Achieving Performance Isolation in Docker Environments with ZNS SSDs

Yejin Han*, Myunghoon Oh*, Jaedong Lee*, Seehwan Yoo*, Bryan S. Kim†, and Jongmoo Choi*
*Dankook University, Yongin-si, Gyeonggi-do, South Korea
†Syracuse University, Syracuse, New York, United States
{yj0225, snt2427, letsdoit, seehwan.yoo, choijm}@dankook.ac.kr, bkim01@syr.edu

2023 IEEE 12th Non-Volatile Memory Systems and Applications Symposium (NVMSA) | 979-8-3503-4496-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/NVMSA58981.2023.00016

***Abstract*—Checkpointing is an essential ingredient in Docker environments for fast recovery and migration. However, it incurs serious interference with normal I/Os and other Docker containers running in parallel. To address this problem, we propose a new checkpoint scheme based on Zoned Namespace Solid State Drives (ZNS SSDs), specifically having small zones. ZNS SSDs are a novel type of SSDs that support a zone interface, allowing workload separation and performance isolation. On small zone ZNS SSDs, we devise two techniques: 1) independent zone allocation and 2) striping across multiple zones to achieve both interference mitigation and Quality of Service (QoS) support. Real implementation-based experiments show that our proposal can reduce interference by up to 40% when multiple dockers perform checkpoints concurrently.**

***Index Terms*—Checkpointing, Docker, Performance interference, ZNS SSDs**

I. Introduction

Docker is a software platform that offers a variety of benefits, including lightweight virtualization, low-performance overhead, and fast deployment. In Docker environments, checkpointing is actively used to provide fast recovery from faults and to enable migration for load balancing [1]–[3]. However, because checkpointing is a heavy I/O job, it interferes with the performance of other containers that share storage.

This paper explores how to exploit *small zone* ZNS SSDs to mitigate performance interference in Docker environments. ZNS SSDs are recently emerging SSDs that support a zone interface to provide the advantage of reduced Write Amplification Factor (WAF) through workload separation and to support host controllability [4], [5]. In particular, small zone ZNS SSDs give a foundation for reducing performance interference since there exist zones that are independent of each other by mapping them into different channels/ways in SSDs [6]–[8].

We first compare the performance of a real small zone ZNS SSD device to a traditional SSD device. Experimental results show that performance interference is noticeable as the number of threads increases in traditional SSDs, while in small zone ZNS SSDs, this interference is eliminated by allocating each thread to an independent zone. However, our experiments also demonstrate that there is performance degradation when using small zone ZNS SSDs due to the limited parallelism in a zone.

In order to overcome the performance degradation problem of small zone ZNS SSDs while obtaining the performance interference benefit, we propose two techniques: (1) independent zone allocation to each Docker container for preventing interference and (2) striping across multiple zones for improving performance. We identify independent zones and allocate them to different containers for improving predictability. In addition, a container can specify its required bandwidth, and we allocate the appropriate number of zones and distribute writes in an interleaved manner to satisfy the requirement.

We implement our techniques by modifying the Linux software tool CRIU (Checkpoint and Restore In Userspace) [9]. The modified CRIU provides not only two techniques but also other facilities such as ZNS SSD controls (e.g. zone reset) and Docker interfaces (e.g. required bandwidth). Evaluation results show that our independent zone allocation technique can prevent normal I/Os from being interfered by checkpointing. It also mitigates interference among multiple checkpoints conducted in parallel. Besides, the striping technique makes container checkpointing performance in ZNS SSDs comparable to traditional SSDs.

The contributions of this paper are as follows.

- To the best of our knowledge, this is the first study that utilizes small zone ZNS SSDs for reducing performance interference in Docker environments.
- We devise two techniques, independent zone allocation and striping on multiple zones, to reap the benefit of performance isolation of small zone ZNS SSDs.
- We present various evaluation results including characteristics of ZNS/traditional SSDs and the effectiveness of two techniques using real devices.

II. Background

A. Zoned Namespace (ZNS) SSDs

SSDs adopt flash memory as storage media. Due to the idiosyncrasies of flash memory such as erase-before-write and limited endurance, SSDs make use of a software layer, called Flash Translation Layer (FTL), to provide out-of-place updates, garbage collection, and wear-leveling. In addition, to improve performance, modern SSDs utilize internal parallel units such as multiple channels, ways, planes, and so on.

ZNS SSDs [10], one of the most recently proposed SSDs, have two characteristics: 1) they divide the entire logical block address (LBA) into zones with sequential write constraints and provide a new zone interface, and 2) a host has complete control over data stored in zones [4], [6], [7], [11]–[15]. ZNS SSDs can provide several benefits over traditional SSDs that

2575-257X/23/$31.00 ©2023 IEEE
DOI 10.1109/NVMSA58981.2023.00016

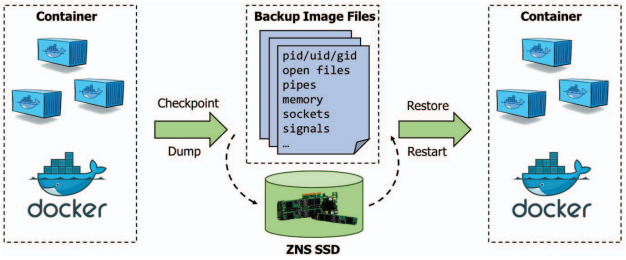

Fig. 1. **Docker container checkpoint and restore**

only support the conventional block interface. By separating different workloads into different zones, ZNS SSDs can reduce WAF, giving a positive impact on performance and reliability. Host-level management can also reduce required FTL functionalities, eventually lowering SSD cost by decreasing device-level DRAM and over-provisioning area.

ZNS SSDs are classified into two categories: *large zone* and *small zone* ZNS SSDs. Large zone ZNS SSDs support a larger zone size (e.g., 1GB), which is mapped into multiple channels/ways [4], [5]. In contrast, small zone ZNS SSDs provide a smaller zone size (e.g., 72MB), and zones are commonly mapped into a single channel/way [6]–[8]. Our interest is to mitigate performance interference, so this paper mainly focuses on small zone ZNS SSDs.

B. Docker container checkpoint

Container-based virtualization, such as Docker, is widely employed in cloud environments. Containers enable rapid application deployment with lower overhead compared to virtual machines, making them an efficient solution to manage software services in the cloud. To effectively manage the container lifecycle, checkpoint and restore (C/R) plays an important role in a virtual environment with a large number of containers [16].

In large-scale systems, failure is the norm, and fault recovery is an indispensable demand. Moreover, cloud providers often need to migrate containers to different hosts for load balancing and resource optimization. For these tasks, checkpointing is an essential method that can minimize downtime and prevent data loss. Additionally, it is popularly used for system maintenance and infrastructure upgrades.

Docker integrates the Linux software tool CRIU (Checkpoint and Restore In Userspace) for container checkpoint and restore. Figure 1 illustrates the overall process. During the checkpoint process, each container's status is stored as image files to the storage. This status includes pid/uid/gid, open files, pipes, memory pages, network sockets, signals, and so on. After checkpointing, we can restart the container at the same host (fault recovery case) or at another host (load balancing case) by reading the saved image files from the storage.

One concern with checkpointing is that it can interfere with normal I/Os and other Docker containers. This is because checkpointing involves saving a container's memory and process states to disk, which generates a significant amount of

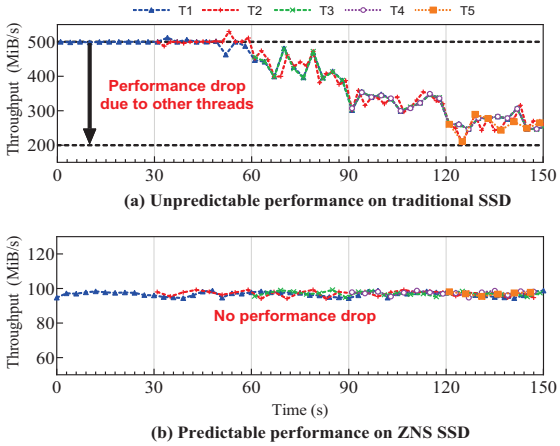

Fig. 2. **Performance characteristics of traditional SSDs and ZNS SSDs**

disk I/Os and can cause reduced performance for other I/O-intensive containers that share the same hardware resources.

Thus, performance isolation for checkpoints is crucial to ensure the stability of Docker environments.

III. Motivation

Traditional SSDs have commonly been used as storage for checkpoints. However, they lack support for workload isolation per Docker. This can lead to interference between different Dockers, potentially causing unexpected delays in cloud environments that require guaranteed QoS. On the other hand, ZNS SSDs have the potential to support such requirements. To explore these performance characteristics, we conduct experiments using both a traditional SSD and a ZNS SSD device.

We set up an experimental environment using two real SSD devices, traditional SSD and ZNS SSD prototype, acquired from an SSD vendor. Both devices have the same internal architecture, 8 channels and 2 ways, and provide the same capacity, 2TB. But, they differ in that the traditional SSD device provides the conventional block interface while the ZNS SSD device provides the ZNS interface. The zone size of the ZNS SSD device is 72MB, which is mapped into a single channel/way. The total number of zones is 29,172.

Figure 2 presents a performance comparison between traditional SSD and ZNS SSD using the FIO benchmark [17]. In this experiment, we make five threads that write data sequentially to both SSDs. Each thread starts at different times, 0, 30, 60, 90, and 120 seconds, respectively. We throttle the write performance of each thread to 500MiB/s to observe under a more realistic cloud scenario [18], [19]. Many cloud service providers employ throttling to limit the maximum throughput, guaranteeing controlled performance across multiple workloads and preventing a single workload from monopolizing shared resources.

From Figure 2, we can make the following two observations:

Observation 1: *Traditional SSDs experience a noticeable performance drop by up to 60% due to interference by other threads, whereas ZNS SSDs do not exhibit such performance degradation.* As shown in Figure 2(a), the write throughput of traditional SSD decreases significantly from 500MiB/s to 200MiB/s as other threads run in parallel. This performance drop is because traditional SSDs cannot prevent interference from other users competing for the same channels/ways. In contrast, Figure 2(b) shows ZNS SSDs can support performance isolation, maintaining stable performance even when five threads access the same device concurrently. This is because each thread accesses an independent zone, which is mapped into a different channel/way in ZNS SSDs.

Observation 2: *Achieving performance isolation using ZNS SSDs comes with the trade-off of considerably lower throughput.* Even though ZNS SSDs can support performance isolation under concurrent jobs, this can result in quite low performance. As shown in Figure 2(b), ZNS SSDs show roughly 100MiB/s write bandwidth, which is significantly lower than traditional SSDs. This is because traditional SSD can utilize all internal channels/ways, while ZNS SSDs can only leverage one channel/way that is mapped into a zone accessed.

Note that the results of ZNS SSDs, shown in Figure 2(b), come from *small zone* ZNS SSDs in specific. We have performed the same experiment with a large zone ZNS SSD device and have observed similar trends as traditional SSDs, achieving better throughput at the cost of interference. This similarity comes from the fact that a zone in large zone ZNS SSDs is mapped into multiple channels/ways, becoming shared resources when we access multiple zones in parallel, just like traditional SSDs.

These observations reveal that traditional SSDs or large zone ZNS SSDs lack performance isolation, while small zone ZNS SSDs provide a predictable foundation. However, small zone ZNS SSDs raise a new question about how to overcome performance degradation due to the limited parallelism in a zone. Our proposal tries to obtain both benefits, performance isolation and throughput comparable to traditional SSDs.

IV. Design and Implementation

In this section, we discuss two proposed techniques, independent zone allocation and striping across multiple zones.

A. Independent zone allocation

Figure 3 illustrates the internal architecture of small zone ZNS SSD device used in this study. It consists of 8 channels and 4 ways, and a zone is mapped into a single channel/way. We define zones are *independent* if there are no performance drops when they are accessed at the same time. We observe that, in this device, zone 0 is independent to zone 1, as shown in Figure 2(b). In general, zones that do not share a way are independent (e.g. zone 0 is independent to zone 1∼7, but not to zone 32). We also discover that zones in different ways but in the same channel (e.g. zone 0 and zone 8) are independent. This is because modern SSDs are equipped with fast channels and serve ways in a pipelined manner so that ways in the same channel can be accessed without a noticeable performance drop.

We design a technique that assigns independent zones to different containers so that there is no interference among them. For this technique, we need to address the following two issues. First, how to identify independent zones? The information about independent zones can be given by vendors or can be probed by comparing bandwidth [7] or latency [8].

In this study, we employ the probing approach and find that the latency between dependent zones is 2 times slower than that between independent zones. Based on this observation, we identify that there are 32 independently operable zone groups in our device (e.g. group 0 consists of zone, 0, 32, 64, ... and group 1 consists of zone, 1, 33, 65, ...). When we allocate zones from different independent groups to different containers, they do not interfere with each other.

The second issue is how to handle a situation when the number of independent groups is insufficient. There are two feasible options. One is employing an admission control to support independence strictly. In other words, when there is no unallocated group, an allocation request of a new container is delayed until other containers release their already-allocated groups.

An alternative option is relaxing independence and allocating a group with the smallest used containers. Note that there are multiple zones in a group so even though one zone has already been allocated to a container, another zone in the group can still be allocated to a new container. Since the latter option can accept all requests while making the best effort to minimize interference, we choose it in this study and leave the first option as future work.

B. Striping across multiple zones

Some containers require not only predictable but also higher performance beyond the capability that a zone can support. To support this requirement, we design our second technique, allocating multiple independent zones and striping data across them. In traditional SSDs, this work is conducted by FTL inside SSDs, which is transparent to a host. However, in ZNS SSDs, this responsibility is transferred to a host. We carefully argue that this kind of software-level explicit parallelism is indispensable in ZNS SSDs.

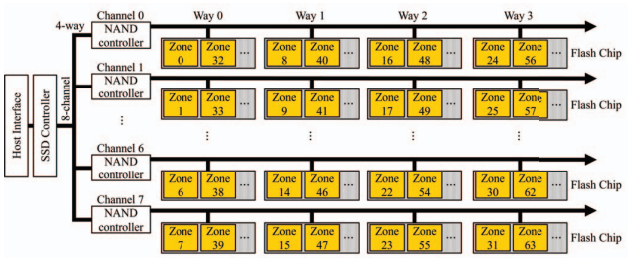

Fig. 3. **ZNS SSD internal architecture (8 channels, 4 ways, and a zone is mapped in a single channel and way)**

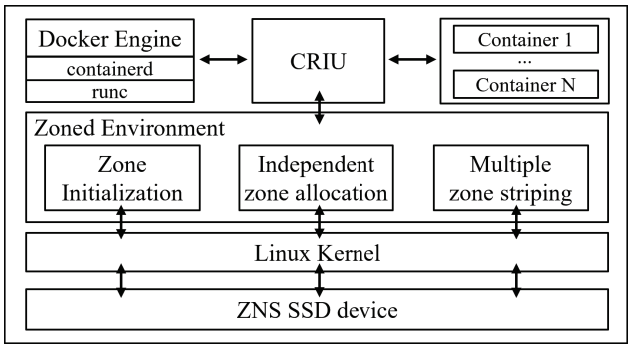

Fig. 4. **ZNS-CRIU overall structure**

For this technique, we have to tackle a question about how many zones need to be allocated to satisfy a container's QoS. We consider both the zone capability and container requirement at the same time. For instance, suppose a container requests a maximum bandwidth of 500MiB/s. In this case, we allocate five independent zones to the container because one zone can support roughly 100MiB/s in our device, as shown in Figure 2(b). Then, the proposed striping technique distributes data across allocated zones in an interleaved manner, which enables us to overcome the limited performance of a zone in small zone ZNS SSDs.

Our striping technique makes the mapping between zones and containers complicated. It converts the mapping from 1-to-1 to 1-to-N, where N is the number of zones allocated to a container. In addition, a checkpointed image consists of several image files such as process identifier, open files, memory context, and so on, as shown in Figure 1. We design a tree-style mapping structure where a container locates in the root node, image files at the next level, and zones at the bottom level. It allows for fast lookups from a container identifier to image files, and to the allocated zones.

C. Implementation of ZNS-CRIU

We implement our proposal using the Linux checkpointing tool, called CRIU. Specifically, we have modified CRIU version 3.8.13 and have added functionalities shown in Figure 4. Our tool, which we refer to as ZNS-CRIU, consists of three core components, zone initialization, independent zone allocation, and zone striping.

The zone initialization component is responsible for ZNS SSD initialization and independent zone group probing. The independent zone allocation component provides an interface so that a container can specify its QoS and takes charge of zone allocation and metadata management to manipulate the mapping between containers and zones. Finally, the zone striping component distributes a checkpointed image across multiple zones by actually controlling ZNS SSDs.

In the current implementation, ZNS SSDs are controlled directly by ZNS-CRIU at a user level. Initially, we plan to use ZNS-aware file systems such as F2FS [20] and ZoneFS [21]. However, since our design requires flexible management such as striping a file across zones and zone probing, we decide to implement all ZNS SSD controls at a user level and issue ZNS commands via the *ioctl* system call.

Algorithm 1 Docker container checkpoint (*pid*)

```
1:  get a required QoS from an CRIU argument
2:  allocate independent zones
3:  prepare image files // id, state, memory context, ...
4:  zone_cur ← zone_first;
5:  zone_cur_offset ← zone_cur_saved_offset;
6:  while each image_file do
7:      image_file_offset ← 0;
8:      while image_file_offset is not the end of a file do
9:          ZNS_write(zone_cur, image_file_offset);
10:         zone_cur_offset += write_size;
11:         zone_cur ← zone_next;
12:         zone_cur_saved_offset ← zone_cur_offset;
13:         zone_cur_offset ← zone_next_saved_offset;
14:         image_file_offset += write_size;
15:         if zone_cur_offset is the end of a zone then
16:             add_new_zone(zone_group);
17:     end
18: end
```

Algorithm 1 summarizes the checkpointing procedure. It first recognizes the required QoS of a container that wants to be checkpointed. Then, it allocates zones from unallocated groups or least used groups and freezes the container to prepare image files. Then, it initializes the current zone and its offset and goes into the first while loop where each image file is actually checkpointed.

In the first while loop, it initializes the offset of an image file and goes into the second while loop where the striping is performed. It writes the image file to ZNS SSDs asynchronously using *ZNS_write* until the offset of the image file reaches the end of the file. It also allocates a new zone from the same zone group when the already allocated zone is used up, as shown in line 15 in this algorithm.

V. Evaluation

We evaluate the performance of our proposal under the following testbed and workloads:

Testbed We use an Intel Core i5-4440 CPU @ 3.10GHz server (Ubuntu 20.04), 32GB DRAM, and two NVMe SSDs including 2TB traditional SSD and ZNS SSD, as mentioned in section III. The zone, whose size is 72MB, is mapped into one channel and way. Although the processor can handle only 4 concurrent threads, potentially leading to some level of contention among the threads for CPU resources, we have observed similar experimental results trend in another server equipped with an 8-core CPU.

Workloads We build several Docker images using various applications such as Redis, Nginx, MongoDB, and MySQL. We assume that each Docker requires 500MiB/s bandwidth for checkpointing. For generating normal I/Os, we run the FIO benchmark in a separate Docker.

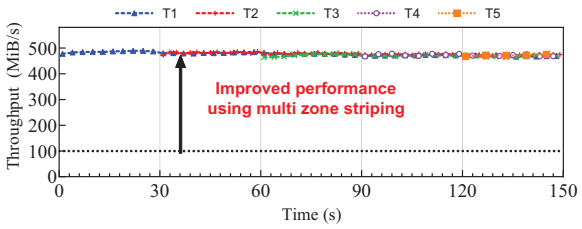

Fig. 5. **Improved performance of ZNS SSDs using multi-zone striping**

A. Zone striping impact on performance

Figure 5 shows how our zone striping technique impacts on the performance of small zone ZNS SSDs. For this evaluation, we conduct the same experiment as described in Figure 2. Each five sequential write threads start every 30 seconds. The only difference is that our striping technique is applied when accessing ZNS SSDs. We allocate five independent zones for each thread to support the 500MiB/s bandwidth and distribute data across the allocated zones.

Note that there are five threads, each requiring five independent zones, implying a total of 25 independent zone groups are needed. Because our experimental ZNS SSD device has 32 groups, we can allocate free independent zone groups for all 5 threads. If two additional threads arrive and each requires 5 independent zones, meaning additional 10 independent zone groups are needed, we serve it in a fair-share manner. For the seventh thread, we allocate two zones from the remaining free groups, while three zones from groups used that have already been allocated to previous threads. We select groups mapped to the smallest number of threads first for fairness.

In Figure 2(b), we have observed that the throughput for a single-thread in ZNS SSDs was significantly lower than that of traditional SSDs, as marked by the dotted line in Figure 5. However, if we stripe data across multiple zones, the write throughput of ZNS SSDs increases almost to 500MiB/s, while providing stable performance regardless of the number of concurrent threads. This demonstrates that with our proposed techniques applied, small zone ZNS SSDs can provide both performance isolation and improved performance.

B. Workload separation using independent zone allocation

We now discuss the result of our independent zone allocation scheme that enables performance isolation, leading to high predictability. We evaluate whether independent zone allocation can support such a goal by running checkpointing with normal I/Os.

Figure 6 shows a comparison between traditional SSDs and ZNS SSDs when Docker checkpointing is performed concurrently to normal I/Os generated by FIO. On traditional SSDs, checkpointing and normal I/Os can go into the same channels and ways, resulting in throughput fluctuations. The throughput of normal I/Os varies from 100 to 900 MiB/s, which significantly produces long tail latencies.

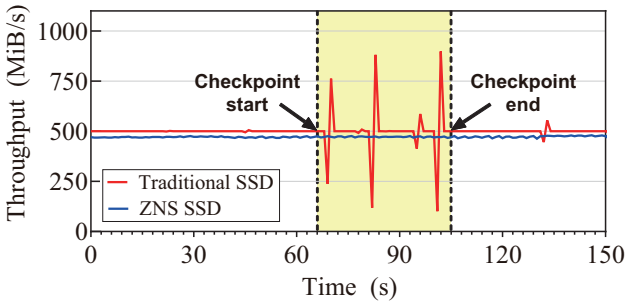

Fig. 6. **Docker checkpoint impacts on traditional SSDs and small zone ZNS SSDs**

Conversely, we can hardly see performance interference caused by Docker checkpointing on ZNS SSDs. This is because our proposal can separate Docker checkpointing from normal I/Os by allocating different zones. Along with the advantage of performance isolation from different zone allocations, Figure 6 also shows that ZNS SSDs can support improved performance, comparable to traditional SSDs.

C. Consolidated impact to multiple Docker checkpointing job

Figure 7 shows a consolidated result applied to our proposal. In this evaluation, we deploy five Docker containers, each running Ubuntu, Redis, Nginx, MongoDB, and MySQL, respectively. In this figure, *TrSSD cp* represents the checkpoint latency of a single Docker, denoted in the x-axis, on traditional SSDs. *TrSSD Parallel cp* and *ZnSSD parallel cp-striping* are the checkpoint latency of a single Docker under traditional SSDs and ZNS SSDs, respectively, when five Docker checkpoints are performed concurrently.

As the size of the Docker image increases (Ubuntu: 63.1MB, Redis:105MB, Nginx: 133MB, Mongo: 303MB, and Mysql: 447MB), the checkpoint latency also increases. When multiple Dockers are checkpointed concurrently, the performance degradation is significant on traditional SSDs. This stems from the lack of performance isolation in traditional SSDs. However, when using ZNS SSDs, the workload of five parallel checkpoints shows comparable performance to that of

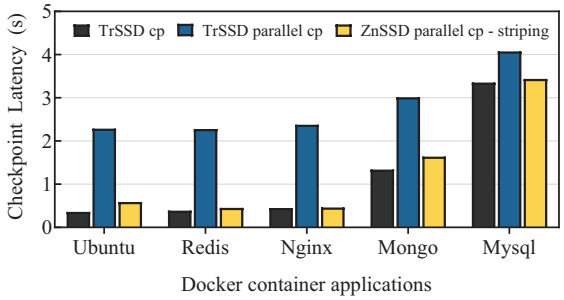

Fig. 7. **Performance comparison when multiple checkpoints are conducted in parallel.**

a single checkpoint on traditional SSDs. This demonstrates that zone-aware allocation and striping based on multiple zones can support isolation even under multi-tenant environments.

VI. Related Work

Existing studies related to our research can be classified into three categories. First, there have been several attempts to address performance interference issues in traditional SSDs [22]–[25]. Park et al. [22] propose a fair efficient flash I/O scheduler. Chang et al. [23] present a framework, called VSSD, that provides performance isolation in a solid-state drive by mitigating inter-reference interference among different processes, thereby enhancing both efficiency and fairness.

Shen et al. [24] introduced FlashFQ, a fair queueing I/O scheduler for flash-based SSDs. Wen et al. [25] proposed a workload-adaptive over-provisioning space allocation mechanism for multi-tenant SSDs. These works have made significant contributions to understanding and enhancing SSD performance isolation. But they primarily investigate these issues in the context of traditional SSDs, not considering the emerging Zoned Namespace Solid State Drives (ZNS SSDs) which our research focuses on.

Second, based on ZNS SSDs, several works on performance improvement or workload isolation in ZNS SSDs [4], [6]–[8], [11], [13], [15]. Bjørling et al. [4] advocate ZNS SSDs since they can avoid device-side and host-side storage overhead, referred to as block interface tax. Han et al. [11] show how flash-level copy-offloading can overcome host-level file system overhead on ZNS SSDs. They use large zone ZNS SSDs to utilize the SSD-internal flash chip parallelism but do not cover channel-level isolation.

Bae et al. [7] evaluate performance characteristics of both large zone and small zone SSDs. In addition, they suggest a novel I/O scheduler that can mitigate zone contention in small zone ZNS SSDs. Im et al. [6] examine the lack of internal parallelism of small zone ZNS SSDs and propose I/O mechanisms applied to ZenFS to exploit external parallelism. However, they do not consider the usage of ZNS SSDs in a cloud environment.

Lastly, there have been several studies about performance isolation and QoS support in the Docker environment [26]–[29]. Kwon et al. [26] propose hardware and software support to isolate noisy neighbor containers and avoid I/O interference. Xavier et al. [27] discuss performance interference from disk-intensive workloads in containerized platforms where QoS is a crucial factor. They propose a workload-balanced scenario to prevent performance degradation.

Li et al. [28] address the performance isolation optimization to allocate the storage resource according to their performance behaviors under resource competition situations. McDaniel et al. [29] point out that existing solutions, such as guaranteeing QoS to mitigate performance loss, do not target managing I/O contention. To control the I/O of Docker containers, they suggest cluster/node approaches. These prior works involve performance isolation and QoS control in Docker environments, similar to our motivation, but they primarily investigate the problems in the context of traditional SSDs.

Contrary to the aforementioned studies, our work is the first proposal about applying ZNS SSDs in a Docker environment. This opens up the possibility that ZNS SSDs can be actively utilized in cloud environments.

VII. Conclusion

In this work, we explore the feasibility of small zone ZNS SSDs in Docker environments. For supporting Docker isolation, traditional schemes make use of I/O scheduling, load balancing, and resource reservation. On the contrary, our scheme takes a new approach, making use of ZNS SSDs. Based on ZNS SSDs, we devise the independent zone allocation to support isolation and striping across multiple zones to provide comparable performance to traditional SSDs.

In future research, We plan to further improve our checkpoint scheme by adding admission control under a multi-tenant environment when free zone groups are not sufficient. Additionally, We aim to compare our scheme with software-based interference mitigation technology and explore the use of ZNS-aware file systems, such as F2FS and ZoneFS.

Acknowledgments

This research was supported by the Ministry of Science, ICT (MSIT) of Korea under the High-Potential Individuals Global Training Program (RS-2022-00154903) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation), as well as by two grants: the Institute for Information and Communications Technology Planning & Evaluation (IITP) funded by the Korea government (MSIT) (No.2021-0-01475, SW StarLab), and the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No.2022R1A2C1006050).

References

[1] A. Webster, R. Eckenrod, and J. Purtilo, "Fast and service-preserving recovery from malware infections using criu," in *USENIX Security Symposium*, 2018, pp. 1199–1211.

[2] S.-H. Choi and K.-W. Park, "icontainer: Consecutive checkpointing with rapid resilience for immortal container-based services," *Journal of Network and Computer Applications*, vol. 208, 2022.

[3] H. Zhang, N. Chen, Y. Tang, and B. Liang, "Multi-level container checkpoint performance optimization strategy in SDDC," in *Proceedings of the 4th International Conference on Big Data and Computing*, 2019, pp. 253—259.

[4] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, and G. Amvrosiadis, "ZNS: avoiding the block interface tax for flash-based ssds," in *USENIX ATC*, USA, 2021, pp. 689—-703.

[5] 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 *ACM HotStorage*, 2022, p. 93–99.

[6] M. Im, K. Kang, and H. Yeom, "Accelerating Rocksdb for Small-zone ZNS SSDs by parallel I/O mechanism," in *ACM Middleware*, 2022, pp. 15—-21.

[7] H. Bae, J. Kim, M. Kwon, and M. Jung, "What you can't forget: Exploiting parallelism for zoned namespaces," in *ACM HotStorage*, 2022, pp. 79—-85.

[8] M. Oh, S. Yoo, J. Choi, J. Park, and C.-E. Choi, "ZenFS+: Nurturing performance and isolation to ZenFS," *IEEE Access*, vol. 11, pp. 26 344–26 357, 2023.

[9] C. Community, "Checkpoint/restore in userspace(criu)," 2019. [Online]. Available: http://criu.org
[10] "Nvm express 2.0 zoned namespace command set specification," 2018. [Online]. Available: https://nvmexpress.org/specifications/
[11] 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.
[12] G. Choi, K. Lee, M. Oh, J. Choi, J. Jhin, and Y. Oh, "A new LSM-style garbage collection scheme for zns ssds." in *USENIX HotStorage*, 2020, pp. 1–6.
[13] 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 *ACM ASPLOS*, 2023, pp. 660–673.
[14] J. Jung and D. Shin, "Lifetime-leveling lsm-tree compaction for zns ssd," in *ACM HotStorage*, 2022, pp. 100–105.
[15] R. Liu, Z. Tan, Y. Shen, L. Long, and D. Liu, "Fair-zns: Enhancing fairness in zns ssds through self-balancing I/O scheduling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.
[16] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, "Fast in-memory criu for docker containers," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 53—-65.
[17] "Flexible I/O tester," https://github.com/axboe/fio, 2022.
[18] P. J. Marandi, C. Gkantsidis, F. Junqueira, and D. Narayanan, "Filo: Consolidated consensus as a cloud service," in *USENIX ATC*, 2016, pp. 237—-249.
[19] J. Ortiz, B. Lee, M. Balazinska, J. Gehrke, and J. L. Hellerstein, "Slaorchestrator: Reducing the cost of performance slas for cloud data analytics," in *USENIX ATC*, 2018, pp. 547—-560.
[20] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *USENIX FAST*, 2015, pp. 273–286.
[21] "File systems for zoned storage," https://zonedstorage.io/docs/linux/fs, 2023.
[22] S. Park and K. Shen, "FIOS: A fair, efficient flash I/O scheduler," in *USENIX FAST*, 2012.
[23] D.-W. Chang, H.-H. Chen, and W.-J. Su, "VSSD: Performance isolation in a solid-state drive," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, no. 4, pp. 1–33, 2015.
[24] K. Shen and S. Park, "FlashFQ: A fair queueing I/O scheduler for Flash-Based SSDs," in *USENIX ATC*, 2013, pp. 67–78.
[25] Y. Wen, Y. Zhou, F. Wu, S. Li, Z. Wang, and C. Xie, "WA-OPShare: Workload-adaptive over-provisioning space allocation for multi-tenant ssds," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4527–4538, 2022.
[26] M. Kwon, D. Gouk, C. Lee, B. Kim, J. Hwang, and M. Jung, "DC-Store: Eliminating noisy neighbor containers using deterministic I/O performance and resource isolation," in *USENIX FAST*, 2020, pp. 183–191.
[27] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, and C. A. D. Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds," in *Euromicro*, 2015, pp. 253–260.
[28] Y. Li, J. Zhang, C. Jiang, J. Wan, and Z. Ren, "Pine: Optimizing performance isolation in container environments," *IEEE Access*, vol. 7, pp. 30 410–30 422, 2019.
[29] S. McDaniel, S. Herbein, and M. Taufer, "A two-tiered approach to I/O quality of service in docker containers," in *2015 IEEE International Conference on Cluster Computing*, 2015, pp. 490–491.