	8GB	
ZNS SSD	# of channels: 8 Dies per channel: 8 Planes per die: 4 Blocks per plane: 128 Pages per block: 2048 Page size: 4KB	Superblock size: 2GB Parity size per zone: 32MB/64MB SSD physical capacity: 256GB Page read latency: 50μs Page write latency: 500μs Block erase latency: 5ms

TABLE III
CHARACTERISTICS OF WORKLOADS

Workloads		Characteristics
Filebench	Varmail	small files with sync write frequently
	Fileserver	middle files with random IOs
	Webproxy	middle files with multi-threaded read,write
	Videoserver	large files with read mostly
YCSB	A	50% read, 50% update (update heavily)
	B	95% read, 5% update (read mostly)
	F	50% read, 50% read-modify-write

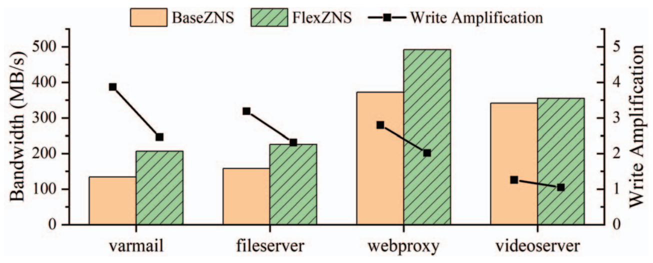

Fig. 5. **F2FS performance under difference schemes.** *BaseZNS*: the size of all zones is 2GB. *FlexZNS*: all zones are configured with the same size, 512MB or 1GB.

exists a trade-off that a smaller zone size configuration would allow FlexZNS to isolate more sets of flash dies but degrade the zone access speed and increase the parity storage cost.

Increasing the number of open zones. An SSD controller can only open a limited number of flash blocks to write data at the same time. Hence, a ZNS SSD can only support a limited number of open zones and the number is inversely proportional to the zone size [2], [7], [11]. FlexZNS is able to open more zones simultaneously for use by reducing the zone size, which can deliver higher zone-level write concurrency. This may have an important impact on performance because the write depth of an open zone may be restricted to 1. Due to the sequential write constraint of zones, host software may need to restrict the write depth of a zone to 1, for example, through the Linux *mq-deadline* I/O scheduler when running F2FS or RocksDB atop a ZNS SSD. When the request size is small, the restriction would lead to poor write performance unless zone-level write concurrency could be adequately utilized. In addition, supporting more open zones allows more tenants or applications to share the ZNS SSD, as each of them requires an independent set of open zones.

IV. EVALUATION

A. Experimental Setup

FlexZNS is implemented on FEMU [12], a widely used NVMe SSD emulator that allows modifying the SSD firmware and conducting full-stack hardware-software research. We evaluate it in two ZNS-compatible application scenarios, F2FS file system [13] and RocksDB database [2]. In particular, we modify ZenFS, which is an in-development file system plugin necessary to run RocksDB on ZNS SSDs, to support flexible configuration of zone sizes. The zone allocation and GC strategies of ZenFS are also optimized to avoid unexpected crashes during runtime and improve performance.[1]

Table II shows the experimental configurations. The server runs the Linux kernel version 5.15 and is equipped with a 16-core 3.0GHz Intel Xeon CPU and 8GB of DRAM. The emulated ZNS SSD has 8 channels, each with 8 parallel flash dies. A die includes 4 planes and a total of 512 flash blocks. A block contains 2,048 pages with a size of 4KB. Thus, the physical capacity of the SSD is 256GB. A group of flash blocks across all dies with the same offset constitute a superblock, whose size is 2GB. A zone whose size is smaller than a superblock has one parity chunk sized in 32MB (as large as 4 blocks in multiple planes of a die), while a superblock-mapped zone has two parity chunks. Flash read, write, and erase latencies are 50μs, 500μs, and 5ms, respectively [14].

We compare FlexZNS with a conventional ZNS SSD, named *BaseZNS*, which employs a fixed zone size configuration and maps each zone to a 2GB-sized superblock (including 2,016MB user-writable capacity and 32MB parity). The performance is evaluated using filebench [15] for F2FS and YCSB [16] for RocksDB. Filebench is a commonly used file system testing tool that can simulate various realistic workloads. YCSB is a database benchmark suite that provides a set of workloads with different read and write patterns. The characteristics of the workloads are summarized in Table III. All workloads are configured with a data set of about 150GB, occupying around 60% of the physical capacity of the SSD.

B. FlexZNS with Small Zones over F2FS

We first measure the overall performance and *write amplification (WA)* of BaseZNS and FlexZNS over F2FS under four filebench workloads. In FlexZNS, all zones are configured with the same small size, where 512MB or 1GB is chosen according to the best performance in the motivation experiments (see Fig. 2). Specifically, the zone size is 512MB for *varmail*, while 1GB for *fileserver*, *webproxy*, and *videoserver*.

As shown in Fig. 5, FlexZNS significantly reduces the GC-induced WA in write-dominant workloads compared with BaseZNS, i.e., by 36.4%, 27.6%, and 27.9% in *varmail*, *fileserver*, and *webproxy*, respectively. Accordingly, the performance is improved by 53.8%, 42.7%, and 32.1%, respectively. The reason why the performance improvement is relatively smaller in *webproxy* workload is that it has higher load intensity due to multi-threading and the decreased flash parallelism of small zones has a more adverse impact on the perfor-

[1]The open source code of FlexZNS is available at https://github.com/ywang-wnlo/FlexZNS-ICCD23-EA.

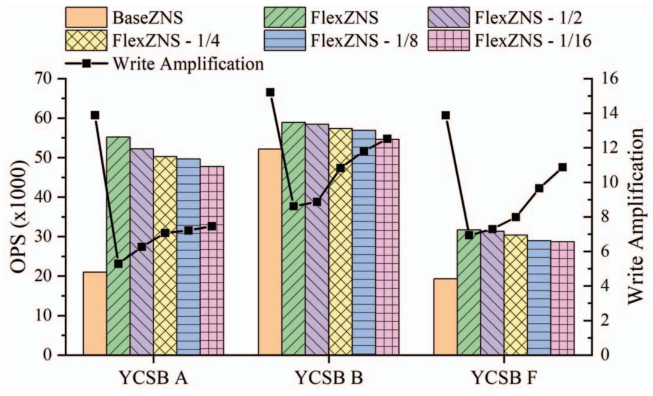

Fig. 6. **Performance comparison of different mixed-size zones configurations for YCSB workloads.** *BaseZNS*: the size of all zones is 2GB. *FlexZNS*: the size of all zones is 512MB. *FlexZNS-1/x*: 1/*x* of SSD storage space is configured with 512MB zones, while the rest contains 2GB zones.

mance. In the read-dominated *videoserver* workload, where GC frequency is low, FlexZNS performs comparably with BaseZNS. These results demonstrate that FlexZNS is effective in improving zone GC efficiency, system performance, and SSD lifetime by adopting small zones while not compromising die-level RAID protection in the SSD.

C. FlexZNS with Mixed-Size Zones over RocksDB

Despite lower GC overhead and higher performance, a smaller zone configuration causes larger parity storage overhead. We explore this trade-off by employing mixed-size zone configurations in FlexZNS. Specifically, a varying proportion of SSD storage space, from 1/16 to 100%, is configured with small 512MB-size zones, while the rest contains large 2GB-size zones. In the evaluation, we run RocksDB and ZenFS on ZNS SSDs. When mixed-size zones are used, FlexZNS assigns small zones to store data or SSTables at higher LSM-tree levels of RocksDB, where data is write-hot and GC frequency is relatively high.

Figure 6 exhibits the OPS/throughput of RocksDB and the WA caused by zone GC of ZenFS with BaseZNS and FlexZNS with different zone size configurations. The workloads include *YCSB A, B, and F*, where *A* and *B* are update-heavily and read-mostly, respectively, while *F* is mixed with reads and writes. The I/O patterns conform to a skewed Zipfian distribution. We can see, compared with BaseZNS, FlexZNS with only small zones improves performance by 2.63×, 1.12×, and 1.64× in YCSB A, B, and F workloads, respectively, while cutting down the WA by 61.9%, 43.3%, and 50.0%. As a fraction of SSD capacity is formatted as small zones, the improvements in performance and WA decrease but are still remarkable. In particular, when small zones account for 1/16 of SSD capacity, the performance gains of FlexZNS over BaseZNS are 2.28×, 1.05×, and 1.49×, respectively, while the WA reductions are 46.3%, 17.7%, and 21.7%.

On the other hand, the parity storage cost is 1.56% of SSD capacity for BaseZNS, while 3.52% and 6.25% for FlexZNS with 1/16 and 100% of SSD capacity configured as small zones, respectively. BaseZNS has an abnormal zone capacity

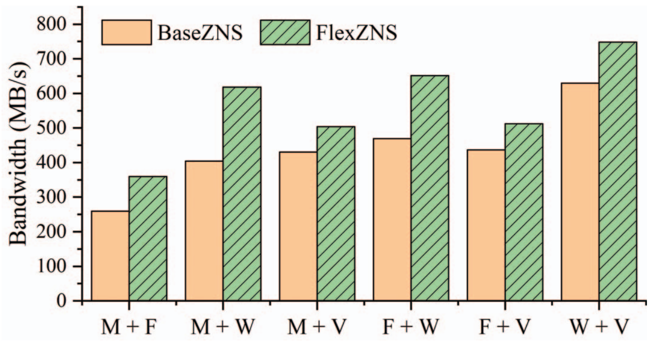

Fig. 7. **Overall performance in multi-tenant workloads over F2FS.** *M*: varmail, *F*: fileserver, *W*: webproxy, and *V*: videoserver.

(i.e., 2,016MB user-writable space) because parity is contained in the superblock mapped to each zone. FlexZNS decouples the parity storage and aligns the user-writable capacity with the nominal zone size. The extra parity storage cost of FlexZNS stems from two aspects: one is the increasing number of zones due to a small zone size configuration, and the other is the parity replication (in two separate flash dies) for large, superblock-mapped zones. It is noted that such a parity setup results in stronger RAID protection and higher SSD reliability than BaseZNS. For the same level of reliability, if BaseZNS maintains two parity chunks per zone as in superblock-mapped zones in FlexZNS, the parity storage cost increases to 3.12%. This indicates FlexZNS with 1/16 of SSD capacity configured as small zones would incur marginal parity storage overhead (3.52% vs. 3.12%).

These results verify that a small zone size configuration can also lead to significant performance and lifetime improvements in RocksDB, as in F2FS. Although it may cause a non-trivial amount of parity storage cost, FlexZNS can keep most of the benefits while minimizing the cost by utilizing a mixed zone size configuration. In the case of running skewed workloads in RocksDB, it is found that, when the number of small zones is reduced by half, the performance improvement degrades slightly while the WA increases considerably. We leave it as future work to study the trade-off of FlexZNS in mixed zone size configurations under a wider range of workloads and applications.

D. FlexZNS in Multi-Tenant Workloads

It is common to run multiple tenants (or applications) in the same ZNS SSD, as it has a large capacity and high bandwidth. Figure 7 shows the overall bandwidth of BaseZNS and FlexZNS in F2FS with multi-tenant configurations, where two tenants runs different workloads. In BaseZNS, two workloads run concurrently without any isolation. In FlexZNS, a small zone size configuration is used for each workload (as in Section IV-B) and the zones of different tenants are mapped to FBGs in separate sets of flash dies (data storage and I/O processing between tenants are isolated). The results exhibit that FlexZNS achieves 17.1% to 52.9% higher bandwidth than BaseZNS, besides the advantage of performance isolation.

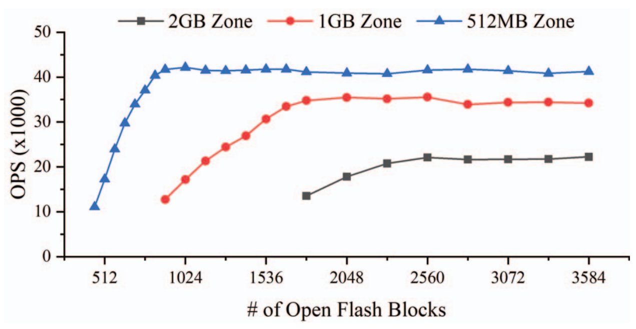

Fig. 8. **Write performance of FlexZNS in RocksDB under three zone size configurations and different limits on the maximum number of open flash blocks.**

E. Impact of Number of Open Zones

An SSD can only open a restricted number of flash blocks simultaneously to serve writes due to limited hardware resources. When the zone size increases, the maximum number of open zones that a ZNS SSD can support is reduced, as each zone is mapped to a higher number of blocks. Note that the write queue depth is limited to 1 for each open zone, because zones must be written sequentially and the NVMe interface does not provide any I/O order guarantee (even if host software issues sequential write requests, they may arrive at the SSD out of order if the write queue depth is larger than 1). Hence, open zones are valuable resources for ZNS SSDs, especially in multi-tenant scenarios, and it is important to exploit zone-level parallelism for high write performance, especially when the request size is small and flash parallelism is under-utilized. As FlexZNS allows flexible configurations of zone sizes, it offers the capability to adjust the maximum number of open zones.

Figure 8 reveals the write performance of FlexZNS in RocksDB under three zone size configurations and different limits on the maximum number of open flash blocks. When the limit becomes larger and more zones can be opened to serve concurrent writes, the performance increases linearly first and then stabilizes. By using a small zone size configuration, FlexZNS not only reduces zone GC overhead but also allows higher zone-level write concurrency. For example, the 512MB zone size configuration with 1,024 open blocks achieves 1.89× higher performance than the 2GB zone size configuration with 2,560 open blocks.

V. Related Work

The ZNS interface was standardized for SSDs by NVMe in 2020, since which a line of research has shifted from conventional block-interfaced SSDs to ZNS SSDs. Bjørling et at. [2] characterized a real ZNS SSD and demonstrated its performance benefits over block-interfaced SSDs in F2FS and RocksDB. As zone GC overhead is a key concern for ZNS storage systems, existing studies have proposed several approaches to address the concern mainly from three perspectives. First, separating data with different lifetimes has been a widely used approach to reduce data migrations of GC. Considering RocksDB is a major application that embraces ZNS SSDs, some works exploit the semantics of its LSM-tree engine to improve data placement for more accurate data separation [9], [10]. Second, ZNS+ [8] offloads data migrations of zone GC from the host into the SSD, which can eliminate data transfer overhead between the host and SSD. Third, by taking advantage of the software definability of the ZNS interface, zone GC can be coordinated or coalesced with the inherent data management of applications, such as LSM-tree compaction in RocksDB [7] and the OS swap logic [17]. Our proposed FlexZNS shares a similar goal with these works to reduce zone GC overhead but in a more fundamental manner. These works are subject to the fixed- and large-size zone configuration of existing ZNS SSDs, which manifests as a severe hardware limitation to realize accurate data separation and reduce data migrations of zone GC.

There are recent studies [11], [18] that advocate ZNS SSDs with a very small zone size configuration (i.e., 96MB). A host-side I/O scheduling scheme is proposed in [11] to be aware of flash-level I/O interference between small zones and exploit flash parallelism. eZNS [18] mitigates inter-/intra-zone interference through a tenant-cognizant I/O scheduler and improves SSD bandwidth by allocating zone resources dynamically based on the application workload profile. Note that these works omit the SSD reliability concern, as the very small zone size configuration disables die-level RAID protection. Differently, our work calls for ZNS SSDs, where the zone size can be configured flexibly and the zones can provide reliable storage. On the other hand, the methods proposed in the studies are orthogonal to FlexZNS and may be applied to FlexZNS with a small zone size configuration.

In addition, RAIZN [19] presents a device-level RAID scheme for ZNS SSDs by addressing several challenges (e.g., managing metadata updates and partial stripe writes). However, ZNS SSD RAIDs would aggravate the zone GC overhead dramatically, because the zones exposed to applications consist of multiple device zones and have a very large size, e.g., four device zones each with a user-writable capacity of 1,077MB in RAIZN. Our proposed design is desirable for building ZNS SSD RAIDs to reduce the zone size and also support flexible zone sizes.

VI. Conclusion

This paper proposes FlexZNS, a software-defined ZNS SSD that allows applications to configure variable-size zones for flexible data placement and low GC overhead. Moreover, each zone is protected by die-level RAID on flash memory for high reliability. We verify FlexZNS in both F2FS and RocksDB. Experimental results demonstrate that it outperforms conventional ZNS SSDs with a fixed (and usually large) zone size in several aspects, including performance, write amplification, and multi-tenant support. We leave it as future work to exploit the zone size configurability of FlexZNS and realize its full potential in wider application scenarios.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable feedback and comments.

REFERENCES

[1] D. Floyer, “Flash-native architectures power next-generation real-time workloads,” 2021. [Online]. Available: https://wikibon.com/flash-native-drives-real-time-business-process/

[2] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. Le Moal, G. R. Ganger, and G. Amvrosiadis, “ZNS: Avoiding the block interface tax for flash-based SSDs.” in *USENIX Annual Technical Conference*, 2021, pp. 689–703.

[3] Y. Cai, S. Ghose, E. F. Haratsch, Y. Luo, and O. Mutlu, “Error characterization, mitigation, and recovery in flash-memory-based solid-state drives,” *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1666–1704, 2017.

[4] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, “Don’t stack your log on my log,” in *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW’14)*, 2014.

[5] W. D. Corporation, “Zoned storage,” 2021. [Online]. Available: https://zonedstorage.io/

[6] Micron, “NAND flash media management through RAIN,” 2011. [Online]. Available: https://www.micron.com/-/media/client/global/documents/products/technical-marketing-brief/brief_ssd_rain.pdf

[7] D. Wu, B. Liu, W. Zhao, and W. Tong, “ZNSKV: Reducing data migration in LSMT-based KV stores on ZNS SSDs,” in *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 2022, pp. 411–414.

[8] K. Han, H. Gwak, D. Shin, and J. Hwang, “ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction.” in *OSDI*, 2021, pp. 147–162.

[9] H.-R. Lee, C.-G. Lee, S. Lee, and Y. Kim, “Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs,” in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022, pp. 93–99.

[10] J. Jung and D. Shin, “Lifetime-leveling LSM-tree compaction for ZNS SSD,” in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022, pp. 100–105.

[11] H. Bae, J. Kim, M. Kwon, and M. Jung, “What you can’t forget: Exploiting parallelism for zoned namespaces,” in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage’22)*, 2022.

[12] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, “The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator,” in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 83–90.

[13] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, “F2FS: A new file system for flash storage.” in *FAST*, vol. 15, 2015, pp. 273–286.

[14] B. S. Kim, J. Choi, and S. L. Min, “Design tradeoffs for SSD reliability,” in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 281–294.

[15] V. Tarasov, “Filebench - a model based file system workload generator,” 2011. [Online]. Available: https://github.com/filebench/filebench

[16] Yahoo, “Yahoo! cloud serving benchmark (YCSB),” 2010. [Online]. Available: https://github.com/brianfrankcooper/YCSB

[17] S. Bergman, N. Cassel, M. Bjørling, and M. Silberstein, “ZNSwap: un-Block your Swap,” in *Proceedings of the USENIX Annual Technical Conference (ATC’22)*, 2022.

[18] J. Min, C. Zhao, M. Liu, and A. Krishnamurthy, “eZNS: An elastic zoned namespace for commodity ZNS SSDs,” in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 461–477.

[19] T. Kim, J. Jeon, N. Arora, H. Li, M. Kaminsky, D. G. Andersen, G. R. Ganger, G. Amvrosiadis, and M. Bjørling, “Raizn: Redundant array of independent zoned namespaces,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 660–673.