Can ZNS SSDs be Better Storage Devices for Persistent Cache?

Chongzhuo Yang, Zhang Cao, Chang Guo, Ming Zhao, Zhichao Cao
School of Computing and Augmented Intelligence, Arizona State University
Tempe, Arizona

Abstract

Block-based regular SSDs have been widely used as storage backends for persistent cache systems due to their explicitly lower cost and persistence compared to DRAM. However, the caching workloads are both write- and update-intensive. It incurs a large amount of device-level write amplification (WA) in the internal garbage collection (GC), which can lead to SSD lifespan and potential performance issues. Zoned Namespace SSDs (ZNS SSDs) offer a new interface for modern SSDs to overcome the limitations of regular SSDs in some use cases. As ZNS SSDs need much lower internal over-provisioning, they can offer a larger capacity compared with regular SSDs. Considering these two advantages of ZNS SSDs, we aim to explore three possible schemes to adapt the existing persistent cache system on ZNS SSDs and analyze their benefits and limitations. We conduct comprehensive evaluations to further illustrate the tradeoffs of each scheme. Based on our research and investigation, we conclude that ZNS SSDs exhibit promising results as better storage backends for persistent cache. Further, the co-design between cache management and zone management can potentially enhance the cache efficiency and performance.

CCS Concepts

• **Information systems** → **Storage management**.

Keywords

ZNS SSDs, Caching System, Write Amplification

ACM Reference Format:
Chongzhuo Yang, Zhang Cao, Chang Guo, Ming Zhao, Zhichao Cao. 2024. Can ZNS SSDs be Better Storage Devices for Persistent Cache?. In *16th ACM Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE '24), July 8–9, 2024, Santa Clara, CA, USA.* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3655038.3665946

HOTSTORAGE '24, July 8–9, 2024, Santa Clara, CA, USA

ACM ISBN 979-8-4007-0630-1/24/07
https://doi.org/10.1145/3655038.3665946

1 Introduction

Flash-based SSDs have been widely used for cache systems due to low cost and persistence compared with DRAM [2, 11]. Existing works mainly focus on caching data on regular block-based SSDs (i.e., regular SSDs). Regular SSDs' garbage collection (GC) is managed by Flash Translation Layer (FTL) and cannot be controlled by applications. With many random and small writes to SSDs, the uncontrollable GC will incur a high write amplification (WA), unstable throughput, and high tail latency [11]. This issue becomes more pronounced when the SSD capacity utilization is high [27]. Unfortunately, caching workloads consist of small, intensive, random updates with high capacity utilization [11, 27, 30]. The additional in-device data movements will further decrease the lifespan of the SSDs [4]. The open-channel SSDs [5, 14, 35] can separate different data streams into different channels, relieving WA and GC penalties. However, open-channel SSDs need extra work on the user client.

Zoned Namespace SSDs (ZNS SSDs) offer a new interface for modern SSDs to overcome the aforementioned limitations of regular SSDs [3]. Similar to other zone-based storage devices [1, 9, 37–39], with sequential write and zone-based cleaning constraints, ZNS SSDs can avoid internal GC. The GC task can be managed by the applications, which provides the potential to reduce WA. More importantly, ZNS SSDs can offer explicitly larger storage capacities and more stable performance compared to compatible regular SSDs due to less over-provisioning (OP) space and simple internal operation logic [4]. Considering the potential of reducing the WA and larger capacity with ZNS SSDs, in this paper, we want to explore and answer the following question: *can ZNS SSDs be better storage devices for persistent cache?* In particular, we will explore the following detailed perspectives: 1) How to adapt the existing flash-based persistent cache systems on ZNS SSDs? 2) Can a ZNS SSD-based cache achieve better performance in terms of throughput, latency, and hit ratio? 3) Additionally, what advantages or disadvantages does the new zone-based interface bring to caching workloads? In this paper, we utilize CacheLib [2], a general cache framework developed by Meta, to demonstrate and analyze.

To answer these questions, we present the three possible schemes that can adapt CacheLib to utilize ZNS SSDs as the persistent back-end and analyze their tradeoffs. In the first scheme, we run CacheLib on a ZNS-compatible file system like F2FS [21] (called **File-Cache**). The file system will handle all low-level operations management, so this approach can provide convenience for CacheLib to use ZNS SSDs. However, this design will lead to serious performance degradation and substantial WA since the management of file system is too heavy for cache access patterns. The internal indexing, I/O, and management are not designed and optimized for cache. The second scheme directly maps the cache on-disk management unit (i.e., region) to the fixed-size zone (called **Zone-Cache**). This method can achieve true zero WA and be GC-free. However, it lacks flexibility due to the fixed zone size and may use an extremely large region size. The large region size may suffer from performance degradation and a low cache hit ratio when the cache size is small. In the third scheme, we propose to use a simple middle layer to translate the zone interface to the region interface (called **Region-Cache**). The middle layer can support flexible region sizes on ZNS SSDs, which avoids overheads of the large region size, but it needs GC to clean the zones. Moreover, it provides opportunities for persistent cache to directly manage and optimize the allocation and zone cleaning.

To further illustrate the tradeoffs of each solution, we conduct comprehensive evaluations in Section 4. Specifically, we utilize the CacheBench from CacheLib [12] and integrate Cachelib into RocksDB as the end-to-end case to evaluate the three proposed solutions compared with the compatible regular SSDs. The analysis and evaluations lead us to the following observations:

1. Using a file system on ZNS SSDs is a convenient solution but it can lead to high overheads and high costs.
2. The specific caching design on ZNS SSDs can achieve a better hit ratio (from 94.29% to 95.08% in Section 4.1) than regular SSDs due to their larger device capacity and lower OP ratio (assuming hardware cost is same).
3. ZNS SSDs can reduce the tail latency and lower WA compared with regular SSDs, which gives benefits for caching workloads (up to 20% throughput improvement and 42% tail latency reduction in RocksDB compared to caching with regular SSDs in Section 4.2).
4. The persistent cache systems can potentially achieve explicitly higher throughput and lower WA with refined co-design between cache management and ZNS SSD zone management.

We concluded ZNS SSDs can be **better storage devices for persistent cache**. To help further investigations and research, we open-sourced our code in **Github** [1].

[1]https://github.com/asu-idi/ZNS-Cache

2 Background and Motivations

In this section, we will present the background and related work about flash cache and ZNS SSDs, motivating our work.

2.1 Flash Cache and CacheLib

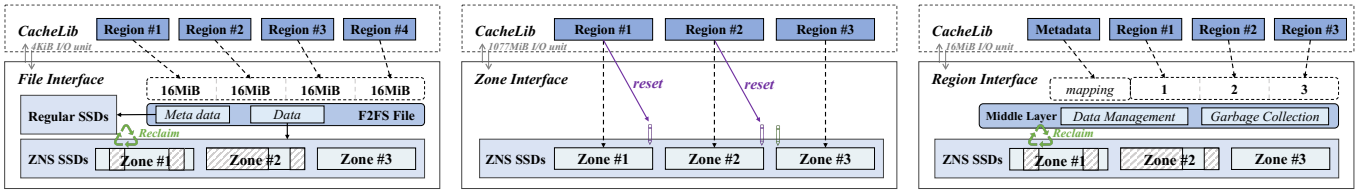
Flash cache has been widely used in current data infrastructure due to its higher cost-effectiveness and significantly larger capacity compared to the DRAM-based cache system. Several studies have been conducted to design and optimize flash-based cache systems [2, 11, 27–29], and they mainly focus on caching data on regular SSDs (i.e., the SSDs with the block interface). One notable example is CacheLib [2]. CacheLib is a pluggable caching engine developed by Meta, aiming to build and scale high-performance cache services. In log-structured cache of CacheLib, the flash space is partitioned into ***regions***, and each region is used to package cache objects with different sizes. To amortize the increasing GC cost of frequent cache object evictions and insertions, CacheLib evicts entire regions rather than individual cache objects, and region size is configurable, e.g., 16 MiB.

2.2 Zoned Namespace SSDs

Zoned Namespace SSDs (ZNS SSDs) are a new type of SSDs that divides flash into different zones [3]. In each zone, we can read randomly like the regular block devices but data can only be written sequentially and in a controlled manner (i.e., only write data at the write pointer's position). The write pointer can be moved sequentially by *write* or *append*, shifted to the start by the *reset*, or jumped to the end of the zone by *finish*. This design simplifies the internal GC in FTL (known as the device-level GC), as the GC task is mainly instead managed by applications. The user-level GC is like the open-channel SSDs [3, 4, 35]. By better managing the data stored in ZNS SSDs, GC and WA can be potentially reduced [17, 36]. Also, ZNS SSDs can have a larger capacity (e.g., 7-28%) than the compatible regular SSD (i.e., the same hardware and cost) since less internal OP space is needed [3, 4]. Existing studies have demonstrated the benefits of deploying and optimizing ZNS SSDs on specified applications, including LSM-based key-value stores [6, 17, 23, 26], RAID systems [20, 24, 36], and log-structured systems [16, 22, 32, 33].

2.3 Motivations

The current persistent cache system based on regular SSDs is facing severe WA and high tail latency issues. Many works have pointed out that the caching data on regular SSDs may incur a large device-level WA factor if the write behavior is not well-designed [11]. WA can be even higher when we have a large number of random and small writes on SSDs. Moreover, the internal GC of regular SSDs can lead to significant performance regression and WA when the capacity utilization is high [27]. Unfortunately, those are the main



(a) File-Cache (b) Zone-Cache (c) Region-Cache

Figure 1: Architecture of the three possible schemes to using ZNS SSDs in a cache system.

writing behaviors and utilization of caching workloads (i.e., small, intensive, and random updates as well as high capacity utilization). Moreover, there exists uncontrollable GC in regular SSDs because each writes to a regular SSD has the potential to trigger GC, leading to a higher tail latency and influencing the performance [4]. Those issues are mainly caused by the internal FTL designs, characteristics of regular SSDs, and the mismatch between legacy block interfaces and cache management units.

Compared to regular SSDs, ZNS SSDs offer the potential to reduce the WA and tail latency by avoiding internal GC. Further, it can offer larger storage capacity for the same device cost [3]. Less WA can increase the lifespan of the SSDs, and avoid throughput reduction and unstable latency caused by additional in-device data movements. Moreover, when the total device cost is the same, a cache system with a larger capacity can typically lead to a better overall cache hit ratio [31] (the cache size is smaller than the working set of the current workload). Given these two advantages of ZNS SSDs, we aim to explore the possibilities and tradeoffs of utilizing ZNS SSDs as the backend of a persistent cache. First, we will explore and analyze three solutions that can represent most of the possible design spaces when utilizing ZNS SSDs as the cache backend (i.e., fully transparent layer, directly managed, and using a customized middle layer). Then, based on the insights and observations, we will discuss more potentials for ZNS SSDs to achieve lower WA and higher throughput compared to the compatible regular SSDs.

3 Are The Three Solutions Good Enough?

In this section, we will explore the following solutions that cover three possible design spaces: 1) *File-Cache*, using a ZNS-compatible file system for full transparency; 2) *Zone-Cache*, the cache system directly manages the ZNS SSDs by matching the cache management unit with the zone; and 3) *Region-Cache*, using a simple middle layer to provide a region interface for cache. The overall architectures of these three schemes are shown in Figure 1. We will focus on the following four measurement metrics (i.e., **flexibility**, **space efficiency**, **performance**, and **WA**) and answer the following questions: 1) Is this a feasible solution? 2) What are the potential benefits? 3) What are the tradeoffs?

To verify our analysis, we also compare these three schemes with CacheLib on regular SSDs (*Block-Cache*). The overall evaluation is shown in Figure 2 (details can be found in Overall Comparison of Section 4.1). We will analyze the results in the following sections.

3.1 Can We Use a ZNS Compatible File System To Support Persistent Cache?

Answer: Absolutely yes! CacheLib and other persistent cache are designed to use either a raw regular block device (no file system) or one large file allocated in a file system. Therefore, the ZNS SSD can be formatted with a compatible file system so that the CacheLib can directly use its file-base engine to read and write regions on a pre-allocated file. The architecture is shown in Figure 1(a). All the low-level operations including zone allocation, zone cleaning with GC, and indexing are applied and managed by the file system, which is fully transparent to CacheLib.

After configuring the file system, all I/O operations can directly leverage the file interface to manage. Thus, the cache can treat the ZNS SSD like a regular device. However, this design pays a high price on both performance and WA. First, current ZNS SSDs compatible file system (e.g., F2FS [21]) needs additional space provisioning (e.g., 20%) to conduct garbage collection management, lowering the benefits of large space of using ZNS SSDs. Second, the file system is designed for general use cases. The frequent cache unit overwrite/update can lead to unexpected file system internal GC and zone cleaning, which causes high overhead and WA. Finally, the full transparency makes us lose the chance to optimize GC and WA by using application-level hints.

In Figure 2, the throughput and hit ratio of File-Cache are lower than those of Block-Cache due to its additional mapping overhead, OP space requirement, and GC overhead [33]. Therefore, using a ZNS-compatible file system like F2FS to support persistent cache on ZNS SSD is feasible and convenient, but it will bring explicitly high overhead.

3.2 How About Matching the Cache Management Unit with Zone?

Answer: It might be a better solution! Most of the persistent cache designs, including CacheLib, group the newly

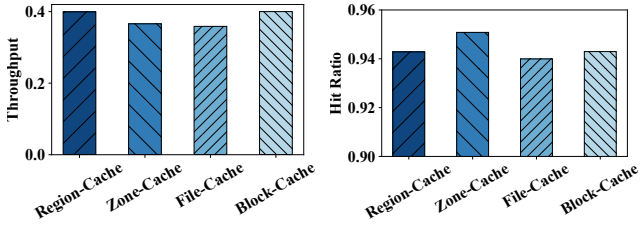
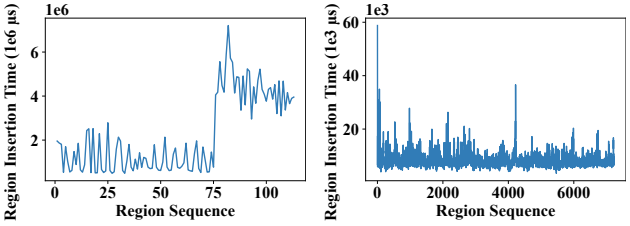


Figure 2: Performance of the four schemes.



(a) Large Region (b) Small Region

Figure 3: Time to fill the region in-memory buffer.

inserted cache objects into a much larger management unit (e.g., fixed-size regions) to reduce the WA and improve the I/O efficiency by allocating and evicting large I/O units on SSDs. If space is not enough, one region (e.g., at the LRU end) will be evicted and overwritten by the upcoming new region. If we enlarge the region size to match the zone size (i.e., one region per zone), CacheLib can directly use ZNS SSDs to persist the regions by adding similar zone management operations in region management as shown in Figure 1(b).

This design brings several benefits. First, since one region fully utilizes one zone, when a region is evicted, the zone can be directly reset without any data migration. This scheme can achieve real zero WA and be GC-free. Second, extra indexing is not needed in this scheme as CacheLib can directly use ZNS SSDs by adding one entry of zone number to the region metadata for I/Os. Finally, this scheme can achieve the best space utilization since no OP is needed for GC.

Zone-Cache seems to be the best design. However, if we need to match the region to a large zone size (e.g., 1077 MiB in Western Digital ZNS SSD [4]), a very large region size can cause other issues. First, we have to evict the entire region when performing a cache eviction. Evicting a large region will cause many valid or hot cache objects to be evicted at the same time, which will explicitly impact the hit ratio (validated in RocksDB evaluation at Section 4.2). Second, a larger region size requires setting up a larger region buffer in memory to cache the newly inserted objects, which consumes more DRAM space. Third, a large region will cause a long allocation time in eviction and a long filling time in insertion, reducing the parallelism effectiveness.

Moreover, we collected the insertion time to fill the region in-memory buffer using a large region (i.e., 1024MiB) as shown in Figure 3(a). The insertion time refers to the duration it takes for CacheLib to insert key-value pairs into the DRAM buffer. When the buffer is full, CacheLib will flush its contents to a flash device. The time significantly increases when region eviction begins at sequence 76, which does not occur in a small region design (i.e., 16 MiB) as shown in Figure 3(b). We think the increased insertion time is caused by eviction operations in other threads, which involve lock controls for the shared index. Additionally, we found that the time to fill a 1024 GiB region is longer than the time to fill 64 regions of 16 MiB each. Unlike the small region, the coarse-grained parallelism of the large zone incurs more overhead. If the ZNS SSD is produced with a small zone size [15] (e.g., 16 or 64 MiB), Zone-Cache might be a good design to avoid the overhead of large region size. However, the smaller zone may have lower per-zone throughput which needs additional designs.

The evaluations in Figure 2 show that *Zone-Cache can get the highest hit ratio but can not do well in throughput.*

3.3 Do We Need A Simpler Middle Layer?

Answer: It can be a better solution for some cases, but not all! In this section, we will explore another way to make the tradeoffs between four matrices. We add a simple middle layer to translate regions to physical zone addresses as shown in Figure 1(c), called *Region-Cache*. Compared to File-Cache, it's much simpler and introduces less overhead. Compared to Zone-Cache, it provides better flexibility and performance.

Data Management In the middle layer, we also use regions as I/O units. Unlike managements in Block-Cache and File-Cache with 4KiB block indexing, it will incur less mapping overhead. The mapping between the region ID and the in-zone address of ZNS SSDs is stored in a mapping (e.g., an ordered map). When CacheLib flushes a region, the middle layer will first write the data to ZNS SSDs and update the mapping. If CacheLib rewrites a region, the mapping corresponding to this region will be deleted, and the bitmap status of the zone will be updated. The bitmap is a set of 0/1 bits, and it will indicate whether the region is valid. For a zone with 1024MiB and 16MiB region, the bitmap will only cost 64 bits of space. The proposed middle layer supports concurrent writing of multiple zones at the same time. The zone is finished when there is no space to write a new region. When one read operation is called, the middle layer will look up the mapping by the region ID, and compute the real physical address using the in-region offset and in-zone address.

Garbage Collection In this design, we will use a background thread to check the empty zone number and valid data size of the finished zones. If the number of empty zones is less than the number of minimal empty zones (e.g., 8 zones),

we will select one zone (e.g., less than 20% of the zone capacity is occupied by the valid regions) to apply GC (i.e., migrating the valid regions to other open zones and reset this zone). Note that, the GC threshold and the zone selection threshold are configurable and can be different in different workloads, cache policies, and cache setups. Exploring the thresholds can be the future work.

The evaluations in Figure 2 show that *Region-Cache can get a high throughput similar to Block-Cache, indicating the design space in caching on ZNS SSDs.*

3.4 Discussion

Based on the analysis and evaluation, we find the File-Cache scheme is not a suitable scheme for caching on ZNS SSDs. It is limited by its file interface and high OP space. For the Region-Cache and Zone-Cache, they have different advantages. Compared to Block-Cache, the Region-Cache can achieve similar throughput and hit ratio, and achieve lower tail latency compared to Block-Cache (shown in Section 4.2). Zone-Cache can get a larger cache size and be GC-free, thus its hit ratio is higher than Block-Cache. Also, zero WA can make Zone-Cache achieve a much longer SSD lifespan. Zone-Cache is a compelling design especially when the zone size is small.

The middle layer design (Region-Cache) makes the tradeoffs between File-Cache and Zone-Cache, which achieves both high flexibility and high performance. More importantly, the middle layer scheme opens the design space to further optimize the throughput and WA by conducting the co-design between cache management and zone management. For example, during the zone GC, not all the valid regions are needed to be migrated. By using the cache or upper application information or hints, the GC overhead can be effectively minimized without explicitly sacrificing the cache hit ratio [34].

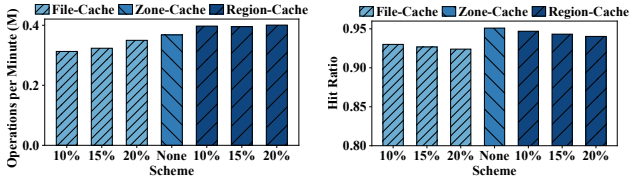
In conclusion, the Zone-Cache can perform **better in the hit ratio**. The Region-Cache can perform **better in throughput**. And the File-Cache is not a suitable design.

4 Evaluations

In this section, we evaluated three schemes (i.e., ***File-Cache***, ***Zone-Cache***, and ***Region-Cache***) on a real ZNS SSD, and compared with the CacheLib on regular SSDs (***Block-Cache***). Our evaluations were conducted on the ASUSTeK ESC4000 server with Intel(R) Xeon(R) Silver 4210 CPUs and 187GiB DRAM memory. The ZNS SSD is 1TB Western Digital Ultrastar DC ZN540 with 904 zones and the zone size is 1077MiB. The regular SSD is a hardware-compatible 1TB SN540 SSD. The 6GiB regular block device for F2FS is created by *nullblk*.

4.1 Micro Benchmark Evaluations

In this section, we will evaluate different schemes using the CacheBench [12] benchmarks from CacheLib. We use the workload at *feature_stress/navy/bc* which has 50% get, 30% set, and 20% delete operations. We use LRU as the cache eviction policy in CacheLib.



(a) Throughput (b) Hit Ratio

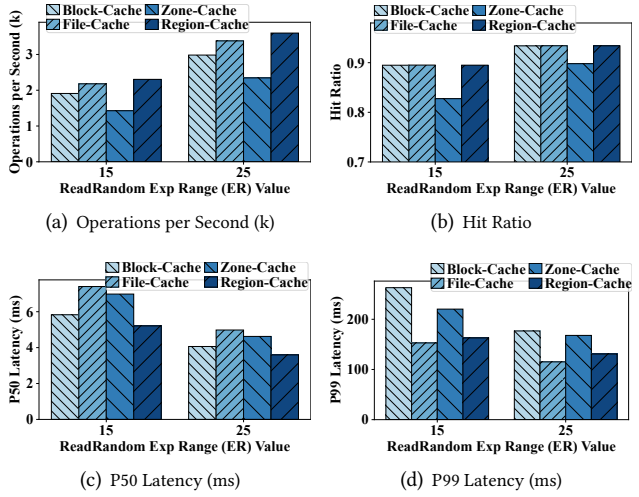
Figure 4: Performance of under different OP schemes.

Overall Comparison. First, we evaluate the four schemes and mainly focus on their throughput and hit ratio. In Zone-Cache and Region-Cache, we all use 25 zones. As Zone-Cache does not need OP space, we set the cache size to 25GiB. For Block-Cache, File-Cache, and Region-Cache, we use 20GiB cache size (assuming at least 5GiB OP space). For File-Cache, F2FS needs at least 38 zones and a 6GiB regular block device to build a 20GiB cache size. The result is shown in Figure 2. Block-Cache, File-Cache, and Region-Cache have similar hit ratios and they are lower than the Zone-Cache which has the largest cache size. Block-Cache has a high throughput compared to other ZNS-based schemes due to its large reserved space and smaller minimal-erase block. Zone-Cache can get higher throughput than File-Cache for its low overhead mapping and GC-free design. Region-Cache gets the best result by resolving the overhead of managing the large region size of Zone-Cache. The Zone-Cache can achieve a better hit ratio (from 94.29% to 95.08%) than the Block-Cache.

Table 1: WA Factor under different OP ratios.

Scheme	10%	15%	20%
Region-Cache	1.39	1.30	1.15
File-Cache	1.25	1.19	1.11

Evaluation under different OP ratios. To demonstrate the tradeoffs of the three ZNS-based schemes, we also conduct experiments with different OP ratios as shown in Figure 4 and Table 1. In this evaluation, we both use device space of 220 zones (about 230 GiB) and set different OP ratios 10%, 15%, and 20% (the Zone-Cache will always use 0% OP ratio). For Region-Cache and File-Cache, a larger OP ratio will lead to higher throughput and lower hit ratio, which indicates the tradeoff between throughput and hit ratio. The results also show higher WA can bring lower throughput. Zone-Cache is GC-free, and the WA Factor is always 1. For the Zone-Cache, it can get a higher hit ratio, but the throughput is limited by the overhead of managing such large regions.



(a) Operations per Second (k) (b) Hit Ratio

(c) P50 Latency (ms) (d) P99 Latency (ms)

Figure 5: Performance of the four schemes serving as the secondary cache of RocksDB.

4.2 End-to-End Evaluations with RocksDB

To conduct the comprehensive evaluation with real-world applications, we integrate the four schemes into RocksDB, a widely used LSM-based key-value stores [7, 18, 19, 25], as its secondary cache [8, 10] and utilize the db_bench [13] to evaluate them. Without explicit explanation, the key and value sizes are 16 bytes and 64 bytes respectively, with *index block caching* and *direct I/O* enabled. For other configurations, we keep the default settings. In the secondary cache (CacheLib), the DRAM size is set to 32MiB (the minimal DRAM size which allows the cache to work well). We use a 5GiB flash cache size and reserve enough OP space to reduce GC and focus on tail latency and throughput. We utilize Seagate ST6000NM0115 HDD (6TiB) as the backend of RocksDB. For the ReadRandom workload settings, we used ReadRandom Exp Range (abbreviated as ER) to control the data skewness (larger ER value means more skewed data). We selected ER values of 15 and 25 to represent different degrees of data skewness. Initially, we used the fillrandom workload to insert 100 million key-value pairs, and subsequently, we employed the readrandom workload to retrieve 1 million keys.

Throughput. As shown in Figure 5(a), the throughput of Region-Cache is highest, with up to 21% throughput improvement compared to Block-Cache. File-Cache gets better throughput than micro-benchmark due to enough OP space. As shown in Figure 5(b), Zone-Cache has the lowest overall throughput caused by its lowest cache hit ratio, which is mainly caused by large region eviction (i.e., all cached objects of a 1077 MiB zone are cleaned). Moreover, since the cache size is small (5 GiB), RocksDB is more sensitive to this large region eviction.

Table 2: Performance of different sizes on Zone-Cache for RocksDB.

Cache Size	4G	5G	6G	7G	8G
Throughput (k ops)	1.869	2.345	2.822	3.378	4.100
Hit Ratio (%)	86.95	89.80	91.54	93.03	94.40

Latency. In Figure 5(d), the tail latency of File-Cache is the lowest, even lower than Region-Cache, this is because F2FS is optimized for tail latency [21], and it achieves at most a 42% latency reduction compared to Block-Cache. We expect a better performance when small zone sizes (e.g., Samsung ZNS SSDs with 96 MiB zone size [15]) are provided. Since the read performance of RocksDB is sensitive to the secondary cache latency, the throughput of RocksDB with Block-Cache is even lower than File-Cache, mainly due to the high tail latency. As shown in Figures 5(c) and 5(d), the P50 latency of Block-Cache is low, but its P99 latency is the highest. The uncontrollable internal GC mainly causes this higher tail latency in regular SSDs.

Evaluation under different cache sizes. The results in Figure 5 show the limitations of Zone-Cache when the cache size is small. To be noted, the Zone-Cache used in the above experiments only uses the same cache size with other designs (i.e., 5GiB). From our previous discussion, the Zone-Cache can provide a larger cache size than other schemes. Therefore, we also conducted experiments on RocksDB where the ER value was 25 with different cache sizes, showing the importance of cache size. The result is shown in Table 2. The workload used in RocksDB is more sensitive to hit ratio as the data is stored in HDDs. We can get higher hit ratios and throughput when large cache sizes are provided to Zone-Cache. It show that using ZNS SSDs for caching can give a larger cache size than regular SSDs.

5 Conclusion

In this paper, we propose, analyze, and evaluate three possible schemes to use ZNS SSDs for persistent cache. Our findings suggest that ZNS SSDs can be better storage devices for persistent cache compared with regular SSDs, which is also verified by using ZNS SSD as a secondary cache for RocksDB. Our further work includes closing the semantic gap between cache management and zone GC with co-designs.

Acknowledgements

We would like to thank our shepherd, Jooyoung Hwang, and all the anonymous reviewers for their valuable feedback. We thank all the members of ASU-IDI Lab for providing useful comments. This work was partially funded by the Arizona State University startup fund and National Science Foundation awards 2311026, 2126291, and 1955593.

References

[1] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. 2015. Skylight—a window on shingled disk operation. *ACM Transactions on Storage (TOS)* 11, 4 (2015), 1–28.

[2] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[3] Matias Bjørling. 2020. Zone append: A new way of writing to zoned storage. *Santa Clara, CA, February. USENIX Association.[Cited on page.]* (2020).

[4] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 689–703. https://www.usenix.org/conference/atc21/presentation/bjorling

[5] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 359–374.

[6] Sungjin Byeon, Joseph Ro, Safdar Jamil, Jeong-Uk Kang, and Youngjae Kim. 2023. A free-space adaptive runtime zone-reset algorithm for enhanced ZNS efficiency. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. 109–115.

[7] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[8] Zhang Cao, Chang Guo, Ziyuan Lv, Anand Ananthabhotla, and Zhichao Cao. 2024. SAS-Cache: A Semantic-Aware Secondary Cache for LSM-based Key-Value Stores. In *38th Intl. Conf. on Massive Storage Systems and Technology*.

[9] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. 2023. SMRTS: A Performance and Cost-Effectiveness Optimized SSD-SMR Tiered File System with Data Deduplication. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 275–282.

[10] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, et al. 2023. Disaggregating RocksDB: A Production Experience. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–24.

[11] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: a Hybrid Key-value Cache that Controls Flash Write Amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 65–78. https://www.usenix.org/conference/nsdi19/presentation/eisenman

[12] FaceBook. 2023. cachebench. https://github.com/facebook/CacheLib/blob/main/BENCHMARKS.md. Accessed March 25, 2023.

[13] FaceBook. 2023. dbbench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools/. Accessed March 25, 2023.

[14] Javier González and Matias Bjørling. 2017. Multi-tenant I/O isolation with open-channel SSDs. In *Nonvolatile Memory Workshop (NVMW)*, Vol. 2017.

[15] Jin Yong Ha and Heon Young Yeom. 2023. zCeph: Achieving High Performance On Storage System Using Small Zoned ZNS SSD. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. 1342–1351.

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

[17] Jeeyoon Jung and Dongkun Shin. 2022. Lifetime-leveling LSM-tree compaction for ZNS SSD. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 100–105.

[18] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2022. Power-optimized Deployment of Key-value Stores Using Storage Class Memory. *ACM Transactions on Storage (TOS)* 18, 2 (2022), 1–26.

[19] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald G Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension.. In *USENIX Annual Technical Conference*. 821–837.

[20] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G Andersen, Gregory R Ganger, George Amvrosiadis, and Matias Bjørling. 2023. RAIZN: Redundant array of independent zoned namespaces. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 660–673.

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

[22] Euidong Lee, Ikjoon Son, and Jin-Soo Kim. 2023. An Efficient Order-Preserving Recovery for F2FS with ZNS SSD. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. 116–122.

[23] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. 2022. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*. 93–99.

[24] Jinhong Li, Qiuping Wang, Shujie Han, and Patrick PC Lee. 2024. The Design and Implementation of a High-Performance Log-Structured RAID System for ZNS SSDs. *arXiv preprint arXiv:2402.17963* (2024).

[25] Gaoji Liu, Chongzhuo Yang, Qiaolin Yu, Chang Guo, Wen Xia, and Zhichao Cao. 2024. Prophet: Optimizing LSM-Based Key-Value Store on ZNS SSDs with File Lifetime Prediction and Compaction Compensation. In *38th Intl. Conf. on Massive Storage Systems and Technology*.

[26] Linbo Long, Shuiyong He, Jingcheng Shen, Renping Liu, Zhenhua Tan, Congming Gao, Duo Liu, Kan Zhong, and Yi Jiang. 2024. WA-Zone: Wear-Aware Zone Management Optimization for LSM-Tree on ZNS SSDs. *ACM Transactions on Architecture and Code Optimization* 21, 1 (2024), 1–23.

[27] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/3477132.3483568

[28] Netflix. 2023. Netflix Technology Blog. Application data caching using ssds. https://netflixtechblog.com/application-data-caching-using-ssds-5bf25df851ef. Accessed March 25, 2023.

[29] Netflix. 2023. Netflix Technology Blog. Evolution of application data caching : From ram to ssd https://netflixtechblog.com/evolution-of-application-data-caching-from-ram-to-ssd-a33d6fa7a690. Accessed March 25, 2023.

[30] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. 2012. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems.. In *FAST*, Vol. 12.

[31] S Prybylski, Mark Horowitz, and John Hennessy. 1988. Performance tradeoffs in cache design. *ACM SIGARCH Computer Architecture News* 16, 2 (1988), 290–298.

[32] Devashish Purandare, Pete Wilcox, Heiner Litz, and Shel Finkelstein. 2022. Append is near: Log-based data management on ZNS SSDs. In *12th Annual Conference on Innovative Data Systems Research (CIDR'22)*.

[33] Dongjoo Seo, Ping-Xiang Chen, Huaicheng Li, Matias Bjørling, and Nikil Dutt. 2023. Is garbage collection overhead gone? case study of F2FS on ZNS SSDs. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. 102–108.

[34] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2018. Didacache: an integration of device and application for flash-based key-value caching. *ACM Transactions on Storage (TOS)* 14, 3 (2018), 1–32.

[35] Haitao Wang, Zhanhuai Li, Xiao Zhang, Xiaonan Zhao, Xingsheng Zhao, Weijun Li, and Song Jiang. 2018. Oc-cache: An open-channel ssd based cache for multi-tenant systems. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 1–6.

[36] Qiuping Wang and Patrick PC Lee. 2023. ZapRAID: Toward High-Performance RAID for ZNS SSDs via Zone Append. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. 24–29.

[37] Fenggang Wu, Bingzhe Li, Zhichao Cao, Baoquan Zhang, Ming-Hong Yang, Hao Wen, and David HC Du. 2019. ZoneAlloy: Elastic Data and Space Management for Hybrid SMR Drives. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*.

[38] Fenggang Wu, Bingzhe Li, Baoquan Zhang, Zhichao Cao, Jim Diehl, Hao Wen, and David HC Du. 2020. Tracklace: Data management for interlaced magnetic recording. *IEEE Trans. Comput.* 70, 3 (2020), 347–358.

[39] Fenggang Wu, Baoquan Zhang, Zhichao Cao, Hao Wen, Bingzhe Li, Jim Diehl, Guohua Wang, and David HC Du. 2018. Data management design for interlaced magnetic recording. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*.