A New LSM-style Garbage Collection Scheme for ZNS SSDs

Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi
Dankook University

Jhuyeong Jhin, Yongseok Oh
SK Hynix

Abstract

This paper explores how to design a garbage collection scheme for ZNS (Zoned NameSpace) SSDs (Solid State Drives). We first show that a naive garbage collection based on a zone unit incurs a long latency due to the huge size of a zone. To overcome this problem, we devise a new scheme, we refer to it as LSM_ZGC (Log-Structured Merge style Zone Garbage Collection), that makes use of the following three features: segment based fine-grained garbage collection, reading both valid and invalid data in a group manner, and merging different data into separate zones. Our proposal can exploit the internal parallelism of a zone and reduce the utilization of a candidate zone by segregating hot and cold data. Real implementation based experimental results show that our scheme can enhance performance by 1.9 times on average.

1 Introduction

ZNS SSDs are a double-edged sword. It gives an opportunity to improve performance and reduce WAF (Write Amplification Factor) by separating different workloads into separate zones [2, 6]. The downside is that a host needs to manage ZNS SSDs directly such as a zone reset and they have the sequential write constraint.

This paper examines design considerations when we devise a garbage collection scheme for ZNS SSDs. One simple approach is applying the conventional scheme such as segment cleaning used by LFS (Log-structured File System) [16, 22] or garbage collection used by FTL (Flash Translation Layer) [11, 14]. The only difference is that it applies garbage collection based on a zone unit instead of a segment one. However, our analysis reveals that this naive approach incurs a long garbage collection latency since the size of a zone (e.g. 0.5 GB or 1 GB) is much larger than that of a segment (2 MB or 4 MB).

To overcome this problem, we propose a new zone garbage collection scheme, called LSM_ZGC. We find that the traditional segment concept is still valuable for reclaiming a zone in ZNS SSDs. Specifically, LSM_ZGC divides a zone into multiple segments and manages their information such as validity bitmap and utilization individually. It conducts garbage collection in a LSM (Log-Structured Merge) style [18, 19] that reads all data from a candidate zone, identifies cold data, merges them into a zone while merging remaining data into another zone.

Our proposal has several advantages. Garbage collection based on a segment unit instead of a zone unit becomes an effective basis for hot/cold segregation and making our scheme in a pipelined fashion. Reading all data, both valid and invalid in a segment, can reduce the garbage collection overhead by exploiting the internal parallelism in a zone. Merging data into separate zones according to their hot/cold characteristics can enhance an opportunity to find a zone with lower utilization.

We evaluate LSM_ZGC on a real ZNS SSD prototype. We implement a user level benchmark tool that can manipulate the prototype directly and measure the overhead of various garbage collection schemes. Experimental results show that our scheme improves garbage collection performance by up to 2.3 times with an average of 1.9 times. It also reduces interference with other applications and provides better scalability with multiple threads.

The rest of this paper is organized as follows. In Section 2, we present our two observations that motivate this study. Section 3 describes how to design our scheme. Evaluation results are discussed in Section 4. We explain conclusion and future work in Section 5. Finally, discussion topics are given in Section 6.

2 Motivation

In this section, we briefly introduce the features of ZNS SSDs. Then, we discuss our observations conducted on a real ZNS SSD prototype. Details of our experimental prototype will be explained further in Section 4.

2.1 ZNS SSDs

ZNS SSDs are a kind of OCSSDs (Open Channel SSDs) that exposes SSD internals using the zone concept [2]. Specifically, the address space of a ZNS SSD device is divided into zones, which provides two benefits. First, they give a chance to enhance performance and decrease WAF by allocating data with different characteristics into separate zones. Second, they can reduce or even remove FTL functionalities, which allows to lessen DRAM usage and over-provisioning area in SSDs. Hence, many vendors recently announce their ZNS SSD solution and plan [3, 4, 13].

However, there are two challenges in ZNS SSDs. The first one is that host software needs to handle a zone explicitly such as a zone reset, open, read, write and zone garbage collection. The second challenge is that ZNS SSDs have a unique constraint, called sequential write constraint. All data in a zone are required to be written sequentially, like SMR (Shingled Magnetic Recording) drives [23, 24].

2.2 Observations

Figure 1 shows the quantitative zone garbage collection overhead on various zone utilizations. In this experiment, the sizes of a zone, a segment and a block are 1GB, 2MB and 4KB, respectively. It implies that a zone consists of 512 segments, while a segment consisting of 512 blocks. We measure the zone garbage collection overhead, that is the total elapsed time to copy a valid block from a candidate zone into a new zone until there is no remaining valid blocks in the candidate zone. The overhead also includes the time to reset the candidate zone.

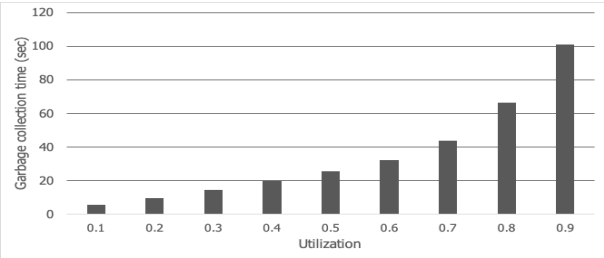

Figure 1: Zone garbage collection overhead under various utilizations of a zone

From this figure, we can observe that the garbage collection overhead increases as utilization grows as expected. One surprising thing is that the overhead becomes more than 20 seconds when utilization is bigger than 0.4 due to the copying overhead of valid blocks. This long elapsed time might interfere with user requests, incurring a long latency. Even though we can hide the overhead by introducing a preemptive approach, the huge size of a zone makes it quite complicated. This observation uncovers that reducing the utilization of a candidate zone is indispensable for ZNS SSDs.

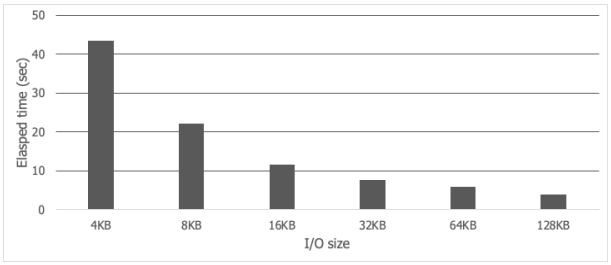

Figure 2: Access blocks individually vs. in a group manner

Figure 2 presents our second observation, comparison between the total elapsed time for reading all blocks in a zone individually (4KB I/O size per each request) and in a group manner (8KB $\sim$ 128KB I/O size per each request). The results reveal that accessing in a group manner is much faster than individual accesses. This is because it can not only reduce the number of requests but also make use of the internal parallelism in ZNS SSDs. In general, a zone in ZNS SSDs is spread into multiple channels which gives a chance to process a request with consecutive blocks in parallel, like OCSSDs [20]. This observation motivates us to design our LSM-style garbage collection scheme.

3 Design

In this section, we first describe a basic garbage collection scheme for ZNS SSDs. Then, we explain our scheme, contrasting differences between ours and the basic scheme.

One simple approach for zone garbage collection in ZNS SSDs is selecting a candidate zone whose utilization is the smallest. Then, it reads valid blocks from the selected zone and write them into a new zone. Finally, it issues the reset command for the selected zone. We refer to this scheme as Basic_ZGC as shown in Figure 3.

Our LSM_ZGC scheme has three differences. First, it conducts garbage collection based on a segment unit for reclaiming a zone. This approach makes it easy to segregate hot and cold data into different zones, which will be discussed further using Figure 4. In addition, it allows a zone garbage collection to be executed with a fined-grained segment unit where reading, merging and writing a segment can be done in a pipelined fashion.

Second, during garbage collection, it reads not only valid blocks but also invalid blocks in 128KB I/O size, whereas Basic_ZGC reads valid blocks only. Note that the original LFS, designed for hard disks, reads all data

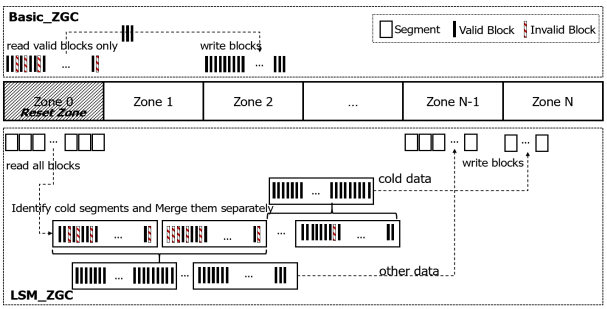

Figure 3: Design concept of LSM_ZGC in comparison with Basic_ZGC

like our scheme [22]. However, most of file systems and FTLs designed for SSDs reads valid data only since there is no seek overhead in flash memory [11, 14, 16]. We carefully argue that reading all blocks is a viable option in ZNS SSDs to fully obtain the internal parallelism observed in Figure 2. In actuality, we consider the utilization of a candidate segment when we design LSM_ZGC. Specifically, when the number of valid blocks in a segment is less than 16, LSM_ZGC reads valid blocks only. Otherwise, it reads all blocks since 16 requests with 128KB size can cover the whole 2MB segment data.

The third difference is that LSM_ZGC tries to identify cold data and merge them into a separate zone. For this purpose, we define four states of a zone, namely C0_zone, C1C_zone, C1H_zone and C2_zone, as shown in Figure 4. Newly arrived data is written sequentially into a zone whose state is C0_zone and deleted data is going out of the states presented in the figure.

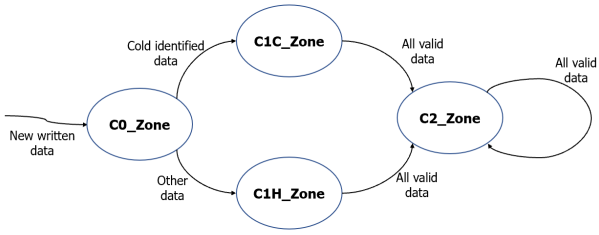

Figure 4: Zone state and transition

Now assume that LSM_ZGC selects a candidate zone whose state is C0_zone. It reads all segments in the zone and tries to identify cold segments. We define a segment whose utilization is above a threshold, called $threshold_{cold}$, as cold. This decision is based on our observation that data which has similar lifetime shows strong spatial locality. For example, in a key-value store, each level shows different lifetime and SSTables in a same level are written in a batch manner, which are also observed in previous studies [8, 15]. Valid blocks in a segment identified as cold are merged and written into a zone whose state is C1C_zone. Valid blocks of other segments are merged and written into another zone whose state is C1H_zone.

When a candidate zone is either C1C_zone or C1H_zone, LSM_ZGC reads all segments and treats all valid blocks as cold. This is because these valid blocks are survived after two garbage collection trials. They are merged and written into a zone whose state is C2_zone. We can extend further such as C3_zone and so on, but, in this study, we stop here and write valid blocks survived from C2_zone into another C2_zone. We expect that this mechanism enables to isolate cold data from others, which enhance an opportunity to find a candidate zone with lower utilization during garbage collection.

4 Evaluation

4.1 Experimental environment

We evaluate LSM_ZGC via real implementation based experiments. Our experimental system consists of Intel Core i7-6700K processor (8 cores), 16GB DRAM and 1TB ZNS SSD prototype. This prototype is developed by our team for research purpose, not a commercial product. Its internals are similar to the published ZNS SSDs [3, 9]. The prototype information is summarized in Table 1.

Table 1: ZNS SSD prototype information

Item	Specification
SSD Capacity	1TB
Size of a Zone	1GB
Number of Zones	1024
Interface	PCIe Gen3
Protocol	NVMe 1.2.1

On this hardware environment, we build a garbage collection benchmark tool that runs at a user level to evaluate our scheme. The tool is composed of three stages. In the first stage, it initializes the ZNS SSD prototype by writing dummy data until the overall utilization becomes a predefined target value. In this stage, we can configure control parameters such as the number of zones used for initialization and an overall target utilization value.

In the second stage, it updates the initialized data under various patterns such as uniform, skewed or user-specified. This updating is carried out until all initialized zones are covered either valid or invalid blocks. For instance, when we set up the number of zones and target value as 512 and 0.5, respectively, it fills 256 zones with dummy data in the first stage. Then, in the second stage, it modifies data (invalidating the original data and writing new valid data) until all blocks in 512 zones are either valid or invalid. Note that the utilization is still 0.5.

In the third stage, the tool executes Basic_ZGC or LSM_ZGC and measures its elapsed time. We can configure the number of free blocks we want to reclaim during garbage collection and $threshold_{cold}$. The tool also equips with functionalities such as a zone reset, open, read and write so that it can manipulate ZNS SSDs directly at a user level. Besides, it takes care of several data structures such as a bitmap per segment to distinguish valid and invalid blocks and usage information per segment and zone. We implement the tool by modifying the NVMe command line interface [1].

4.2 Garbage collection overhead

Figure 5 shows performance comparison results between two schemes. In this experiment, we set up the number of zones for initialization as 512 and the overall target utilization values as denoted in the X-axis in the figure. We also configure the update pattern as uniform, meaning that all segments have utilization similar to the overall target utilization. The number of reclaimed blocks and $threshold_{cold}$ are set as 512x512 and 0.4, respectively.

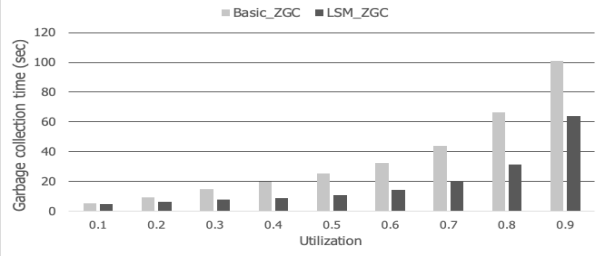

Figure 5: Performance comparison between Basic_ZGC and LSM_ZGC under uniform update pattern

From Figure 5, we can observe that our proposed LSM_ZGC outperforms Basic_ZGC by up to 2.3 times with an average of 1.9 times. When we count the number of copied blocks to a new zone during garbage collection, LSM_ZGC and Basic_ZGC show same numbers. It implies that the performance gain is derived from requesting all I/Os with 128KB size. Note that, for fair comparison, we implement Basic_ZGC that writes all data in 128KB size and reads valid blocks as large as possible if they are consecutive. However, the existence of invalid blocks prevents Basic_ZGC from generating requests with 128KB size. On the contrary, our scheme generates all requests with 128KB size, which are allowed by reading both valid and invalid blocks.

4.3 Effect of data separation

To evaluate the effect of data separation, we carry out an experiment that updates data in a skewed pattern. Specifically, in the second stage, the tool updates 30% data with 70% probability. Then, some segments have lower utilization than others, which eventually merged and written into zones with different states explained in Figure 4. After performing the second and third stages repeatedly, we measure the overhead under two schemes.

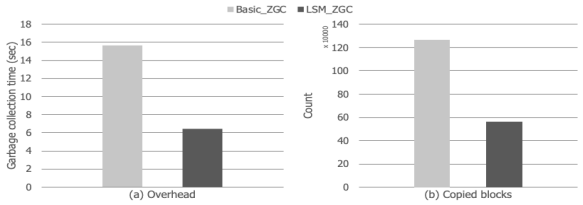

Figure 6: Performance comparison between Basic_ZGC and LSM_ZGC under skewed update pattern

Figure 6 shows performance comparison result when utilization is 0.8 (due to the page limitation, we present this result only, but other results with different utilizations show similar trends). It reveals that LSM_ZGC can reduce the garbage collection overhead, compared with Basic_ZGC. This gain comes from two sources. One is requesting I/Os with 128KB size. The second source is decreasing the copied blocks during garbage collection as shown in Figure 6 (b). LSM_ZGC segregates cold data from others, which allows to select a candidate zone that has lower utilization. This is more evident in Figure 7 that presents the utilization distribution of Basic_ZGC and LSM_ZGC. As you can see, data separation produces a bimodal distribution, which allows to boost the possibility to find a zone with low utilization.

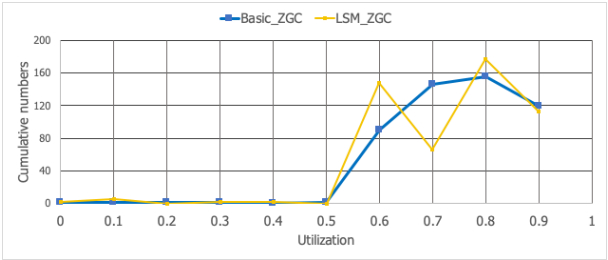

Figure 7: Zone utilization distribution between Basic_ZGC and LSM_ZGC

One essential reason that we employ the segment-based fine-grained garbage collection for reclaiming a zone is to separate hot and cold data appropriately. Due to the huge size of a zone, a zone has a tendency to contain both hot and cold data even in the skewed pattern, blurring whether a zone is hot or cold. But, it is more obvious at a segment level since a segment is much smaller than a zone. However, the effectiveness of data separation

depends on various parameters including hot/cold ratio, hot/cold data size, our control parameter $threshold_{cold}$, and initial hot/cold placement which are governed by allocation policy. We leave the investigation of these issues as a future work.

4.4 Effect on other applications

Figure 8 presents how LSM_ZGC affects the latency of other applications. For this experiment, we build a worker thread that writes a new 1GB file into the ZNS SSD prototype and reads it randomly. We measure the execution time of the worker under three cases; 1) when it runs alone, 2) when it runs concurrently with LSM_ZGC and 3) with Basic_ZGC.

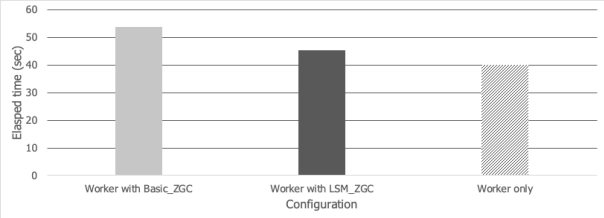

Figure 8: Effect of garbage collection on other applications

From Figure 8, we can observe that the execution time of the worker is around 40 seconds when it runs alone. With LSM_ZGC, the time is 45 seconds while it becomes 53 seconds with Basic_ZGC. These results show the tradeoff between the cost and benefit of LSM_ZGC. The cost is that LSM_ZGC increases the amount of read by reading both valid and invalid blocks during garbage collection, while the benefit is that it reduces the garbage collection overhead. Figure 8 demonstrates that the benefit can compensate the cost, diminishing the interference greatly.

4.5 Scalability

Figure 9 shows the performance when we run the worker with multiple threads. In this figure, the Y-axis is the throughput (MB/sec) relative to the case when we run a single worker thread with Basic_ZGC. These results reveal that multiple threads can enhance throughput on ZNS SSDs. We also observe that LSM_ZGC provides better scalability than Basic_ZGC.

However, the scalability of our ZNS SSD prototype is not linear. When we design ZNS SSDs, there is a spectrum between two extremes. One extreme is assigning channels into a zone as many as possible to improve intra-parallelism in a zone. The other extreme is assigning a different channel (or channels) into a zone to support isolation. Real ZNS SSDs lie between these two extremes. In actuality, due to the limited number of channels, sharing channels among zones is inevitable, which causes the non-linear scalability. But it is clear that our scheme can reduce the interference among zones by reducing the garbage collection overhead.

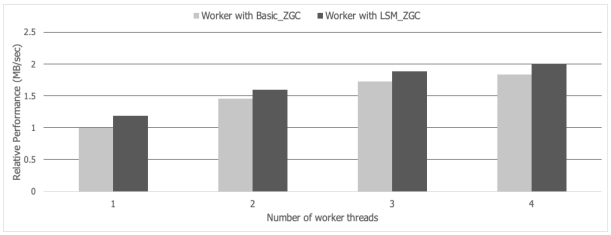

Figure 9: Performance comparison using different number of threads

One beneficial feature of LSM_ZGC is that it gives a positive effect on generating sequential writes. It writes new written data sequentially into a zone whose state is C0_zone. In addition, reclaimed data managed by LSM_ZGC is also written sequentially. Hence, our proposal is in accordance with the sequential write constraint required by ZNS SSDs. However, guaranteeing the constraint is a complex problem related to not only allocation policy but also caching and I/O scheduling. Exploring this issue is left as a future work.

5 Conclusion

This paper proposes a new zone garbage collection scheme, called LSM_ZGC. Our contributions include 1) devising a new LSM-style garbage collection scheme, 2) providing real implementation based evaluation results and 3) raising several issues to address the ZNS SSD features such as zone size and sequential write constraint.

There are two directions for future research. The first one is implementing our scheme in a real file system. We are currently extending F2FS on our ZNS SSD prototype to integrate LSM_ZGC and to comply with the sequential write constraint for not only data but also metadata such as checkpoint and NAT (Node Address Table) [16]. The second direction is evaluating LSM_ZGC under diverse workloads with different hot/cold ratio, data size, initial placement and classification policies [12].

Acknowledgement

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by SK Hynix and by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. 2019R1F1A1062284).

6 Discussion

Our proposal introduces several interesting topics for future studies. They can be classified into three categories. The first one is about what features a file system for ZNS SSDs are required? How to change the conventional caching and I/O scheduling mechanism to guarantee the sequential write constraint? How to identify different workloads and distribute them depending on their characteristics? How to make use of the nameless writes [25] to support the zone append command in ZNS SSDs [6]?

The second category is regarding which applications can obtain the benefits of ZNS SSDs. For example, are applications that have the property of write once and read multiple times suitable for ZNS SSDs? A key value store is considered as a good candidate since it has a sequential write pattern [18, 24]. However, there still exist several issues such as a new compaction algorithm and how to allocate levels into zones. In addition, how to integrate ZNS SSDs into a distributed storage backend, such as Ceph, is an open question [5].

The final category is about the internal structure of ZNS SSDs. How to balance the tradeoff between parallelism in a zone and among zones. It affects greatly between performance and isolation. The zone size and the number of zones that can be opened concurrently also impact on designing system software for ZNS SSDs. More fundamental question is which one is better? Either managing SSDs in a host [7, 17] or processing in storage [10, 21]. We would like to ask for feedback about these topics.

References

[1] NVMe Management Command Line Interface. https://github.com/linux-nvme/nvme-cli.

[2] NVMe Zoned Namespaces. https://zonedstorage.io/introduction/zns/.

[3] SK hynix demonstrates industry's first ZNS-based SSD solution for data centers. https://news.skhynix.com/sk-hynix-demonstrates-industrys-first-zns-based-ssd-solution-for-data-centers-2/.

[4] Western Digital Champions Zoned Storage, An open initiative to redefine data centers for zettabyte scale. https://www.westerndigital.com/company/newsroom/press-releases/2019/2019-06-11-western-digital-champions-zoned-storage.

[5] AGHAYEV, A., WEIL, S., NELSON, M., GANGER, G. R., KUCHNIK, M., AND AMVROSIADIS, G. File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution. In *SOSP* (2019).

[6] BJØRLING, M. From Open-Channel SSDs to Zoned Namespaces. In *USENIX VAULT* (2019).

[7] BJØRLING, M., GONZALEZ, J., AND BONNET, P. The Linux Open-Channel SSD Subsystem. In *FAST* (2017).

[8] CAO, Z., DONG, S., VEMURI, S., AND DU, D. H. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *USENIX FAST* (2020).

[9] CHUNG, W. Benefits of ZNS in Datacenter Storage Systems. In *Flash memory summit* (2019).

[10] GU, B., YOON, A. S., BAE, D.-H., JO, I., LEE, J., YOON, J., KANG, J.-U., KWON, M., YOON, C., CHO, S., JEONG, J., AND CHANG, D. Biscuit: A framework for near-data processing of big data workloads. In *ISCA* (2016).

[11] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS* (2009).

[12] HSIEH, J.-W., , KUO, T.-W., AND CHANG, L.-P. Efficient identification of hot data for flash memory storage systems. *Transactions on Storage* (2006).

[13] HWANG, J.-Y. Towards even lower total cost of ownership of data center it infrastructure. http://www.sigfast.or.kr/nvramos/nvramos19/presentation/nvramos19-samsung.pdf, 2019.

[14] KIM, J., LIM, K., JUNG, Y., LEE, S., MIN, C., AND NOH, S. H. Alleviating garbage collection interference through spatial separation in all flash arrays. In *USENIX ATC* (2019).

[15] KIM, T., HONG, D., HAHN, S. S., CHUN, M., LEE, S., HWANG, J., LEE, J., AND KIM, J. Fully automatic stream management for multi-streamed ssds using program contexts. In *FAST* (2019).

[16] LEE, C., SIM, D., HWANG, J.-Y., AND CHO, S. F2FS: A new file system for flash storage. In *FAST* (2015).

[17] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., AND ARVIND. Application-managed flash. In *FAST* (2016).

[18] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in SSD-conscious storage. In *FAST* (2016).

[19] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica* (1996).

[20] PICOLI, I. L., HEDAM, N., BONNET, P., AND TÖZÜN, P. Open-Channel SSD (What is it good for). In *CIDR* (2020).

[21] PITCHUMANI, R., AND KEE, Y.-S. Hybrid data reliability for emerging key-value storage devices. In *FAST* (2020).

[22] ROSENBLUM, M., AND OUSTERHOUT, J. K. The Design and Implementation of a Log-structured File System. *Transactions on Computer Systems* (1992).

[23] WU, F., YANG, M.-C., FAN, Z., ZHANG, B., GE, X., AND DU, D. H. Evaluating host aware SMR drives. In *Hotstorage* (2016).

[24] YAO, T., WAN, J., HUANG, P., ZHANG, Y., LIU, Z., XIE, C., AND HE, X. GearDB: A GC-free key-value store on HM-SMR drives with Gear Compaction. In *FAST* (2019).

[25] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based SSDs with nameless writes. In *FAST* (2012).