Para-ZNS: Improving Small-zone ZNS SSDs Parallelism through Dynamic Zone Mapping

Zhenhua Tan[1], Linbo Long[1*], Jingcheng Shen[1], Congming Gao[2], Renping Liu[1], and Yi Jiang[1]

[1]School of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China
[2]School of Information Science and Engineering, Xiamen University, Xiamen 361005, China

***Abstract*—The emerging Zoned Namespace (ZNS) interface helps flash-based SSDs achieve high performance by dividing the logical space into fixed-size zones. Typically, a zone is mapped to blocks across multiple dies to achieve I/O parallelism. Small zones can make better use of space and are therefore widely studied. However, a small zone fails to be mapped to blocks residing on all dies, causing underutilized die-level parallelism. Meanwhile, a fine-grained (i.e., plane-level) parallelism is rarely exploited for ZNS SSDs due to a strict limitation mandating that only the same type of operation can be simultaneously performed on the same address across different planes within a die. To address these issues, this paper proposes a novel small-zone ZNS-SSD design with dynamic zone mapping, named Para-ZNS. First, a new parallel block grouping module is devised to group blocks across all planes from multiple dies as a basic unit to be mapped to a zone. Such a basic mapping unit achieves parallelism among multiple dies and plane-level parallelism. Then, a die-parallelism identification module is implemented to locate idle dies. Subsequently, to fully exploit the die-level parallelism, a dynamic zone mapping scheme is employed to intelligently map the basic mapping units on the identified idle dies to open zones. The evaluation results based on a widely-used I/O tester (FIO) demonstrate that Para-ZNS improves the bandwidth by $3.42\times$ on average in comparison to state-of-the-art work.**

***Index Terms*—ZNS-Interface, Flash-based SSDs, Parallelism, Zone Mapping**

I. Introduction

The NVMe Zoned Namespace (ZNS) interface has emerged as an alternative to the traditional block interface for flash-based SSDs [1–4]. ZNS SSDs divide the logical address space into fixed-size zones and thus mitigate the write amplification issue by performing effective data placement at the zone level [5–9]. Moreover, each zone is mapped to blocks across multiple dies from flash chips to exploit the die-level parallelism achieving high I/O performance [10–12]. The size of zone is typically small (e.g., 96 MB, 128 MB) [10, 13–15], causing a zone to be mapped to blocks from a proportion of dies in the flash chips. To make matters worse, the host lacks means to manage the mapping relationship between zones and dies for zone allocation, resulting in a low efficiency in exploiting die-level parallelism. Meanwhile, a fine-grained (i.e., plane-level) parallelism is not considered to be leveraged in ZNS SSDs. To address these issues, this paper focuses on enhancing the zone mapping mechanism by proposing a novel dynamic zone mapping scheme for small-zone ZNS SSDs, which improves both die-level and plane-level parallelism.

*Corresponding author: Linbo Long. E-mail: longlb@cqupt.edu.cn.

Some prior studies have been proposed to improve the I/O parallelism for ZNS SSDs. Im *et al.* [12, 13, 15, 16] first proposed to simultaneously operate on more zones to improve die-level parallelism. Bae *et al.* [14] proposed to detect and avoid conflicts between zones with probing requests and scheduling, respectively. Moreover, SplitZNS [11] improves die-level parallelism by adding buffers in chips to distribute write requests to multiple dies. Finally, eZNS [17] proposes a logical zone (i.e., v-zone) to improve die-level parallelism by adaptively adjusting its striping configuration and hardware resources during writing.

In general, the aforementioned studies can only take advantage of die-level parallelism at the cost of a substantial overhead or in restricted scenarios, hence resulting in low scalability and high software complexity. Besides, the plane-level parallelism in ZNS SSDs is rarely harnessed due to a strict limitation that requires the same type of operation to be simultaneously performed on the same address in different planes. Compared to the host, the device can easily access the mapping information between zones and blocks. As a result, the plane-level and die-level parallelism can be more easily exploited by mapping blocks on planes from different dies to open zones (i.e., the zones being concurrently written). Based on this observation, we therefore focus on fully exploiting the plane-level and die-level parallelism of ZNS SSDs using device-side dynamic zone mapping.

To these ends, this paper proposes a novel small-zone ZNS-SSD design with dynamic zone-address mapping, named **Para-ZNS**. The basic idea is to dynamically map blocks residing on planes from multiple dies to open zones, which exposes maximal plane-level and die-level parallelism to applications running on ZNS SSDs. First, a new parallel block grouping module is designed to manage basic mapping units. To ensure the parallelism among multiple dies and the plane-level parallelism, the basic mapping unit consists of blocks on all planes from multiple dies. Secondly, according to zone states (i.e., *Open, Closed, Empty and Full*), a die-level parallelism identification module is presented to examine the usage of dies and thus identify idle dies. Thirdly, a dynamic zone mapping scheme is given to dynamically map basic mapping units on the idle dies to open zones, notably enhancing the die-level parallelism of ZNS SSDs.

The primary contributions of this paper can be summarized as follows:

- A grouping mechanism of the basic mapping unit is proposed to ensure that accesses to the basic mapping

979-8-3503-4859-0/DATE24/(c) 2024 EDAA

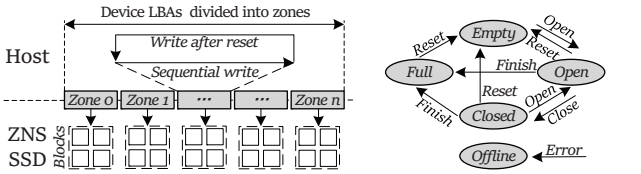
unit can exploit parallelism among multiple dies and plane-level parallelism.
- A device-side die-parallelism identification is designed to identify idle dies for further utilization according to zone states.
- A dynamic zone mapping mechanism is presented to maximize the number of dies utilized by open zones, fully exploiting the die-level parallelism of ZNS SSDs.

The remainder of this paper is organized as follows. Section II and Section III present the background and the motivation of this paper, respectively. In Section IV, we offer the details of Para-ZNS. Section V presents the experimental results and the related analysis. Finally, we discuss related work in Section VI and conclude this paper in Section VII.

II. Background

In this section, the basic structure of ZNS SSDs is first given. Then, we illustrate the current zone mapping mechanism of small-zone ZNS SSDs and discuss its efficiency in exposing parallelism.

A. ZNS SSDs



(a) ZNS interface. (b) Zone states and Transitions.

Fig. 1. Basic structure of ZNS interface and ZNS states.

As shown in Fig. 1a, the ZNS interface divides logical address space into fixed-size and sequential-write zones, with each zone being mapped to a collection of physical blocks in flash chips [1, 4]. Besides, Fig. 1b shows the transitions between the five zone states: *Empty*, *Open*, *Closed*, *Full*, and *Offline*. When a zone is reset, its state transitions to *Empty*, indicating that the zone contains no data within it. When a zone is opened, its state transitions to *Open* and the zone needs to be mapped. When an open zone is closed, its state transitions from *Open* to *Closed* and the resources occupied by the zone are freed. When a zone with the state *Open* or *Closed* is no longer being written, its state transitions to *Full* by the *Finish* command. Moreover, the zone state of *Offline* is configured when an error occurs in the zone.

In summary, an empty or closed zone must be opened to be associated with internal device resources before being written. Simultaneously, an open zone will be closed, finished, or reset once write operations on this zone stop. By inspecting the zone states and the commands received by the device, we can obtain the zones that are being or going to be written by the host. Furthermore, the die utilization can be easily calculated and predicted combined with the mapping information between zones and dies on the device side.

B. Current Zone-Mapping Mechanism of Small-zone ZNS SSDs

ZNS SSDs can be classified into small-zone ZNS SSDs and large-zone ZNS SSDs [10–12, 17]. Large-zone ZNS SSDs have a large zone size (e.g., 1GB, 2GB) to map a zone to blocks across all dies in the flash chips, thereby fully taking advantage of the die-level parallelism when accessing a single zone. However, a large zone size may cause low space utilization, especially for applications with a moderate amount of data [11, 12, 14]. On the contrary, small-zone ZNS SSDs are explored to achieve high space utilization for less space-demanding applications. Small-zone ZNS SSDs are typically configured with a small zone size (e.g., 96MB, 128MB), yet a small zone can only be mapped to blocks on a proportion of dies in the flash chips (Fig. 2). Thus, when a zone is being written, only partial parallelism on the dies mapped to the zone can be exploited (e.g., writes on zone 0 can only exploit the parallelism on die *0-3*). To enhance the die-level parallelism, the existing works primarily focus on host-side software optimizations to simultaneously write more non-conflicting zones (i.e., zones mapped to different dies).

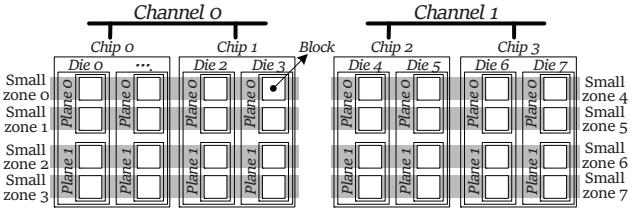


Fig. 2. Current zone mapping mechanism of small-zone ZNS SSDs.

Nevertheless, the host cannot retrieve the mapping information between zones and dies. Despite the prior research efforts can improve the die-level parallelism by avoiding the write on conflicting zones, a significant overhead is incurred which reduces the performance benefit of exploiting die-level parallelism. Moreover, the current mapping mechanism in ZNS SSDs ignores the plane-level parallelism. To harness the plane-level parallelism in ZNS SSDs, the host must ensure operations and addresses are consistent on different planes within a die [18, 19]. Since blocks in different planes within a die is dispersedly mapped to different zones, it is difficult for the host to retrieve the mapping information between zones and planes from different dies. Another challenge is to ensure that operations with the same type are performed on the same address in different planes of a die. These observations therefore indicate that device-side optimizations are necessary for zone mapping mechanism to enhance the plane-level and die-level parallelism for small-zone ZNS SSDs.

III. Motivation

In this section, we evaluate the die and plane utilization on a small-zone ZNS SSD, which employs the mapping mechanism described in Section II-B. The detailed experiential setup is presented in Section V. On the host side, the number of open zones (i.e., the number of zones that can be concurrently written by the host) is set to 2, 4, and 8 for three cases, respectively. In addition, three zone allocation schemes are examined, including *Seq* (sequentially allocating zones), *Ran* (randomly allocating zones), and *CG* [14]. *CG* detects conflict zone groups and prioritizes the execution of

I/O requests from different conflict zone groups to improve parallelism.

A. Die Utilization in ZNS SSDs

The die utilization exhibits a large gap in small-zone ZNS SSDs with various zone-allocation schemes and different numbers of open zones . As shown in Fig. 3a, we first set the number of open zones to 4 and record die utilization and bandwidth for the three zone-allocation schemes. Compared to *Seq*, *Ran* increases die utilization by 1.8× and therefore achieves a bandwidth 1.8× as fast. Since *CG* prioritizes the execution of I/O requests from zone groups that are mutually non-conflicting, *CG* increases both die utilization and bandwidth by 2× in comparison to *Ran*. According to these results, die utilization varies notably for different zone-allocation schemes and achieves its maximum for *CG*.

However, the implementation of *CG* incurs significant overhead [17]. Specifically, *CG* employs a conflict detection mechanism which sends respectively a page-sized request to two zones and detects a conflict by comparing the achieved bandwidth with a threshold. To obtain the threshold and detect conflicts between all zones, the mechanism needs to send N×M requests, where N is the number of zones and M is the number of conflict groups [14]. For example, a 2-TB ZNS SSD is configured with a zone size of 128 MB (i.e., N=16384) and a page size of 16 KB. Given eight conflict groups (i.e., M=8), a single implementation of *CG* requires sending useless requests that amount to 4 GB. Consequently, this incurs significant overhead and obtained *CG* only works as long as the Zone-Block mapping is unchanged.

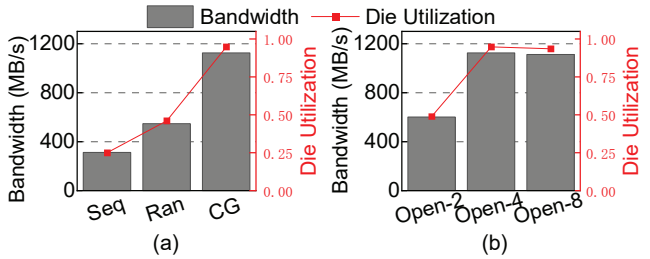


Fig. 3. Die utilization and bandwidth: (a) the die utilization and bandwidth on ZNS SSDs with different zones allocation scheme and (b) the utilization and bandwidth on ZNS SSDs with different the number of zones allocated to be opened.

Furthermore, we record die utilization and bandwidth achieved by *CG* for different numbers of open zones (2, 4, and 8). *Open-N* represents the number of open zones is *N*. As shown in Fig. 3b, *Open-2* only achieves 50% of the die utilization and bandwidth achieved by *Open-4* and *Open-8*. Such a low utilization is because a small zone is mapped to blocks from 25% of the dies in the ZNS SSD and two concurrent open zones therefore can only utilize 50% of the dies. The results indicate that the die utilization of small-zone ZNS SSDs is also impacted by the number of open zones. In general, we must ensure that sufficient non-conflicting zones can be simultaneously written so as to fully exploit die parallelism.

B. Plane Utilization on ZNS SSDs

The plane-level parallelism is hardly exploited in ZNS SSDs with the current zone-mapping mechanism. Typically, a die comprises two planes, but only one plane can be used at the same time. As shown in Fig. 4, the average plane utilization is merely 50% for small-zone ZNS SSDs with different zone-allocation schemes and numbers of open zones. To analyze the reasons for the low plane utilization, we further examine three access operation categories: 1) *SinPl*: only a single plane in a die is accessed, 2) *DiffAddr*: different addresses of two planes within a die are accessed at the same time, and 3) *Para*: two identical access operations simultaneously access the same address in both planes, allowing parallel execution. As shown in Fig. 4, *SinPl* and *DiffAddr* account for 84% and 15% of the total number of access operations, respectively. Almost all access operations are executed in one plane in different dies or different addresses of different planes in the same die. These access operations cannot exploit the plane-level parallelism due to the strict limitation that requires the same type of operation to be simultaneously performed on the same address in different planes of a die. Consequently, the plane-level parallelism cannot be exploited due to the current zone-mapping mechanism of ZNS SSDs.

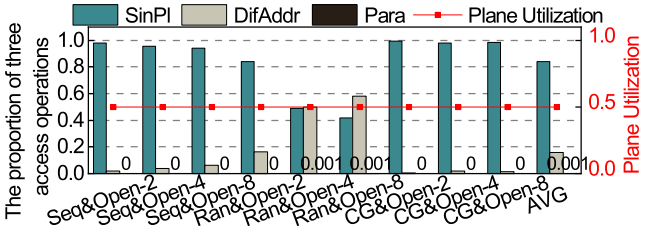


Fig. 4. Plane utilization and three access operation catagories.

Summarily, the die-level and plane-level parallelism in small-zone ZNS SSDs is difficult to be fully exploited due to current zone-mapping mechanism. Therefore, Section IV presents a novel small-zone ZNS-SSD design with dynamic zone mapping to maximize the die-level and plane-level parallelism.

IV. Para-ZNS: A Novel Small-Zone ZNS-SSD Design with Dynamic Zone Mapping

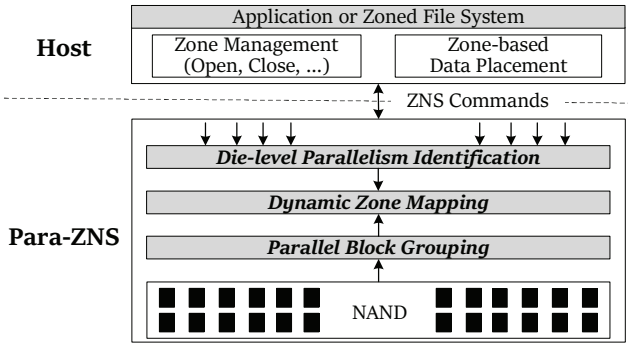


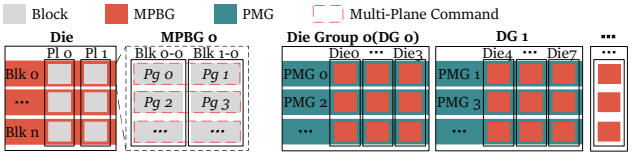
Fig. 5. Overview of Para-ZNS.

In this section, we present a small-zone ZNS-SSD design with dynamic zone mapping, named **Para-ZNS**. As shown in Fig. 5, a *Parallel Block Grouping* module is first designed

to reorganize the basic mapping unit for a zone, ensuring the parallelism among multiple dies and the plane-level parallelism. Then, a *Die-level Parallelism Identification* module is given to examine die utilization and identify idle dies (i.e., dies freed from the open zones). To fully exploit die-level parallelism, Para-ZNS further proposes a new *Dynamic Zone Mapping* scheme to dynamically map basic mapping units on the identified idle dies to newly opened zones.

A. Parallel Block Grouping

As described in Section II-B and Section III-B, the current zone-mapping mechanism ensures the parallelism among multiple dies each time a zone is accessed. However, yet the plane-level parallelism is ignored in zone mapping and is thus rarely leveraged. Therefore, this section presents a new parallel block grouping module to reorganize the basic mapping unit to guarantee plane-level parallelism.



(a) The structure of MPBG. (b) Internal organization structure.

Fig. 6. Basic mapping unit of Para-ZNS.

Figure 6 shows the structure of the basic mapping unit. As shown in Fig. 6a, we first group the blocks with the same address offset on all (i.e., two) planes within a die into a block group, named MPBG (multiple plane block group). MPBG 0 serves as an example that consists of block 0 on plane 0 (*Block 0-0*) and block 0 on plane 1 (*Block 1-0*). According to the sequential-write feature of ZNS SSDs, write operations on MPBG 0 can be executed in parallel on plane 0 and plane 1 because *Block 0-0* and *Block 1-0* have the same offset. Generally, *Multi-Plane Command* can execute read and write requests on an MPBG in parallel, achieving plane-level parallelism.

Furthermore, we use MPBG as the minimal granularity to form a zone mapping unit (Fig. 6b). First, all dies in flash chips are divided into multiple die groups (DGs). Each DG has the same number of dies. Then, MPBGs with same offset in each DG is further organized as a block group, named a parallel MPBG group (PMG). Specially, the number of dies in each DG is equal to the ratio of the zone size to the MPBG size. A PMG can be used as the basic mapping unit for a zone. In doing so, access operations on a zone can achieve plane-level parallelism as well as maximize the die-level parallelism exposed to the zone. Nevertheless, a small zone fails to be mapped to blocks across all dies, causing underutilized die-level parallelism. Therefore, we propose a module to inspect the die utilization and thus locate idle dies in next section.

B. Die-level Parallelism Identification

To retrieve the information of die utilization and identify idle dies, the die-level parallelism identification module categorizes the zones states into three groups, including *Waiting zones*, *Ready zones*, and *Running zones*. As shown in Fig. 7, when a zone is opened, its state transitions to *Open* and the zone is categorized into *Ready zones*. If a zone in *Ready zones* has already been mapped to basic mapping units, the zone is directly moved into *Running zones* and waits to be written. Otherwise, the zone waits to be mapped to basic mapping units. When the zone state transitions to *Closed*, *Reset*, or *Full*, the zone is moved to *Waiting zones* and waits to be opened.

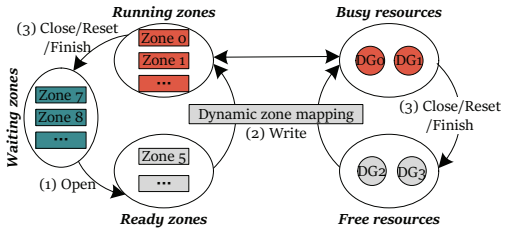
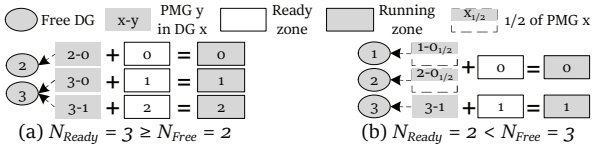


Fig. 7. Diagram of die-level parallelism identification.

By inspecting the zones categorized into the three zone groups, we can retrieve the information of die utilization and locate idle dies. The basic idea is similar to that of CPU resource allocation. A newly opened zone is moved into *Ready zones* to wait for die resource allocation. The die resource is the die group (DG) utilized by the zone. After zone mapping, the zone is moved to *Running zones*, its die resource remains occupied (i.e., busy). Only the dies occupied by zones in *Running zones* are busy resources. Conversely, the other die resources are idle and can be considered as free resources. Therefore, we prioritize forming basic mapping units (i.e., PMGs) with free resources (i.e., idle die groups) and mapping the PMGs to zones in *Ready zones*. The details of zone mapping are illustrated in next section.

C. Dynamic Zone Mapping

In this section, we propose a new dynamic zone mapping scheme to map basic mapping units (i.e., PMG) to zones in the *Ready zones*. The basic idea is to prioritize mapping the basic mapping unit on free resources to maximize the die-level parallelism of ZNS SSDs. The zone mapping process can be divided into two cases.



(a) $N_{Ready} = 3 \geq N_{Free} = 2$ (b) $N_{Ready} = 2 < N_{Free} = 3$

Fig. 8. Diagram of dynamic zone mapping.

Case 1*: the number of zones in Ready zones (N_{Ready}) is larger than or equal to that of the idle DGs (N_{Free}).* In this case, we prioritize mapping a PMG on an idle DG to a zone so as to put the idle DG into utilization. We unrepeatably map PMGs in the idle DGs to zones until all the idle DGs are utilized, thereby fully exploiting all dies of the SSD. If there are zones that are not mapped, we loop to map the basic mapping unit on the utilized DGs to those zones, achieving the load balancing. As shown in Fig. 8a, PMG 0 in DG 2 and 3 (i.e., PMG 2-0 and PMG 3-0) are first mapped to *Ready zone* 0 and 1, respectively. Then, another PMG 1 in DG 3 (i.e.,

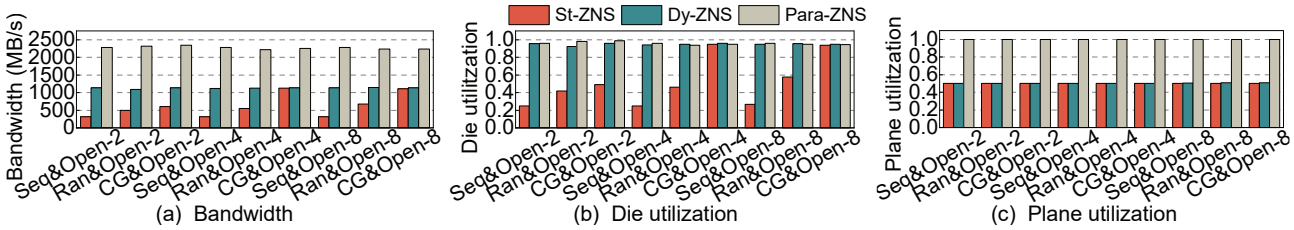


Fig. 9. Bandwidth, die utilization, and plane utilization achieved by three ZNS-SSD designs for various Randwrite workloads.

PMG 3-1) is mapped to *Ready zone* 2. Finally, these *Ready zones* are transformed to *Running zones*.

Case 2*: the number of zones in Ready zones (N_{Ready}) is smaller than the number of the idle DGs (N_{Free}).* In this case, the previously mentioned scheme cannot fully utilize the idle DGs. A new partially mapping scheme is thus proposed. The basic idea is to map partial PMGs from multiple idle DGs to form a zone so that all dies can be utilized. Besides, the rest of these PMGs can be further mapped to subsequent zones, satisfying the sequential-write requirement for blocks while avoiding space fragmentation. As shown in Fig. 8b, *1/2* of PMG 1-0 in DG 1 and *1/2* of PMG 2-0 in DG 2 are first mapped to *Ready zone* 0 in order to utilize all dies in DG 1 and 2. Then, PMG 3-1 is mapped to *Ready zone* 1. Based on this, all idle dies are utilized;

In general, combining with the corresponding method on the two cases, the zone mapping scheme can always maximize the number of utilized dies. Even if the number of open zones is smaller than that of idle DGs, we can fully exploit die-level parallelism by the partially mapping scheme.

V. Evaluation

In this section, we conduct a series of experiments to evaluate the Para-ZNS. The experimental setup is first described in Section A. Then, Section B presents the performance of Para-ZNS.

TABLE I
Experiment setup.

Configuration	Description
FEMU	FEMU version: 5.2, Linux kernel: 5.12
SSD	Channels: 8, Chips/Channel: 4, Dies/Chip: 2, Planes/Die: 2, Blocks/Plane:64, Pages/Block: 1024, Page capacity: 4KB
ZNS	Zone size: 128MB, Number of zones: 256
Flash latency	Read latency: 40us, Program latency: 200us, Erase latency: 2000us
Software Version	FIO version: 3.35
FIO	Rw: randwrite/write/randread/read, Zonemode: zbd, Max_open_zones: 2/4/8, Iodepth: 64, BS=256k, Zone_alloc_order: Seq/Ran/CG, I/O mode: Direct I/O

A. Experimental setup

To evaluate Para-ZNS, we first implemented it using FEMU, a general QEMU-based SSD emulator [20]. Then, a flexible I/O tester (FIO) is used to evaluate comprehensively the performance of Para-ZNS. The detailed configurations of these experiments are shown in Table I. Moreover, we compared Para-ZNS with St-ZNS[10, 13, 14] and Dy-ZNS, where St-ZNS adopts the static zone mapping strategy shown in Figure 2 and each zone is fixedly mapped to a group of blocks spanning 16 dies, DyZNS also adopts the dynamic zone mapping mechanism but uses two blocks from the same plane to replace MPBG structure. Note that the Para-ZNS is dedicated to improve the parallelism of ZNS SSDs without increasing software complexity. Therefore, SplitZNS[11] requiring major host changes is not compared.

Further, we generated four classic workloads through FIO, including *Randwrite*, *Seqwrite*, *Randread*, and *Seqread*, where *Randwrite* opens multiple zones at the same time in a certain zone allocation scheme for writing concurrently, *Seqwrite* writes serially zone one by one, *Randread* reads concurrently and randomly multiple zones, and *Seqread* reads serially zone one by one. Furthermore, specific zone allocation scheme have been described in Section III. A&Open-n indicates that the host opens n zones for writing concurrently in A manner.

B. Performance of Para-ZNS

Randwrite **performance.** Since *Randwrite* is the most commonly used write pattern, we evaluate comprehensively the performance of Para-ZNS under various *Randwrite* workloads. As shown in Fig. 9a, compared with St-ZNS, Para-ZNS can improve the bandwidth by $7.31\times$, $4.64\times$, $3.89\times$, $7.28\times$, $4.05\times$, $2\times$, $7.22\times$, $4.29\times$, $2.01\times$, and average $4.63\times$, respectively. Meanwhile, compared with Dy-ZNS, Para-ZNS can improve the bandwidth by $2\times$, $2.12\times$, $2.05\times$, $2.03\times$, $1.97\times$, $1.97\times$, $2\times$, $1.95\times$, $1.96\times$, and average $2.01\times$, respectively. There are two reasons for this. First, as shown in Fig. 9b, Para-ZNS and Dy-ZNS exploit fully the die-level parallelism by the dynamic zone mapping, while St-ZNS causes underutilized die-level parallelism due to mapping conflicts between the open zones. Secondly, as shown in Fig. 9c, Para-ZNS effectively exploit plane-level parallelism by the MPBG structure, while Dy-ZNS and St-ZNS without MPBG structure can hardly exploit plane-level parallelism either because there are no requests distributed to multiple planes in a die at the same time, or because requests distributed to multiple planes have different in-plane addresses.

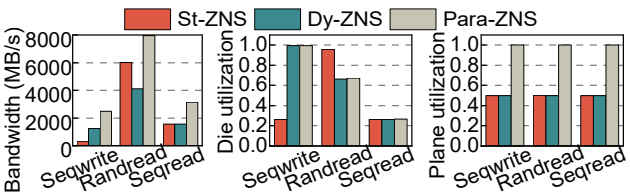


Fig. 10. Bandwidth, die utilization, and plane utilization achieved by three ZNS-SSD designs for Seqwrite, Randread, and Seqread workloads.

Seqwrite and Read performance. Further, we studied the performance of Para-ZNS under *Seqwrite*, *Randread*, and

Seqread workloads. As shown in Fig. 10, compared with St-ZNS and Dy-ZNS, Para-ZNS improves the bandwidth by $7.99\times$ and $2\times$, respectively, under *Seqwrite* workload. This is a similar result to *Randwrite* and the reasons are also similar. In addition, under *Randread* workload, compared with St-ZNS and Dy-ZNS, Para-ZNS improves the bandwidth by $1.32\times$ and $1.93\times$, respectively. And compared with St-ZNS and Dy-ZNS, Para-ZNS improves the bandwidth by $2.02\times$ and $2.01\times$, respectively, under *Seqread* workload. Although Para-ZNS also achieves higher performance under two types of read workloads, this is only because Para-ZNS exploits plane-level parallelism. The die utilization of Para-ZNS is still low, since the randomness of the read operation is too high and the access location is fixed to the write location of the target data.

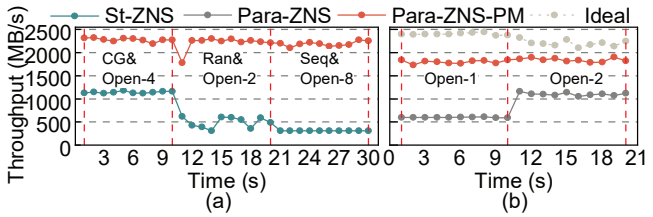


Fig. 11. Throughput. (a) throughput achieved by two ZNS SSD designs for dynamically changing workloads and (b) throughput achieved by Para-ZNS with and without partially mapping scheme (PM).

Robustness analysis. The pattern of host-to-SSD accesses definitely changes according to the application, and the garbage collection (GC) is performed to enhance space utilization. Therefore, to evaluate the robustness of Para-ZNS, we first tested the throughput of Para-ZNS under dynamically-changing *Randwrite* workloads which randomly selects the zone allocation scheme and changes the number of open zones every 10 s. As shown in Fig. 11a, Para-ZNS improves the throughput by an average factor of $3.42\times$ and achieves a stable performance in comparison to St-ZNS. Furthermore, we tested the effect of partially mapping scheme on GC. As shown in Fig. 11b, during the GC execution, under *Open-1* and *Open-2* workloads, Para-ZNS with the partially mapping scheme (Para-ZNS-PM) improves the throughput by $3\times$ and $1.6\times$, respectively, compared to Para-ZNS without PM. Although the partially mapping scheme introduces additional data migration, the traffic of the data migration occupies only a portion of the increased bandwidth (refer to the gap between Ideal and Para-ZNS-PM), and the remainder of the increased bandwidth can be exploited by normal writes. Therefore, the overall performance can still be improved.

VI. Related Work

Many studies have been proposed to improve the parallelism of ZNS SSDs. Im *et al.* [12, 13, 15, 16] initially proposed to distribute I/O requests to multiple zones to exploit the internal parallelism. *Bae et al.* [14] proposed to detect conflict between zones by sending requests in advance and further avoiding conflict by scheduling, thereby improving die-level parallelism. Moreover, SplitZNS [11] improves die-level parallelism of small zones by adding buffers in chips to distribute write requests to multiple dies. Finally, eZNS [17] was proposed to adjust zone striping configuration and mapping resources during writing so as to improve die-level parallelism. On the one hand, Para-ZNS improves die-level parallelism for small zones by implementing a lightweight dynamic zone-mapping mechanism on the device side which can be easily applied to most applications. On the other hand, compared to the prior studies that lack exploration of the plane-level parallelism in ZNS SSDs, Para-ZNS fully exploits plane-level parallelism by introducing the MPBG mapping unit.

VII. Conclusion

In this paper, a dynamic zone-mapping ZNS-SSD design, named Para-ZNS, is proposed. Para-ZNS first reorganizes the basic mapping unit to improve the plane-level parallelism within each die. Secondly, a die-level parallelism identification module is designed to retrieve the information of die utilization and thus identify idle dies. Thirdly, Para-ZNS prioritizes mapping the basic mapping units on the identified idle dies to zones so as to maximize die-level parallelism of the ZNS SSD. The experimental results show that significantly improve the performance for various workloads.

VIII. Acknowledgements

This work was supported by grants from the National Natural Science Foundation of China 62102219, 61902045, and 62172067, Chongqing High-Tech Research Key Program cstc2021jcyj-msxmX0981, cstc2021jcyj-msxmX0530, cstc2019jscx-mbdxX0035, and cstb2022nscq-msx0601, and Chongqing Natural Science Foundation Innovation and Development Joint Fund CSTB2022NSCQ-LZX0074.

References

[1] M. Bjørling, A. Aghayev, H. Holmberg, and et al., "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs," ATC, 2021, pp. 689–703.
[2] T. Stavrinos, D. S. Berger, E. Katz-Bassett, and et al., "Don't be a blockhead: zoned namespaces make work on conventional SSDs obsolete," HotOS, 2021.
[3] R. Liu, Z. Tan, Y. Shen, and et al., "Fair-ZNS: Enhancing Fairness in ZNS SSDs through Self-balancing I/O Scheduling," TCAD, 2022.
[4] S. Bergman, N. Cassel, M. Bjørling, and et al., "ZNSwap:un-Block Your Swap," ATC, 2022, pp. 1–18.
[5] Z. Tan, L. Long, R. Liu, and et al., "Optimizing Data Migration for Garbage Collection in ZNS SSDs," DATE, 2023, pp. 1–2.
[6] W. Qi, Z. Tan, J. Shao, and et al., "InDeF: An Advanced Defragmenter Supporting Migration Offloading on ZNS SSD," ICCD, 2022, pp. 307–314.
[7] H. Lee, C. Lee, S. Lee, and et al., "Compaction-aware Zone Allocation for LSM based Key-value Store on ZNS SSDs," HotStorage, 2022, pp. 93–99.
[8] J. Jung and D. Shin, "Lifetime-leveling LSM-tree Compaction for ZNS SSD," HotStorage, 2022, pp. 100–105.
[9] H. Wang, Y. Liu, P. Jin, and et al., "Efficient Data Placement for Zoned Namespaces (ZNS) SSDs," NPC, 2022, pp. 302–314.
[10] K. Han, H. Gwak, D. Shin, and et al., "ZNS+: Advanced Zoned Namespace Interface for Supporting In-storage Zone Compaction," OSDI, 2021, pp. 147–162.
[11] D. Huang, D. Feng, Q. Liu, and et al., "SplitZNS: Towards an Efficient LSM-tree on Zoned Namespace SSDs," TACO, 2023.
[12] J. Y. Ha and H. Y. Yeom, "zCeph: Achieving High Performance On Storage System Using Small Zoned ZNS SSD," SAC, 2023, pp. 1342–1351.
[13] M. Im, K. Kang, and H. Yeom, "Accelerating RocksDB for Small-zone ZNS SSDs by Parallel I/O Mechanism," Middleware, 2022, pp. 15–21.
[14] H. Bae, J. Kim, M. Kwon, and et al., "What You Can't Forget: Exploiting Parallelism for Zoned Namespaces," HotStorage, 2022, pp. 79–85.
[15] M. Oh, S. Yoo, J. Choi, and et al., "Zenfs+: Nurturing performance and isolation to zenfs," IEEE Access, vol. 11, pp. 26 344–26 357, 2023.
[16] G. Choi, K. Lee, M. Oh, and et al., "A New LSM-style Garbage Collection Scheme for ZNS SSDs," HotStorage, 2020.
[17] J. Min, C. Zhao, M. Liu, and et al., "eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs," OSDI, 2023, pp. 461–477.
[18] C. Gao, L. Shi, K. Liu, and et al., "Boosting the performance of ssds via fully exploiting the plane level parallelism," TPDS, vol. 31, no. 9, pp. 2185–2200, 2020.
[19] "Open NAND flash interface specification 5.1," https://media-www.micron.com/-/media/client/onfi/specs/onfi_5_1_final_1,-d-,0.pdf.
[20] H. Li, M. Hao, M. H. Tong, and et al., "The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator," FAST, 2018, pp. 83–90.