A Free-Space Adaptive Runtime Zone-Reset Algorithm for Enhanced ZNS Efficiency

Sungjin Byeon[1], Joseph Ro[1], Safdar Jamil[1], Jeong-Uk Kang[2], and Youngjae Kim[1]
[1]Sogang University, [2]Samsung Electronics Co.
{sjbyeon, josephro12, safdar, youkim}@sogang.ac.kr, ju.kang@samsung.com

ABSTRACT

While the state-of-the-art runtime zone-reset algorithm of the ZenFS in RocksDB is optimized for the performance of Zone Namespace (ZNS) SSDs, it does not take into account the lifetime constraint of ZNS SSDs. To address this issue, we present FAR, a Free-space Adaptive Runtime Zone-Reset algorithm for ZenFS, which dynamically adjusts the frequency of runtime zone-reset calls based on the available free-space in the ZNS SSD. We developed FAR with the ZenFS of RocksDB using a ZNS SSD prototype based on the Cosmos+ OpenSSD platform and compared it with the state-of-the-art runtime zone-reset algorithm used in ZenFS. Our extensive evaluations demonstrate that FAR improves the lifetime of ZNS SSD by 2× without compromising performance.

CCS CONCEPTS

• **Information systems → Flash memory; Filesystem; Key-value stores**.

KEYWORDS

File System, Key-Value Database, Solid State Drive

ACM Reference Format:
Sungjin Byeon[1], Joseph Ro[1], Safdar Jamil[1], Jeong-Uk Kang[2], and Youngjae Kim[1]. 2023. A Free-Space Adaptive Runtime Zone-Reset Algorithm for Enhanced ZNS Efficiency. In *15th ACM Workshop on Hot Topics in Storage and File Systems* (*HotStorage '23*), *July 9, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3599691.3603410

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotStorage '23, July 9, 2023, Boston, MA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0224-2/23/07…$15.00
https://doi.org/10.1145/3599691.3603410

1 INTRODUCTION

ZNS SSDs [6, 10] use a "zone" that enforces sequential writes and disallows overwrites. [4, 5, 14, 16, 27]. This removes the need for in-device Garbage Collection (GC) in the Flash Translation Layer (FTL), enhancing I/O performance [6, 14, 16, 20, 27]. However, when free-space is needed in the ZNS SSD, applications that access ZNS SSDs must perform Zone Cleaning (ZC) [7, 14, 22, 23]. During ZC, a victim zone is selected for erasure and the valid data in the victim zone is safely moved to a free zone, after which the victim zone is erased [7, 26]. This movement of valid data creates write amplification (WA) problems. [17]

Meanwhile, Log-Structured Merge-tree (LSM-tree)-based Key-value stores such as RocksDB [12] and LevelDB [13] are typical applications suitable for ZNS SSDs because they only allow sequential writing through append-only [5, 24, 28]. However, to use ZNS SSDs, such key-value stores require middleware to manage stored data. So, RocksDB uses a user-level file system, called ZenFS [9]. ZenFS adopts LSM-tree-aware zone allocation strategies [6, 22] to minimize the WA problem explained earlier with the ZNS SSD by writing data with the same lifetime to the same zone. This helps to reduce the WA overhead during ZC by minimizing the amount of valid data present in the victim zone. However, it is not possible to completely eliminate the WA overhead associated with ZC.

Moreover, during ZC, the foreground I/Os being handled by RocksDB may be blocked due to interference with ZC, resulting in decreased throughput. To minimize performance degradation caused by ZC, ZenFS employs a runtime zone-reset, which triggers zone-reset to a zone with only invalid data during runtime. This runtime zone-reset can free up space without any data copying, which was the main cause of the WA problem during ZC. Moreover, the runtime zone-reset can free up space before ZC, delaying the need for ZC calls and ultimately reducing the I/O blocking issues of the foreground I/Os that occur during ZC. However, the current state-of-the-art runtime zone-reset implementation in ZenFS uses a greedy approach, resetting a zone if all data before the zone's write pointer (WP) is invalid, regardless of the WP's location. This greedy approach can potentially shorten the device's lifetime due to excessive zone-resets.

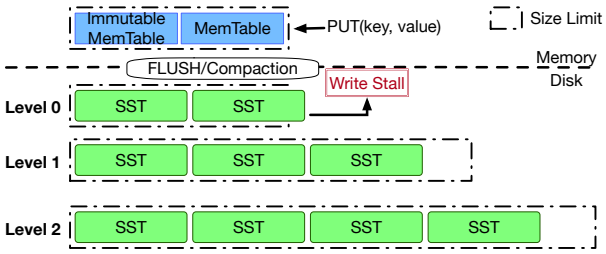


Figure 1: Description of RocksDB's LSM-tree.

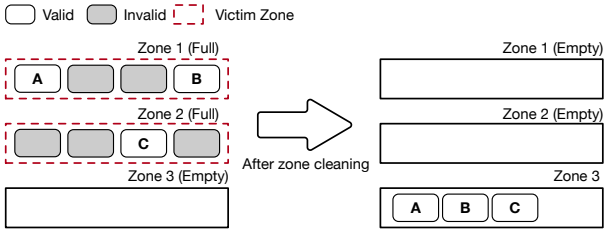


Figure 2: Description of Zone Cleaning in ZenFS.

Thus, in this paper, we propose FAR, a free-space adaptive runtime zone-reset algorithm that dynamically adjusts the frequency of runtime zone-reset calls based on the available free-space in the ZNS SSD. Consequently, FAR achieves a balance between performance and device lifetime, effectively increasing longevity of the device without compromising performance. For evaluations, we developed the ZNS SSD prototype on the Cosmos+ OpenSSD platform [1] and implemented FAR by modifying ZenFS v2.1 in RocksDB v7.4. Our extensive evaluations of FAR confirm that it improves the device's lifetime by an average of 50% compared to ZenFS's current runtime zone-reset implementation using a variety of workloads with minimal performance degradation.

2 BACKGROUND

2.1 Zone Namespace SSD

The NVMe Zoned Namespace (ZNS) [6, 10] uses a zone interface for flash-based SSDs [4, 14–16, 27]. Each ZNS SSD zone is composed of multiple NAND erase blocks, which are directly exposed to the host machine through the zone interface. ZNS enforces only sequential writes to each zone with reset commands, eliminating the need for GC in the FTL in the SSD. However, ZNS requires modifications to the software running on the host due to the change made on the SSD hardware [6, 7, 14, 27]. For example, applications that use the ZNS SSD must perform data placement by selecting zones when writing data. Applications must be in charge of free-space reclamation [6, 14, 27], explicitly erasing zones instead of relying on the FTL of the SSD. The free-space reclamation process involves executing zone-reset. The zone-reset entails erasing blocks, decreasing the Program/Erase (P/E) cycle [11] of cells in the NAND flash memory. During the zone-reset, valid data in the zone to be reset must be copied into a free zone before the reset, leading to increased I/O blocking time.

2.2 Log-Structured Merge-Tree

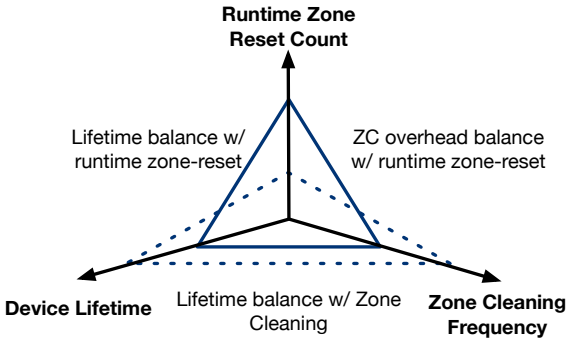
LSM-tree [24] is a write optimized data structure and have been widely adopted in various key-value stores such as RocksDB [12], LevelDB [13], and MongoDB [18]. Figure 1 describes the architecture and operation of RocksDB's LSM-tree. LSM-tree is comprised of in-memory (MemTable and Immutable MemTable) and persistent, Sorted String Table (SSTable) components in an hierarchical levels of increasing sizes. The in-memory components buffer the incoming PUT request from application and later flushed, in sequential order, to persistent storage in SSTable format at Level 0.

Once level i reaches a certain size threshold, a compaction operation is triggered. The compaction operation selects a victim SSTable file from level i and the corresponding SSTable files from level $i+1$ with overlapping key ranges and perform merge-sort operation on the selected SSTable files to write a new SSTable file at level $i+1$. The compaction operation maintains the structural constraint and lookup performance of LSM-tree. However, compaction operation can lead to write stall problem when level 0 reaches size threshold and not able accommodate incoming in-memory components. The write stalls block the foreground I/Os at application level, thus resulting in high latency spikes and decreased overall throughput [3, 8, 29, 30].

2.3 Zone Management Middleware

Applications that use the ZNS SSD require middleware that can manage the zones of the ZNS SSD. ZenFS, a user-level file system used as a middleware for RocksDB, is responsible for allocation space for SSTables in the zones and reclaiming zones with invalid data.

Zone Allocation: ZenFS employs Lifetime-Based Zone Allocation (LIZA) when allocating space for new SSTable in the zone, which is to reduce the data copying overhead during ZC. LIZA assigns a unique lifetime hint value to each level of the LSM-tree which indicates the lifetime of SSTables in that level. LIZA uses this lifetime hint information to allocate zones for SSTables such that SSTables with similar lifetime hints are grouped and placed in the same zone. LIZA, consequently, allows SSTables in the same zone more likely to be erased together in the future when ZC is performed, eventually minimizing the valid data copy overhead during ZC. Furthermore, Compaction-Aware Zone Allocation (CAZA) [22] is another study with the same goal as LIZA. CAZA allocates SSTables that participate in compaction to the same zone [22].



	Runtime Zone Reset Count	ZC Overhead	Device Lifetime	I/O Blocking Time
Solid Line	High	Low	Bad	Low
Dotted Line	Low	High	Good	High

Figure 3: An analysis of the balance between runtime zone-reset and ZC in terms of ZC overhead and device's lifetime.

Zone Cleaning: ZC is similar to segment cleaning in log-structured file systems [21, 25]. Figure 2 illustrates the process of ZC in ZenFS. When there is no available space for writing, ZC is triggered to produce free-space/zones by erasing invalid data in the zone through following steps:

(1) First it selects a victim zone to be erased.
(2) All valid data within the victim zone are copied to a free zone. This process is necessary to ensure that valid data is safely stored during the ZC process.
(3) ZenFS sends a zone-reset command to the ZNS SSD to erase the victim zone.

In ZenFS, ZC is triggered in background when the free-space falls below 20% of the entire space.

As explained, ZC entails valid data copying, which is the main cause of blocking foreground I/O and lowering I/O performance. Therefore, several previous works [19, 22] have focused on minimizing the data copying overhead during ZC by placing SSTables to be erased in the same zone. However, the valid data copy overhead during ZC cannot be completely eliminated. Thus, ZenFS adopted the runtime zone-reset to alleviate the performance issues caused during ZC.

3 RUNTIME ZONE-RESET ALGORITHM

3.1 Eager Zone-Reset Algorithm

The state-of-the-art runtime zone-reset algorithm implemented in ZenFS of RocksDB checks all zones and find zones with invalid data up to the Writer Pointer (WP) whenever a file or directory is deleted. If such zones exist, ZenFS performs a zone-reset on them, without paying attention to WP. This algorithm is referred to as the eager runtime zone-reset (EZReset), as it performs zone-resets during runtime rather than during ZC.

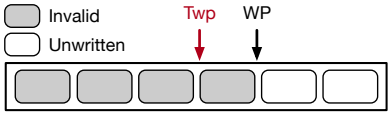


Figure 4: Description of LZReset with T_{WP}.

EZReset has the advantage of reducing the frequency of ZC calls and minimizing foreground I/O's blocking times that occur during ZC. This is because it frees up space by performing a zone-reset to zones before ZC calls, ultimately reducing the total number of ZC calls.

However, EZReset may call zone-resets excessively, even when it is unnecessary (erase blocks in the zones excessively), potentially increasing the number of P/E cycles, leading to device failure. To mitigate this issue, EZReset should be carefully optimized to ensure that it calls zone-resets only when necessary, thus minimizing the impact on device's lifetime.

Figure 3 shows the trade-off between runtime zone-reset and ZC in terms of data copying overhead during ZC and device's lifetime. If the number of runtime zone-reset calls increase, the number of ZC call decrease, thus reducing the data copying overhead during ZC and minimizing foreground I/O blocking time that may occur during ZC (as shown by the solid lines in the Figure 3). However, excessive zone-reset calls have adverse effects which decreases device's lifetime, meanwhile fewer calls have favorable effects.

3.2 Free-Space Adaptive Runtime Zone-Reset Algorithm

Unlike EZReset, Lazy runtime Zone-reset (LZReset) is an algorithm that controls zone-reset calls according to the position of WP in runtime.

Figure 4 describes the working flow of LZReset where it maintains a WP threshold (T_{WP}), which determines when to call for runtime zone-reset. When WP in a zone is greater than or equal to T_{WP} and all the data before WP is invalid, LZReset calls zone-reset. Therefore, LZReset utilizes T_{WP} to control the frequency of runtime zone-reset calls, striking a balance between the performance degradation caused by the data copying overhead during ZC and the device's lifetime impacted by excessive runtime zone-reset calls.

However, an inappropriate setting for T_{WP} can lead to negative effects, so it needs to be carefully tuned. If T_{WP} is set too large, the frequency of calling runtime zone-reset is reduced, which does not help minimize the performance degradation caused by the aforementioned ZC. Conversely,

if T_{WP} is set too small, it behaves similarly to EZReset and inherits its advantages and disadvantages. Thus, our goal is to develop an algorithm to set T_{WP} appropriately to minimize the lifetime problem associated with EZReset and reduce the overhead caused by ZC.

To achieve this goal, we propose a Free-space-Adaptive Runtime zone-reset (FAR) algorithm that dynamically sets T_{WP} based on the remaining free space on the ZNS SSD. The idea behind FAR is that the blocking time of foreground I/O during ZC varies with the percentage of free space remaining on the ZNS SSD. As the free-space ratio decreases, the blocking time of foreground I/O increases during ZC, and vice versa. FAR is a variant of LZReset as it controls runtime zone-reset calls dynamically based on the amount of free space remaining on the device.

To implement FAR, we use two variables: the ratio of free space to the maximum device capacity (R_{free}) and a turning point (T). R_{free} indicates the amount of remaining free space in the ZNS SSD, with 0.0 indicating no free space and 1.0 indicating all free space. T is a threshold value for the free-space ratio, and FAR actively utilizes T for its operations.

For instance, consider the case where T is set to 0.8 in FAR. When the free-space ratio (R_{free}) is higher than 0.8, FAR operates similar to LZReset with a fixed T_{WP} value set to the end of a zone. However, if R_{free} is lower than 0.8, FAR gradually moves T_{WP} from the end of the zone to the beginning based on the R_{free}. In summary, the setting of T_{WP} depends on the value of R_{free} as follows.

$$T_{WP} = \begin{cases} The\ end\ of\ a\ zone, & \text{if } R_{free} \geq T \\ func(R_{free}), & \text{otherwise} \end{cases}$$

In the above equation, $func$ is a function of free-space (R_{free}) remaining on the device. We consider Linear, Log and Exponential functions for $func$. The Linear function has a fixed change rate of T according to R_{free}, while the Log function has a small change of T at first according to R_{free}, but then a very large change later on. Exponential function has opposite tendency compared to Log function, it has a large change of R_{free} at first, small change later on.

4 EVALUATION

4.1 Experimental Setup

We developed the ZNS SSD prototype on the Cosmos+ OpenSSD platform [1] which has Xilinx Zynq-7000 SoC, 1 TB NAND, and 1 GB DRAM. The zone size of the ZNS SSD prototype was set to 512 MB, and for experimental convenience, we configured the device with a total capacity of 20 GB (40 zones in total). To evaluate on the ZNS SSD prototype, we connected it to a server equipped with an Intel(R) Core i7-4790 3.60Ghz CPU (8 cores) and 16 GB of memory, running Linux Kernel v5.18.0.

We implemented FAR[1] by modifying ZenFS v2.1 in RocksDB v7.4. ZenFS used a greedy ZC algorithm, in which zones with the highest levels of invalid data are selected as victims for ZC. We configured RocksDB setting with a size limit of 64 MB for MemTables and L0 SSTables. And we set the size limit of each level starting from 256 MB and increases by a factor of ten for each subsequent level.

We compared three different runtime zone-reset algorithms for our evaluation, namely:

- EZReset: Default runtime zone-reset in ZenFS
- LZReset: Lazy runtime zone-reset with T_{WP}=1.0
- FAR(T, $func$): free-space adaptive zone-reset algorithm proposed in this paper.

Workloads: We used RocksDB's db_bench. We first fill the RockSDB using db_bench with "fillrandom" option for workloads of 16 B key and 1 KB value pairs. We used three workloads, small (9GB), medium (12GB), and large (15GB), depending on the total amount of writes performed. For writes, all workloads used the "fillrandom" option of db_bench. An increase in write amount means an increase in the number of ZC calls in ZenFS. We run multiple experiments for each setting and show the mean and standard deviation in error bars.

4.2 ZNS SSD Performance Analysis

We compared the device-level performance of the SSD and ZNS SSD implementation of the Cosmos+ OpenSSD platform using a Fio benchmark [2] for various I/O pattern workloads excluding the random write I/O pattern workload (RanWrite), as ZNS SSD only allows sequential writes. We set I/O size as 16 K with I/O queue depth as 4, total size of file as 16 GB. The greedy FTL was employed for the conventional SSD prototype [1].

Table 1: Evaluation of the performance of ZNS SSD and conventional SSD prototype implementations.

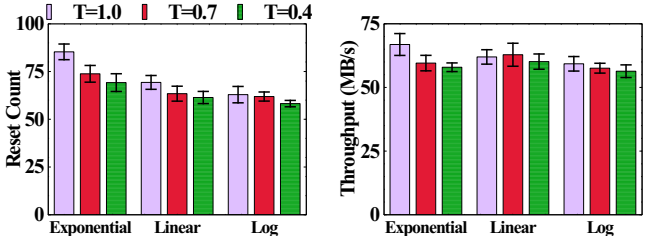
(MB/s)	SeqWrite	SeqRead	RanRead
ZNS SSD	401	394	393
Conventional SSD	403	391	394

Table 1 shows the results. As expected, there is little performance difference between the two device implementations. This depicts that our prototype of ZNS SSD is stable.

4.3 Evaluation of FAR

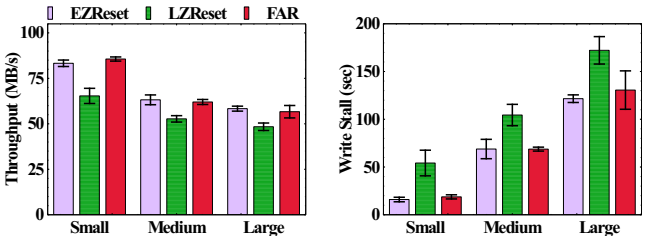
To compare the efficiency of FAR, we evaluated throughput and total zone reset count for two dependent variables (T

[1]https://github.com/lass-lab/FAR-ACMHOTSTORAGE2023



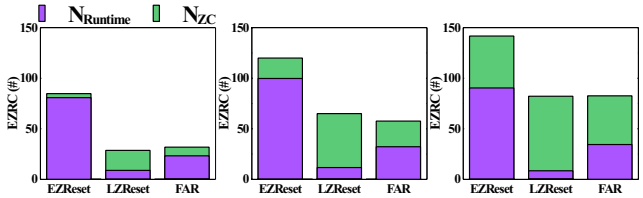
(a) Zone-Reset Count (b) Throughput

Figure 5: Throughput and zone-reset count comparison of FAR with different settings of T and $func$.



(a) Throughput (b) Write Stall

Figure 6: Comparison of throughput and aggregate write stall time for EZReset, LZReset, and FAR.



(a) Small (b) Medium (c) Large

Figure 7: Comparison of lifetime of three algorithms.

and $func$). We used 1.0, 0.7, and 0.4 for T and exponential, linear and logarithmic functions for $func$. When the free space reaches the turning point T, the value of T_{WP} gradually decreases from 1.0 to 0.0. The rate of decrease varies for each function type, such as exponential, linear, and logarithmic, until the available space reaches zero. Figure 5 shows the results of different settings of FAR using the medium workload as the insights with small and large workload are also similar.

Figure 5(a) shows the zone-reset count with different settings. As explained in Section 3.2, reducing the turning point (T) generally leads to fewer ZC calls. However, if T becomes too small (approaching 0.0), the algorithm operates like LZReset with T_{WP} set to the end of a zone, which can increase the burden on ZC calls by rarely calling runtime zone-resets. Thus, reducing the turning point too much is not desirable from the point of performance (referring to Figure 5(b)). Therefore, we consider $T = 0.7$ as the most appropriate turning point, taking into account the trade-off between lifetime and performance.

Figure 5(b) shows that the throughput decreases as the value of T decreases in all functions. Among them, $func$=Exponential showed the high throughput (61.45 MB/s) because it actively considers free space and sets the T_{WP} value low initially, leading to similar operation to EZRest while increasing ZC calls. On the other hand, $func$=Log sets T_{WP} to the end of a zone initially and does not actively consider free space, resulting in the lowest throughput overall (57.54 MB/s), reducing the number of calls to runtime zone-reset while increasing ZC overhead. Meanwhile, $func$=Linear has a zone-reset count comparable to $func$=Log but exhibits similar throughput/performance to $func$=Exponential (61.65 MB/s)

Therefore, we consider FAR (T=0.7, $func$=Linear) the optimal setting for guaranteeing lifetime in our settings due to its low zone-reset count and performance close to EZRest. Thus, we use FAR(T=0.7, $func$=Linear) for the rest of the experiments.

4.4 Performance versus Lifetime

We evaluated the throughput and aggregate write stall time of RocksDB using three workloads for EZReset, LZReset (T_{WP}=1.0), and FAR (0.7, Linear). In RocksDB, a write stall can happen when the memory components cannot be flushed to L0 because L0 is full. Write stalls can be even worse if a write stall occurs while ZenFS is running ZC.

Figure 6(a) shows that FAR is almost the same throughput as EZReset regardless of the workloads, whereas LZReset is the worst throughput. LZReset show higher write stall time than EZReset and FAR, as shown in Figure 6(b). Fundamentally, EZReset and FAR avoid write stalls as much as possible by appropriately making free zones before ZC calls.

Next, we evaluated the impact of the runtime zone-reset algorithm on the device's lifetime. In order to evaluate it, we defined a lifetime estimation metric such as Equation 1.

$$EZRC = N_{Runtime} + N_{ZC} \quad (1)$$

Effective Zone-Reset Count (EZRC) is represented as the sum of total runtime zone-reset count ($N_{Runtime}$) and total zone-reset count that occurs during ZC (N_{ZC}).

Figure 7 shows the breakdown of EZRC into $N_{Runtime}$ and N_{ZC} for three algorithms. Overall, EZReset performs greedy runtime zone-reset, resulting in a higher EZRC value than the other two algorithms, irrespective of the workload. Moreover, EZReset has a higher $N_{Runtime}$ value, accounting for over 60% of the EZRC. In contrast, LZReset has a lower $N_{Runtime}$ value of the EZRC. Importantly, FAR properly controls the calls for runtime zone-reset, resulting in a similar

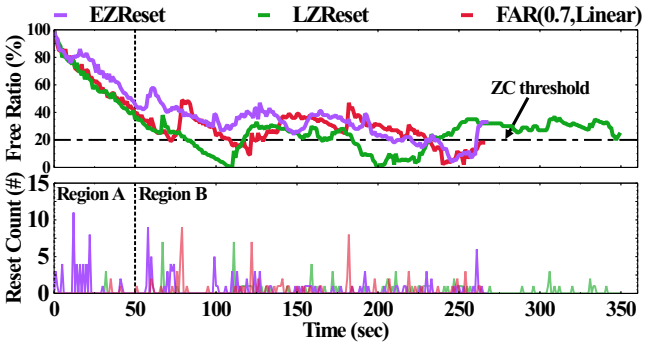


Figure 8: Microscopic analysis of EZReset, LZReset, and FAR with a large workload.

EZRC value with LZReset. Therefore, FAR achieves almost similar device's lifetime as LZReset.

Consequently, FAR maintains similar throughput as EZReset while ensuring similar device's lifetime as LZReset.

4.5 Microscopic Analysis

Figure 8 shows the results of a time-series analysis for the remaining free-space, frequency of zone-reset calls, and execution time for a large workload.

Figure 8 (top) shows that LZReset has approximately 30% longer run time than the other two algorithms, while the other two algorithms show similar execution times. This phenomenon can be explained in relation to the number of zone-reset calls. Figure 8 (bottom) shows total zone-reset counts with time. Referring to the [0:50] time range (Region 'A'), EZReset calls runtime zone-reset very frequently (refer to purple spikes). On the other hand, LZReset rarely calls runtime zone-reset as it relies mostly on ZC for free space reclamation. Thus, LZReset shows gradual free space lost than EZReset. On the other hand, FAR initially runs like LZReset (because of T=0.7) (Region 'A'), and after that, it runs like EZreset. Thus, FAR initially loses free-space similar to LZRest, but after 50 seconds behaves similar to EZReset (Region 'B').

To compare the average WP values of each algorithm, we measured the average value of the WP offset relative to a zone size (R_{WP}) during the workload execution time.

Table 2: Comparison of the ratio of WP to a zone size for various runtime zone-reset algorithms.

Algorithm	EZReset	LZReset	FAR
R_{WP}	0.55	1	0.92

Table 2 shows the result. As expected, EZReset's R_{WP} value is the smallest among the algorithms. On the other hand, the R_{WP} value of FAR is between the values of EZReset and LZReset.

5 CONCLUSION

In this paper, we identified that the current implementation of ZenFS's eager runtime zone-reset algorithm can have a negative impact on device's lifetime. Thus, we proposed a free-space adaptive runtime zone-reset (FAR) algorithm that balances application's performance and device's lifetime. Extensive evaluation shows that FAR maintains a high level of performance compared to the ZenFS's runtime zone-reset algorithm while significantly improving device's lifetime by a factor of 2.

ACKNOWLEDGMENTS

This work was funded in part by Samsung Electronics Co., Ltd (IO221014-02908-01), and in part by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. NRF-2021R1A2C2014386). Y. Kim is the corresponding author.

REFERENCES

[1] 2017. Cosmos+ OpenSSD Platform. http://www.openssd.io/.
[2] Jens Axboe. 2022. Flexible I/O tester. https://github.com/axboe/fio
[3] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference* (*USENIX ATC 19*). 753–766.
[4] Matias Bjørling. 2019. From Open-channel SSDs to Zoned Namespaces. In *Linux Storage and Filesystems Conference* (*Vault '19*), Vol. 1.
[5] Matias Bjørling. 2020. Zone Append: A New Way of Writing to Zoned Storage. In *Linux Storage and Filesystems Conference* (*Vault '20*), Vol. 1.
[6] 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 *Proceedings of the USENIX Annual Technical Conference* (*ATC '21*). 689–703.
[7] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. 2020. A New LSM-style Garbage Collection Scheme for ZNS SSDs. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems* (*HotStorage '20*).
[8] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference* (*ATC '20*). 49–63.
[9] Western Digital Corporation. 2022. ZenFS. https://github.com/westerndigitalcorporation/zenfs
[10] Western Digital Corporation. 2022. Zoned Storage. https://zonedstorage.io/docs/introduction/zoned-storage
[11] Peter Desnoyers. 2022. Empirical Evaluation of NAND Flash Memory Performance. In *ACM SIGOPS Operating Systems Review*. Northeastern University, 50–54.
[12] Facebook. 2022. RocksDB. https://github.com/facebook/rocksdb
[13] Google. 2021. LevelDB. https://github.com/google/leveldb
[14] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting

In-Storage Zone Compaction. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI '21*). 147–162.

[15] Miryeong Kwon Myoungsoo Jung Hanyeoreum Bae, Jiseon Kim. 2022. What you can't forget: exploiting parallelism for zoned namespaces. In *Proceedings of the 14th USENIX Workshop on Hot Topics in Storage and File Systems* (*HotStorage '22*).

[16] Hans Holmberg. 2020. ZenFS, Zones and RocksDB - Who Likes to Take out the Garbage Anyway? https://snia.org/sites/default/files/SDC/2020/074-Holmberg-ZenFS-Zones-and-RocksDB.pdf

[17] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. 2009. Write Amplification Analysis in Flash-based Solid State Drives. In *Proceedings of the ACM International Systems and Storage Conferences* (*SYSTOR '09*). 1–9.

[18] MongoDB Inc. 2022. MongoDB. https://github.com/mongodb/mongo

[19] Jeeyoon Jung and Dongkun Shin. 2022. Lifetime-Leveling LSM-Tree Compaction for ZNS SSD. In *Proceedings of the 14th USENIX Workshop on Hot Topics in Storage and File Systems* (*HotStorage '22*).

[20] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and Youjip Won. 2022. IPLFS: Log-Structured File System without Garbage Collection. In *In Proceedings of the 2022 USENIX Annual Technical Conference* (*ATC 22*). 739–754.

[21] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (*FAST '15*). 273–286.

[22] 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 USENIX Workshop on Hot Topics in Storage and File Systems* (*HotStorage '22*).

[23] Animesh Trivedi Nick Tehrany. 2022. Understanding NVMe Zoned Namespace (ZNS) Flash SSD Storage Devices. (June 2022). https://arxiv.org/abs/2206.01547

[24] Patrick O' Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O' Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[25] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-structured File System. In *ACM Transactions on Computer Systems, Volume 10, Issue 1* (*Transactions on Computer Systems*). 26–52.

[26] Reza Salkhordeh, Kevin Kremer, Lars Nagel, Dennis Maisenbacher, Hans Holmberg, Matias Bjørling, and André Brinkmann. 2021. Constant Time Garbage Collection in SSDs. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage* (*NAS '21*). 1–9.

[27] Theano Stavrinos, Daniel S Berger, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Don't be a blockhead: Zoned namespaces make work on conventional SSDs obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (*HotOS '21*). 144–151.

[28] Denghui Wu, Biyong Liu, Wei Zhao, and Wei Tong. 2022. ZNSKV: Reducing Data Migration in LSMT-Based KV Stores on ZNS SSDs. In *Proceedings of the IEEE 40th International Conference on Computer Design* (*ICCD '22*). 411–414.

[29] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Balancing Garbage Collection vs I/O Amplification using hybrid Key-Value Placement in LSM-based Key-Value Stores. arXiv:cs.DB/2106.03840

[30] Giorgos Xanthakis, Giorgos Saloustros, Nikos Batsaras, Anastasios Papagiannis, and Angelos Bilas. 2021. Parallax: Hybrid Key-Value Placement in LSM-Based Key-Value Stores (*SoCC '21*). 305–318.