What You Can't Forget: Exploiting Parallelism for Zoned Namespaces

Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)
http://camelab.org

ABSTRACT

This paper discusses the main benefits of ZNS and shows why ZNS can be deprived of internal parallelism when downsizing its zone writable capacity. To this end, we use two production ZNS SSDs and quantitively analyze the performance degradation caused by inter-zone interference. We then suggest a simple mechanism to detect zone-to-zone relationships generating the interference and schedule I/O requests by being aware of internal parallelism. Our evaluation results using real production ZNS devices show that our mechanism can improve the bandwidth and latency of Linux's multi-queue I/O scheduler by 1.98× and 2.2×, respectively.

1 INTRODUCTION

Zoned namespaces (ZNS) are an emerging storage interface, which can match how data is written to the device-side backend flash while hiding the details of flash-specific management [1]. Specifically, ZNS offers a concept of zones, each being mapped to one or more flash physical blocks and exposing their operational constraints to a host (e.g., sequential writes in a block and erase-before-write). The main benefit of ZNS is to make the underlying solid-state drives (SSDs) cost-efficient. Since ZNS lets the host manage data over zones, SSDs can be free from the management of many restraints and get lighter. For example, ZNS SSDs do not require preserving lots of overprovisioned flash blocks for garbage collections [2] as the host is solely responsible for reclaiming zones and making a decision on the physical data layout. In addition, all writes for a zone should be performed in sequential, which makes block-to-block (B2B) mapping sufficient rather than page-to-page (P2P) mapping. This can also make SSDs cheaper by removing a large size of internal DRAMs used for covering TB-scale backend flash's address space.

However, there is no free lunch. When ZNS is applied, the host should manage the P2P mapping on behalf of SSDs and obey all the block-level constraints that flash exhibits in nature (sequential writes). In addition, it requires periodically resetting zones (i.e., block erases), which makes the host-side storage software stack a bit more complicated. To address these shortcomings, several studies propose advanced schemes and try to amend the ZNS interface by integrating new features. For example, [3] introduces an in-storage compaction mechanism, which can internally move data between different zones, thereby reducing the zone reclaiming overhead. On the other hand, a technical proposal [4] suggests a zone random write area (ZRWA) that allows overwriting a few data such that it can relax the host-side constraint in a particular case, such as metadata updates.

These technical suggestions can advance ZNS, but there is another important system parameter that we should NOT forget, *SSD's internal parallelism*. This paper mainly argues why the conventional software stack and operating systems need to take care of internal parallelism. Note that most SSDs aggregate many flash chips by interconnecting them through multiple flash I/O channels to overcome the wide performance disparity between what the host system bus requires (a few GBs) and what a low-level flash chip can support (several tens to hundreds of MB). The underlying flash firmware and/or controllers parallelize incoming I/O requests across the different channels and chips, satisfying the host's performance demands. While ZNS successfully abstracts the address space of SSD's backend and flash intrinsic characteristics using ZNS, there is a lack of abstraction to manage such internal parallelism at the host side. Specifically, we observe that if the host accesses ZNS SSDs with ignorance of their hardware layout, it degrades the bandwidth of the ZNS SSDs by 2.84×, on average, compared to an ideal case.

This paper reconsiders what the main benefits of ZNS bring (Section 2) and discusses why ZNS can be deprived of internal parallelism when downsizing its zone writable capacity (Section 3). In particular, we use two production ZNS SSDs and quantitively analyze the performance degradation caused by inter-zone interference and the challenges therein. We then suggest an interference profiler and zone-aware I/O scheduler, which can recognize zone-to-zone relationships generating such an interference and maximize the degree of internal parallelism, respectively (Section 4). Lastly, we show how much our simple suggestion can improve the device-level performance by being aware of the interference. Specifically, when we evaluate large-scale, data-intensive workloads on the production ZNS SSD, our scheme can improve the bandwidth

HotStorage '22, June 27–28, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9399-7/22/06…$15.00
https://doi.org/10.1145/3538643.3539744

and latency of Linux's multi-queue I/O scheduler by 1.98× and 2.2×, on average, respectively.

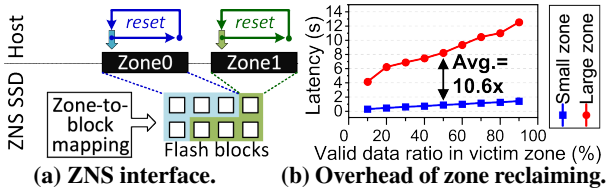
2 RETHINKING ZONED NAMESPACES

2.1 What are the Benefits of ZNS?

Recently, several studies report that ZNS is expected to deliver better performance compared to conventional storage interfaces because of its transparent design [2, 3, 5, 6]. For example, [5, 7] claims that, since the host is in a much better position to leverage application knowledge than the underlying flash firmware, it can do better application-aware data placement and perform near-optimal garbage collections (GCs). This can take GCs off the critical path such that ZNS is expected to increase performance predictability and reduce read tail latency. This can be true, but we argue a different side story of ZNS; we advocate ZNS from the cost-efficiency angle. SSDs are a complicated system, not a passive device like DRAM storage [8]. SSDs require page-to-page address translation and maintain all the mapping information in their device. This incurs higher monetary cost and scalability limitations, which hinders datacenters and enterprise servers from replacing HDDs with SSDs more aggressively. Note that the flash firmware and internal DRAMs take half the cost of SSDs or more than that [9, 10].

There have been many efforts to make SSDs cost-efficient. For example, Open-Channel SSD (OCSSD [11]) is the interface to expose the backend flash storage to the host via logical block chunks directly. This characteristic is similar to what ZNS wants to achieve, but there are two practical issues rendering OCSSD difficult to be widely adopted in various computing. First, the host side storage stack should manage all reliability and data consistency issues, which can significantly vary based on SSD vendors and their implementation. Second, the SSD's cost is not surprisingly reduced. This is because the hardware implementation for OCSSD still requires complex computing logic to expose all the flash natures to the host through a simple, thin block interface.

We believe ZNS is a much more practical interface (than OCSSD) to make storage cost-efficient. As shown in Figure 1a, ZNS defines a set of *zone*s, each being directly mapped to one or more physical flash blocks. In contrast to OCSSD, ZNS allows the underlying SSDs to have flash firmware, but it can be much lighter as the firmware only manages the mapping and device-level errors across different zones, not pages. From the host-side view, the address space of a ZNS SSD is exposed by a set of zones, each containing its own logical addresses, which can access just like conventional SSDs. However, as zones are an abstract of flash blocks, each zone's address space should be written in sequential and requires a reset (i.e., erase) before storing data. The host thus takes over all the heavy tasks of the existing flash firmware like GCs, caching, and page-to-page address translation. This clear functional



(a) ZNS interface. **(b) Overhead of zone reclaiming.**

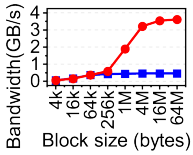
Figure 1: Zoned namespace.

boundary between the host and SSD can reduce the monetary cost paid for the large size of internal DRAMs and firmware.

2.2 Who Can Benefit from ZNS?

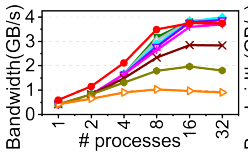
High-density flash. In addition to the cost efficiency mentioned above, ZNS can reduce the write amplification factor, making the underlying storage much more reliable as the host needs to perform writes in sequential and reclaim zones (i.e., GCs) explicitly. It is beneficial for storage vendors to build datacenter/enterprise SSDs using highest-density flash technologies, such as a triple-level cell (TLC) and quad-level cell (QLC). Since the highest-density flash can triple or quadruple the storage capacity, it can make SSDs better than HDDs in terms of the cost per performance. However, their low endurance and shorter lifetime characteristics can ironically increase the total cost of ownership (TCO) and challenge a wide adoption of such SSDs in various data-intensive computing. For example, QLC's number of program per erase (P/E) is 10× lower than MLC [12]. ZNS can address this shortcoming as it allows the host to reclaim zones directly aware of application knowledge and only permits sequential writes at the device level. This property significantly improves the flash block utilization without amplifying writes, thereby reducing P/Es and making the high-density flash practical in many data-intensive computing domains. Note that, even though writes should be performed in sequential, there is no constraint on read services; arbitrary addresses in a zone can be read. Many read-intensive applications can enjoy the low-cost, high-performance capabilities that ZNS support.

Reads are all that matter. A specific use case scenario which takes advantage of ZNS SSDs is large-scale recommendation systems [13, 14], distributed key-value store (KVS) for AI/ML services [15], and social graph data management in datacenters [16]. For example, the recommendation systems of Facebook and Baidu keep user's TB∼PB-scale embedding tables into enterprise SSDs (rather than HDDs) and read feature vectors of the embedding tables in a random manner [15]. Facebook also maintains social graph data in MySQL tables, and all the corresponding table rows are stored and serviced from KVS (RocksDB). Note that the recommendation system and RocksDB scenarios sporadically update large-scale user data (e.g., profiles and graphs) in a mostly sequential manner while intensively reading the data for training and inference of several AI/ML services or graph analysis. We believe that their high demands of massive storage capacity and excellent read performance make ZNS and the high-density flash technologies promising in such datacenter scenarios.



(a) Sequential. (b) Random.

Figure 2: Different block size.



(a) Overall. (b) Speedup.

Figure 3: Different # of processes.

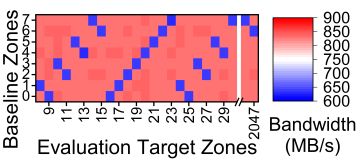


Figure 4: Example of profiling.

2.3 How Should Zones be Configured?

Thanks to ZNS's zone abstraction, the host does not have a restriction on read accesses and can enjoy the true read performance that the low-level flash devices expose. While the write performance may not be as crucial as reads in the aforementioned scenarios, the quality of read services (including the tail latency) can severely degrade due to periodic zone reclaiming on a write. ZNS SSDs translate a zone to many flash blocks spanning across 16∼128 flash chips, which exhibits a few GB writable capacity per zone. This *large zone* configuration is typical [2], and it can erase multiple blocks in parallel, thereby exhibiting high performance. However, it can unfortunately make the latency of zone reclaiming longer, which blocks following read services for a while. Since a large zone can contain many valid pages existing across different flash blocks, it obviously exhibits a longer time to keep the valid data in a safe space (e.g., page migration) before resetting the target zone. To address this, it is recently discussed to introduce a smaller writable capacity to a zone [3, 17].

In this paper, we also advocate configuring the zones as small as possible, called *small zone*. It is apparent from zone downsizing that we can address the long latency issue of zone reclaiming, but to be precise, we evaluate two production ZNS SSDs, each configuring the address space as a large zone (2.18GB) and small zone (96MB). Figure 1b analyzes the read latency of those two ZNS SSDs postponed by zone reclaiming. In this evaluation, we vary the ratio of valid pages per zone from 10% to 90% for both large and small zones. The specific system parameters and configurations for our ZNS evaluations will be explained in Section 5. As the ratio of valid pages increases, the read latency (blocked by zone reclaiming) of both ZNS SSDs increases. Nevertheless, the read latency of the ZNS SSD configured by small zones is shorter than that of large zones by 10.6×, on average. Note that, considering a read of the ZNS SSDs takes 480 us, we believe that longer than a ten seconds delay with large zones is not acceptable in many read-intensive computing scenarios.

3 ABSTRACTION GRAY AREA

While ZNS is an excellent interface to draw the clear boundary between the host and flash firmware through zones/blocks, there is a lack of hardware abstraction to exploit SSD's internal parallelism explicitly from the host. In this section, we first examine what the host should consider using the small zone ZNS, and then, analyze the performance degradation coming from the lack of ZNS's hardware abstraction.

3.1 Parallelism Loss and Challenges

There are several studies to urge ZNS to support the small zone [2, 3], but its real implementation can lose the parallelism that large zones bring, thereby degrading performance seriously. Figure 2 compares sequential and random read bandwidth of the two production ZNS SSDs that we tested. These two SSDs, called *ZNS-small* and *ZNS-large*, use the same version of flash firmware controller and the same technology of backend storage (less than 32TB, TLC-based high-density flash). Each flash die of the backend storage contains four planes, operating altogether in parallel. The only difference is the ZNS configuration. ZNS-large configures 2.18GB for each zone and supports 12 open zones (allowing 12 current writes), while ZNS-small uses 96MB with 4096 open zones. All the ZNS SSDs are connected to a host with a 2.3GHz Intel Xeon 20 core CPU (offering 40 vCPUs) through PCIe 3.0×4 (3.94GB/s max). The access patterns are generated by FIO [18] and ZoneFS [19].

Performance degradation. One can be observed from the figure that the bandwidth of ZNS-large gets better as the request size of reads increases. When we increase the block size by 16MB, it reaches the maximum bandwidth of PCIe and gets saturated. The reason why it exhibits better performance with a longer request length is SSD's internal parallelism [20–25]. Since a large request can be split into multiple sub-requests, it can be striped across different internal resources (associated with one or more zones) of ZNS-large and served in parallel (e.g., flash channel and chips). Other conventional types of SSDs also observe this phenomenon since the underlying firmware controls all the internal parallelism in cases where the request size is sufficient to span over all the internal resources.

While it is vital to secure a high degree of internal parallelism for better performance, ZNS-small cannot. As shown in the figure, ZNS-small's bandwidth is limited by 460 MB/s even with the large requests and 7.85× worse than ZNS-large. This is because, when it configures the zone writable capacity as 96MB, the zone loses all the parallelism therein. Specifically, the size of a ZNS-small's physical flash block for each plane is 24MB, and thus, a zone (96MB) is solely accommodated by a single flash chip. This makes all pages of a zone located in a flash chip, which in turn can make the request unable to be striped across different chips.

Note that, as shown in Figures 2a and 2b, both ZNS-large and ZNS-small show the similar performance trend between sequential and random reads. The reason why there are some

production SSDs exhibiting better performance on sequential is to prefetch requests using their internal DRAMs. Unlike these SSDs, ZNS SSDs are designed towards minimizing their internal hardware resources. Thus, it does not apply the prefetch, which makes the read performance trend entirely depend on low-level flash latency and their parallelism. For example, the production ZNS SSDs, only employ internal DRAMs accounting for 0.05∼0.1% of the storage capacity, which is 10∼20× smaller than the conventional SSD's internal DRAMs.

Parallelism improvement limits. One of the ways for ZNS-small to catch up with the performance of ZNS-large is increasing the number of application processes at the host side. Specifically, the host can allocate different zones across the processes, making each process simultaneously operate on its dedicated zone. Even though it gives more burdens to the host to manage multi-tenant environment (e.g., process and zone management), the host can activate many zones in an attempt to improve ZNS-small's internal parallelism. However, since the host does not have specific information on the underlying SSD's hardware, it can allocate zones interfering each other (because of flash channel/die sharing) to different processes thereby limiting the degree of the internal parallelism. For example, if different requests associated with two different zones should be served from one or two resources in a channel, their performance can significantly degrade because of flash-level conflicts and channel contention [20–25].

Figure 3a shows the read performance comparison between ZNS-large and ZNS-small operating with many application processes. One can be observed from the figure that ZNS-small's performance significantly varies based on the level of inter-zone interference; we classify the interference level (*IL*) ranging from 0 to 7 according to their different bandwidth behaviors. Note that, we only show the result of random reads in this evaluation since our production ZNS SSDs exhibit the same performance trend for both reads and writes. As shown in the figure, ZNS-small with IL 7 exhibits 809 MB/s, on average, while ZNS-small's bandwidth with 32 processes reaches the maximum bandwidth (3.94 GB/s) that its PCIe interface delivers if there is no interference (IL 0). As reducing the interference level, ZNS-small is expected to improve the degree of SSD's internal parallelism thereby increasing the bandwidth significantly.

Figure 3b summarizes the performance improvement of ZNS-small based on varying levels of the interference. ZNS-small with IL0 is 2.84× better than that with IL 7. Even in cases where one can reduce the interference level at some extents, not completely, the performance gain is significant. For example, if the host can allocate zones to different processes wisely by managing the interference level from 7 to 4, it improves ZNS-small's bandwidth by 1.33GB/s, on average. However, the host cannot unfortunately control the interference level as *there is no interface abstraction in ZNS, exposing the internal hardware configuration to the host*.

4 EXPLOITING PARALLELISM

The aforementioned shows that, if one can add simple features to figure out the interference level or hardware layout into ZNS, the host can control ZNS-small without losing parallelism while significantly reducing the zone reclaiming latency (cf. Section 2.3). While revising the interface would be the best to remove the inter-zone interference, in this paper, we show a simple, effective method that improves ZNS SSDs' bandwidth by exploiting the internal parallelism. This method consists of an interference profiler and a zone-aware scheduler. Our profiler detects per-zone performance degradation and classifies zones into multiple *conflict groups* (CGs), each containing the zones interfering with each other. Then, the zone-aware scheduler does issue I/O requests of multiple CGs in an evenly distributed manner.

4.1 Inter-Zone Interference Detection

Figure 4 shows how we can detect and profile the interference among the standard ZNS's zones. In this example, we show a set of baseline zones (0∼7) to compare in the y-axis, while the x-axis shows different zone indices that we evaluate in parallel with the corresponding baseline zone. For brevity, the figure configures the baseline zones by selecting the zone having completely no interference with each other. One can observe from this figure that each comparison item exhibits identically different performance values; 600MB/s (blue) vs. 900MB/s (red). The blue item means there is interference between the baseline line and target zones. For example, the zones 9, 16, and 25 exhibit the interference with the zone 0 (e.g., the bottom row of the figure). Note that the zone 9 is not interfered by the zones 1∼7, but only interfered by the zone 0. Thus, we can collect all the zones highlighted by blue for each row and call them a CG.

Even though the high-level concept for profiling CGs is straightforward and simple, its actual method is practically a bit more complicated. This is because, as there is no information regarding the zone-level hardware information at all, the host also has no knowledge about the baseline zones. In addition, the baseline zones to compose their CGs can vary based on how the host initially assigns running processes to the different zones. For example, Figure 5 illustrates all the CGs made for the entire zone space of our ZNS-small. In cases where there are 16 co-running processes, Figure 5a assigns the zones to each process in a sequential manner, whereas Figure 5b allocates the zones in a wrap around manner (e.g., a process has zones 0, 16, 32, ..., 2047). The actual baseline zones of Figure 5a and Figure 5b are {0, 1, 2, 3, 4, 5, 6, 7} and {0, 1, 4, 5, 8, 9, 12, 13}, respectively. Because of the dynamics, we need to compare all zones appropriately. Algorithm 1 explains how to classify zones into different CGs. At the very beginning, the algorithm initializes the baseline zone of CG0 with the zone 0 (line 1). It also requires setting a performance threshold to determine whether the visiting

Algorithm 1: Interference detection.

Input: N := Number of zones
Output: CG := an array of conflic groups

```
1  CG[0][0] = zone_0 // Set a init baseline zone
2  for k ← 1 to N − 1 do // For all zones
3      bw.append(bandwidth(zone_k, CG[0][0]))
4  threshold = average(bw.max, bw.min)
5  for k ← 1 to N − 1 do // For all zones
6      newCG = 1
7      for l ← 0 to len(CG)-1 do // For all CGs
8          if bandwidth(zone_k, CG[l][0]) < threshold
9              newCG = 0
10             CG[l].append(zone_k)
11             break
12     if newCG
13         CG.append([]) // Add new CG
14         CG[len(CG)][0]=zone_k // Set new baseline zone
```

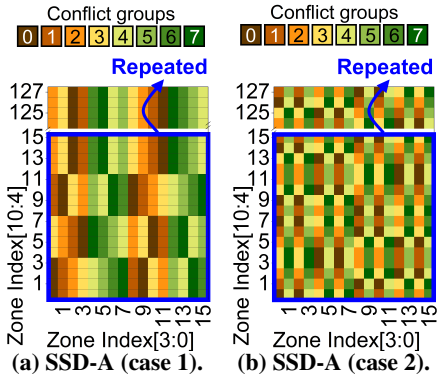


Figure 5: Result of Zone-to-CG.

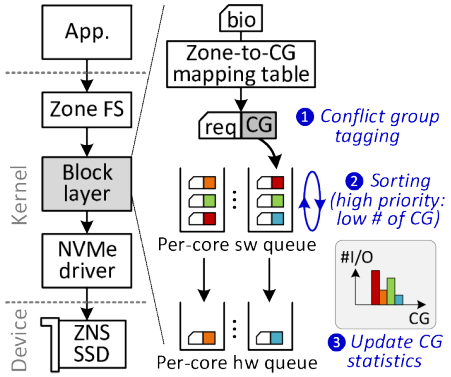


Figure 6: Interference-aware scheduler.

zones interfere with a baseline zone or not. The threshold can be simply achieved by calculating the average mean of two zones each exhibiting high and low performance (lines 2 ∼ 4). It then visits all zones (line 5) and evaluates whether each zone is associated with CG's baseline zone or not (line 8). During the evaluation, the host can issue a few requests for both the visiting zone (i.e., $zone_k$) and CG's baseline zone (i.e., CG[l][0]) in parallel. If the zone's bandwidth is lower than the threshold, it adds this zone into the target CG (CG[l]). Otherwise, it creates a new CG and sets the baseline zone with the visiting zone (line 12 ∼ 14). Once we visit all zones, this algorithm turns out a *zone-to-CG* (Z2C) mapping table, which can be used for a runtime I/O scheduler.

4.2 Interference-aware I/O Scheduling

The main goal of the interference-aware scheduler is to schedule I/O requests coming from different CGs, not the same CG as many as possible. Figure 6 shows an example of the runtime I/O scheduler that increases SSD's internal parallelism using the Z2C mapping table. Since block I/O requests (`bios`) have no information regarding zones, the block layer (e.g., `blk_mq`) checks the logical block address (LBA) of an incoming `bio` to figure out which zone owns it. Based on the retrieved zone index, the block layer can find the corresponding CG by referring to the Z2C mapping table and tags the CG index to the target `bio`. After this CG tagging (❶), the scheduler checks how many requests have been issued for each CG and gives the highest priority for the CG having lower numbers of outstanding requests to be scheduled (❷). This can help to fairly schedule the I/O requests across different CGs by considering their load balance while exploiting the internal parallelism. Obviously, the block layer needs to keep the number of outstanding requests per CG (❸), which will be referred in its interference-aware scheduling.

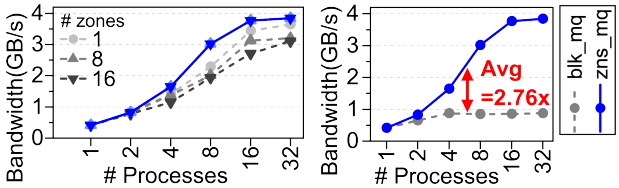
5 SCHEDULING IMPACT

In this section, we focus on evaluating the performance impact on realistic workloads that may have benefit from ZNS. Specifically, we setup the evaluation environment by mimicking two enterprise-scale application scenarios, RocksDB and recommendation systems. Using the testbed with ZNS-small (cf. Section 3.1), we co-run 1∼32 processes, each having 200∼256 zones. RocksDB allocates each SST file (sorted string table) to varying numbers of zones, ranging from 1 to 16. For the recommendation system, 128 zones are allocated to manage an embedding table containing 50 million indices with 64 dimension (12GB) [26]. We prepare two schedulers, a multi-queue block IO queueing (`blk_mq`) and a zone interference-aware multi-queue (`zns_mq`) that uses our interference profiling information.

5.1 Bandwidth Impact

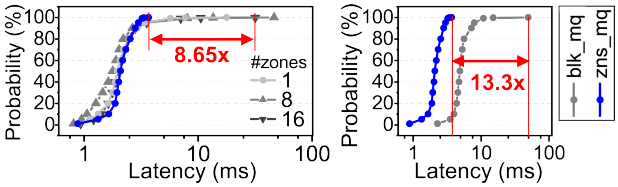
Figure 7 compares the performance of `blk_mq` and `zns_mq` for RocksDB. Since the typical size of each SST file is several tens to hundreds MB, RocksDB has an excellent position for multiple applications to exploit small zones. As the number of application processes increases, the bandwidth for both `blk_mq` and `zns_mq` increases. However, one can observe from this figure that `blk_mq` degrades the bandwidth as increasing the size of SST files (16%, on average). This is because each process has zones sitting on a contiguous address space, which increases the number of zones being classified by the same CG. When all co-running processes access such zones, the service is delayed because of their inter-zone interference. In contrast, `zns_mq` has no performance drop and shows very sustainable performance irrespective of the number of zones allocated for each SST file. Even though there are many zones associated with the same CG, `zns_mq` schedules I/O requests of multiple processes being aware of their inter-zone interference and distributes the requests across all the conflict groups in the balanced manner. Thus, `zns_mq` can improve the bandwidth of `blk_mq` as high as 55%.

As shown in Figure 7b, this performance trend becomes more identical when we run multiple training tasks on large-size embedding tables. The bandwidth of `blk_mq` is limited and saturates at 878MB/s even though we increase the number of processes training the embeddings in parallel. This is because the host does not have any knowledge of SSD's internal parallelism, and `blk_mq` serves incoming `bios` in the first come, first served manner. Thus, while there are 40 per-core



(a) RocksDB. (b) Recommendation system.

Figure 7: Bandwidth analysis.



(a) RocksDB. (b) Recommendation system.

Figure 8: Tail latency analysis (32 processes).

queues, `bios` heading to the same target CG suffer from a low degree of internal parallelism. As shown in the figure, `zns_mq` performance trend for this recommendation system is similar to what we observed from RocksDB, which makes `zns_mq` better than `blk_mq` by 2.76×, on average.

5.2 Latency Impact

Figure 8a shows cumulative distribution function (CDF) of RocksDB latency when we co-run 32 processes, and allocate 1, 8, and 16 zones for each SST file. `zns_mq` achieves narrow width of distribution, while `blk_mq` experiences wider distribution in the tails. This is because there are huge interference level differences between I/O requests in `blk_mq`, whereas `zns_mq` can guarantee all I/O requests experiencing similar interference levels at any time by considering conflict groups of the outstanding I/O requests. Specifically, when we allocate 16 zones for each SST file, `blk_mq` exhibits 5.7% shorter average latency than `zns_mq`, but its three nine (99.9th percentile) long-tail latency is worse than `zns_mq` by 8.65×. Note that, the number of inter-zone interferences for `blk_mq` increases as the number of SST files for each zone increases. However, the impact of interference appears to become saturated from the eight zones (cf. Figure 7a).

As shown in Figure 8b, recommendation system introduces similar inter-zone interference impacts as RocksDB. Specifically, `zns_mq` can mitigate three nine long-tail latency as much as 92% for the recommendation system, which is the same amount as the RocksDB. However, unlike RocksDB, `zns_mq` for the recommendation system can reduce not only long-tail latency but also average latency by 70% due to the high interference level (e.g., IL 7) when we allocate 128 zones for each embedding.

6 RELATED WORK AND DISCUSSION

Related work. [2] argues that ZNS is better than the conventional block interfaces since it can reduce all the overhead imposed by the host-side and device-side storage stack. It shows how RocksDB can take advantage of ZNS. [3] moves further and improves a ZNS-enabled system's performance by utilizing flash-level data copy operations. These studies advocate that a small zone configuration is better than a large zone configuration as large zones can reduce the degree of freedom for host-level data placement. However, these studies did not consider SSD's internal parallelism that should be taken into account. [27] improves garbage collection overhead being aware of SSD's internal parallelism, but it is related to ZNS using large zones. In contrast, this work focuses on analyzing challenges of ZNS using small zones and studies a simple, but efficient scheduling mechanism to improve the parallelism being aware of inter-zone interference.

Limits and future work. The proposed host-side profiler can detect the interference inherited from the underlying hardware sources and classify zones based on their zone-to-zone relationships. However, the inter-zone interference and zone-to-zone relationships can vary based on how we allocate zones to different processes. Thus, the proposed software-based profiling may bring undesirable overhead when there is a change for the zone allocations. The current prototype exhibits a few milliseconds to replace the existing interference information with new one (if there are many zone allocation changes). We believe that revising ZNS interface itself to expose the interference information for each zone is better for the host to take SSD's internal parallelism. As an alternative option, we can interleave interference profiling with process scheduling and/or namespace allocations or schedule it in idle. As our technique to exploit the parallelism is simple to implement, we believe that it can be easily integrated into existing and future ZNS studies [2–5, 17, 27].

7 CONCLUSION

We use two production ZNS SSDs and quantitively analyze the performance degradation caused by inter-zone interference. In this paper, we also suggest a simple mechanism to detect zone-to-zone relationships and schedule I/O requests by being aware of internal parallelism. Our evaluation results show that our mechanism can improve the bandwidth and latency of Linux's multi-queue I/O scheduler by 1.98× and 2.2×, on average, respectively.

ACKNOWLEDGEMENT

The author thanks to Janki Bhimani for shepherding this paper. The author also thanks to anonymous reviewers for their constructive feedback. This work is mainly supported by S3RC Hynix Center and SK-Hynix (G01200477). This work is also in part supported by NRF's 2021R1A2C4001773, IITP's 2021-0-00524 & 2022-0-00117, KAIST start-up package (G01190015), and KAIST IDEC. Myoungsoo Jung is the corresponding author.

REFERENCES

[1] NVM Express. *Zoned Namespace Command Set Specification*. Rev. 1.1.

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

[3] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 147–162, 2021.

[4] Matias Bjørling. Zoned namespaces (ZNS) SSDs: Disrupting the storage insdustry. Technical report, 2020.

[5] Theano Stavrinos, Daniel S Berger, Ethan Katz-Bassett, and Wyatt Lloyd. Don't be a blockhead: zoned namespaces make work on conventional ssds obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 144–151, 2021.

[6] Umesh Maheshwari. From blocks to rocks: A natural extension of zoned namespaces. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pages 21–27, 2021.

[7] Western Digital. ZenFS, Zones and RocksDB - Who Likes to Take out the Garbage Anyway? Technical report.

[8] Lorenzo Zuolo, Cristian Zambelli, Rino Micheloni, and Piero Olivo. Ssdexplorer: A virtual platform for ssd simulations. In *Solid-State-Drives (SSDs) Modeling*, pages 41–65. Springer, 2017.

[9] Myoungsoo Jung, Wonil Choi, John Shalf, and Mahmut Taylan Kandemir. Triple-a: A non-ssd based autonomic all-flash array for high performance storage systems. *ACM SIGARCH Computer Architecture News*, 42(1):441–454, 2014.

[10] Nand flash spot price. Technical report, 2014.

[11] LightNVM. *Open-Channel Solid State Drives Specification*. Rev. 2.0.

[12] The advantages and disadvantages of nand flash system. Technical report, 2021.

[13] Hu Wan, Xuan Sun, Yufei Cui, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. Flashembedding: storing embedding tables in ssd for large-scale recommender systems. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 9–16, 2021.

[14] Udit Gupta, Samuel Hsia, Jeff Zhang, Mark Wilkening, Javin Pombra, Hsien-Hsin Sean Lee, Gu-Yeon Wei, Carole-Jean Wu, and David Brooks. Recpipe: Co-designing models and hardware to jointly optimize recommendation quality and performance. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 870–884, 2021.

[15] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems*, 2:412–428, 2020.

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[17] Youngjae Lee, Jeeyoon Jung, and Dongkun Shin. Buffered i/o support for zoned namespace ssd. In *2021 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pages 1–4, 2021.

[18] Jens Axboe. fio.

[19] ZoneFS - Zone filesystem for Zoned block devices.

[20] Myoungsoo Jung, Ellis H Wilson III, and Mahmut Kandemir. Physically addressed queueing (paq) improving parallelism in solid state disks. *ACM SIGARCH Computer Architecture News*, 40(3):404–415, 2012.

[21] Myoungsoo Jung and Mahmut T Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *HotStorage*, 2012.

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

[23] Myoungsoo Jung and Mahmut T Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 524–535. IEEE, 2014.

[24] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T Kandemir. Hios: A host interface i/o scheduler for solid state disks. *ACM SIGARCH Computer Architecture News*, 42(3):289–300, 2014.

[25] Congming Gao, Liang Shi, Mengying Zhao, Chun Jason Xue, Kaijie Wu, and Edwin H-M Sha. Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2014.

[26] Hui Guan, Andrey Malevich, Jiyan Yang, Jongsoo Park, and Hector Yuen. Post-training 4-bit quantization on embedding tables. *arXiv preprint arXiv:1911.02079*, 2019.

[27] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. A new LSM-style garbage collection scheme for ZNS SSDs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.