Journal of Multimedia Information System VOL. 9, NO. 4, December 2022 (pp. 261-268): ISSN 2383-7632 (Online)
https://doi.org/10.33851/JMIS.2022.9.4.261

Separating the File System Journal to Reduce Write Amplification of Garbage Collection on ZNS SSDs

Young-in Choi[1], Sungyong Ahn[1*]

Abstract

Solid-State Drives (SSDs), despite their outstanding I/O performance, suffer from the garbage collection overhead incurred by out-place update scheme that addresses the erase-before-write constraint of NAND flash memory. The Zoned Namespace SSD (ZNS SSD) is a state-of-the-art storage technology that divides NAND flash memory space within an SSD into zones and allows the host to directly assign certain zone for a write request. Therefore, the ZNS SSD can minimize the garbage collection overhead by allocating data with similar update patterns to the same zone. However, legacy journaling file systems still cannot actively exploit the advantages of ZNS SSDs, and journal data and file data with obviously different update patterns are written together in the same zone. In this paper, we propose new file system journal placement scheme that separates journal data into a dedicated zone to minimize write amplification due to garbage collection, noting the characteristics of journal data in the Ext4 file system. The proposed scheme is implemented in Linux kernel and carefully evaluated. The evaluation results show that our scheme removes the unnecessary copying of journal data during garbage collection of ZNS SSD and reduces the valid page copies caused by garbage collection by up to 26.8%.

Key Words: Ext4 File System, Zoned Namespace SSD, NAND Flash Memory, Journaling, Dm-Zoned.

I. INTRODUCTION

Recently, NAND flash memory based Solid-State Drives (SSDs) are widely used as a storage device because of its outstanding performance compared to Hard-Disk Drives (HDDs). However, SSDs suffer from physical constraints of NAND flash memory such as erase-before-write and limited erase count. To address these limitations and support identical block-based interface with HDDs, the conventional SSDs employ an internal software, Flash Translation Layer (FTL) which supports out-place update and garbage collection. The garbage collection reserves free space by erasing invalid data incurred by out-place update. Here, because the erase unit (NAND block) of NAND flash memory is larger than the read/write unit (NAND page), valid pages of an erase block must be copied to another block during garbage collection. As a result, write amplification is induced, shortening the lifespan of the SSD. Therefore, many studies are conducted to alleviate the garbage collection overhead by writing data with similar update pattern to the same NAND block to reduce the number of valid page copies during garbage collection [18-19]. However, because of the inherent limitation of the traditional block interface that host-side information cannot be delivered to the SSD, it is hard to understand the data characteristic inside the SSD. As a result, new storage interfaces such as Open-channel SSD [13] and Zoned Namespace (ZNS)[5] are proposed for the host-managed SSD where the host-side FTL directly manages data placement and various SSD internal operations instead of a firmware-level FTL.

In particular, the zone-based interface of ZNS is standardized in NVMe specification [20] and is expected to replace the conventional block-based interface. In the ZNS SSD, NAND flash memory space is divided into zones which have sequential write constraint and can be directly specified for read/write request by the host. Therefore, the host can allocate zones separately for different I/O streams to reduce performance interference between them and garbage collection overhead. On the other hand, legacy file systems still do not fully utilize the characteristics of ZNS SSDs.

Especially, in this paper, we focused on the journaling system of the journaling file system including Ext4 [1]. When a write request is issued on the journaling file systems, a log of the write request, called as the journal, is permanently recorded to a separate journal area before the requested data is written in the storage device. In case of sud-

Manuscript received December 20, 2022; Revised December 24, 2022; Accepted December 25, 2022. (ID No. JMIS-22M-12-052)
Corresponding Author (*): Sungyong Ahn, +82-51-510-2422, sungyong.ahn@pusan.ac.kr
[1]School of Computer Science and Engineering, Pusan National University, Busan, Korea, everimind4@gmail.com, sungyong.ahn@pusan.ac.kr

den power loss or system crash, the journal is used to recover the lost data and maintain the consistency of the file system [2]. However, after a write request is successfully completed in the storage device, the associated journal is rarely used. Moreover, outdated journals will be overwritten by new journals in the near future. Therefore, journals can be said to have different access patterns from file data.

Therefore, the garbage collection overhead of ZNS SSDs can be reduced by separating journal data and general file data into different zones. However, dm-zoned [6], device mapper of Linux kernel which emulates ZNS SSDs to block storage for legacy filesystems, does not separate journal data. In this paper, we propose a simple and effective placement method to reduce the garbage collection overhead of ZNS SSDs by separating journal data and file data physically for journaling file system.

In this paper, we propose a simple and effective data deployment method to separate journal data into journal dedicated zones by exploiting information about journal data delivered from Ext4 file system and JBD2 daemons. This method was implemented by modifying dm-zoned and Ext4 file system in Linux kernel. In the modified version of dm-zoned, a journal zone for writing journal data was allocated separately and excluded from garbage collection of the ZNS SSD. Experimental results show that the proposed method eliminates unnecessary copying of journal data during garbage collection and improves I/O performance by reducing garbage collection overhead.

The rest of this paper is organized as follows. In Section 2, we briefly describe the background of the ZNS SSD and dm-zoned. Then, Section 3 introduce related works which reduces garbage collection overhead of SSDs by using host-side information or host-side FTL. In Section 4, we present the proposed scheme to separate journal data from file data for the ZNS SSD. The experiment results are presented in Section 5. Finally, we conclude the paper in Section 6.

II. BACKGROUNDS AND MOTIVATION

2.1. ZNS SSD and Dm-Zoned

ZNS SSD is a new emerging type of SSD using zone-based interface instead of conventional block-based interface. In the ZNS SSD, the flash memory area is divided into multiple zones that must be written sequentially and exposed to the host. Moreover, unlike conventional block-interface SSDs, the host directly can manage the data placement and garbage collection. Therefore, the garbage collection overhead of ZNS SSDs can be minimized by optimized data placement using host-side information such as file type and correlation between files.

However, the legacy file systems such as Ext4 file system cannot use ZNS SSD directly because they are designed for block-interface storages. For that reason, the latest Linux kernel provides dm-zoned, a device mapper that allows ZNS SSDs to be used like conventional block storage devices, as you can see in Fig. 1.

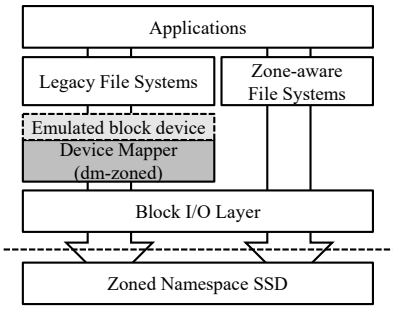


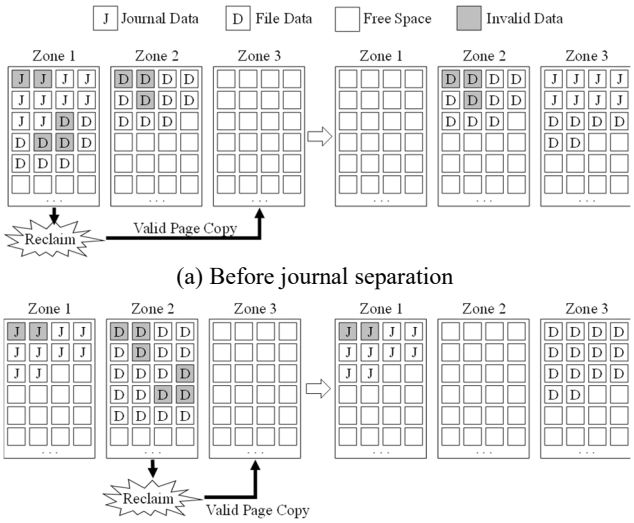
Fig. 1. Overview of dm-zoned in Linux kernel.

The dm-zoned combine conventional block-based SSD and ZNS SSD into one virtualized block-based storage device. And virtual block storage device is divided into the same zone size as ZNS SSDs. The zones in conventional SSD are called ‘Conventional Zones’ that can be written randomly, the other zones in ZNS SSD are called ‘Sequential Zones’ that can only be written sequentially. The conventional zone of block-based SSD caches data from the ZNS SSD and stores internal metadata related to the virtual storage device.

The dm-zoned manages the logical address space of the device based on a chunk, which is a logical unit of the same size with a zone. When a write request with a logical address occurs, the chunk number is calculated from the logical address, and the request is sent to the zone mapped to the corresponding chunk. If there is no zone mapped to the chunk, new conventional zone is allocated and mapped to the chunk. And after the mapping, all write requests about the chunk are sent to the zone mapped to the chunk.

The dm-zoned performs reclaim operations to prevent the lack of allocatable conventional zones. Since most write requests are redirected to the conventional zone, it is important to maintain free conventional zone through reclaim operation. If a reclaim request occurs for a conventional zone, collect all valid data in the zone, select an empty sequential zone, and delete the data of the zone to be reclaimed.

And after copying data to the sequential zone, the data cannot be updated directly due to sequential write constraints of ZNS SSD. Therefore, if a update request is issued, another conventional zone is allocated as a buffer of the sequential zone, and the data is updated the corresponding conventional zone. That is, two zones are mapped to single chunk. After the updated data is written in the associated buffer zone, the outdated data in the sequential zone is invalidated. And when a reclaim request is occurs for freeing conventional zone used as a buffer, valid data in sequential zone and valid data in corresponding buffer are merged and



(a) Before journal separation

(b) After journal separation

Fig. 2. Overview of the reclaim operation.

copied into the new sequential zone. As the amount of data copied in this process increases, the write amplification of the ZNS SSD becomes more severe and consumes P/E (Program/Erase) cycle of SSD unnecessarily.

2.2. Journaling in Ext4 File System

The journal overwrites fixed space repeatedly, and after changes of file system which is journal contains is written to the storage device permanently, corresponding journal is rarely used. However, the traditional method of using Ext4 file system on ZNS SSD via dm-zoned results in unnecessary copies of journal data. Fig. 2(a) shows the processing of reclaim operation before present dm-zoned architecture. Because journal data and general file data are stored without distinction, the reclaim operation copies the journal data to another zone.

Fig. 3 shows the zone where the journal is actually stored is changed continuously, even though the journal continues to use the same logical address space. Thus, by modifying the way dm-zoned behaves, it is possible to prevent copying of journal data and extend the life of the SSD by separating journal data into separate zones and setting it not to be retrieved, as shown in Fig. 2(b).

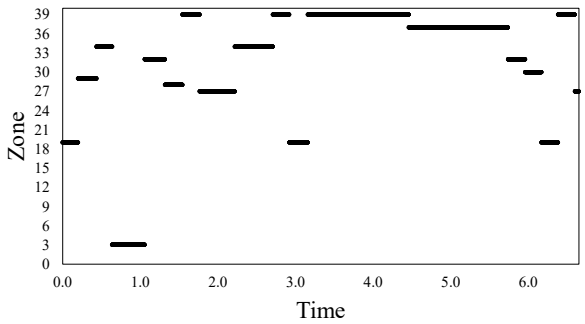


Fig. 3. Zone number used for journal write requests.

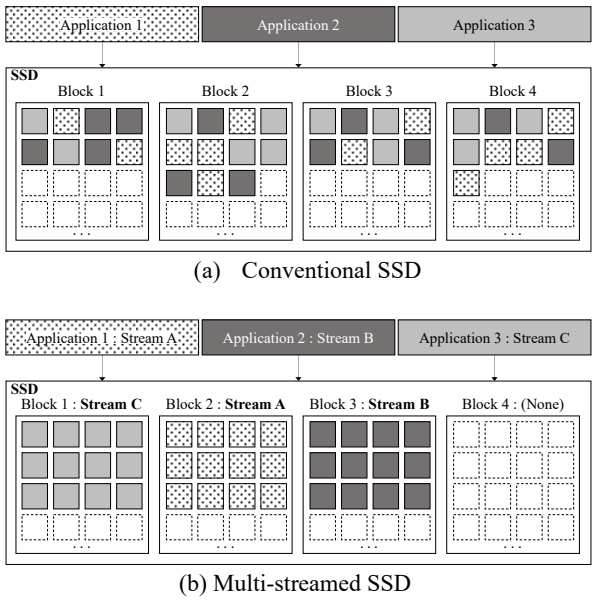
III. RELATED WORKS

In this section, we examine the previous studies and techniques for improving the performance of SSDs using host information.

3.1. Multi-Streamed SSDs

The Multi-Streamed SSD [7] is an kind of SSD that partially discloses the internal behaviors of the SSD to the host. Multi-Streamed SSD allows the host to specify stream number when write requests are issued by using a stream-based interface. So, write requests associated to different stream are written to different blocks in NAND flash memory. Fig. 4(a) shows that data from different applications can be mixed in a single block, increasing garbage collection overhead in the conventional SSD. However, the multi-streamed SSD can appropriately identify the update pattern of the data by using stream number specified by the host as described in Fig 4(b). As a result, the lifetime and performance of the multi-streamed SSD can be improved by preventing data having different update pattern is mixed in a single block.

AutoStream [8] proposes a method of automatically assigning streams instead of directly managing them. As the amount of data to be managed increases, the update pattern of data becomes more and more complicated, and it becomes difficult to allocate streams. Therefore, in that paper, to reduce the stream management overhead, they used a method of automatically allocating streams by grasping the



(a) Conventional SSD

(b) Multi-streamed SSD

Fig. 4. Comparison of conventional SSD with multi-streamed SSD.

characteristics of the actual input/output operation in real time when it is being performed.

FStream [9] proposes a method of allocating and managing streams at the file system layer. The file system logically stores general file data and file metadata separately but does not store them physically on the actual flash memory. So, FStream proposes a method of separating general file data and file metadata into separate streams.

PCStream [10] proposes a method of tracking 'write()' system call to group data with similar update patterns. Although general stream allocation techniques distinguish data by characteristics such as access frequency of data, but, PCStream used a method of allocating streams after predicting the lifetime of the data each time a write() system call is called.

vStream [11] proposes a method of allocating and managing streams regardless of the number of physical streams by generating virtual streams, noting that the number of allocatable streams in a typical SSD is limited.

DStream [12] proposes a method of adjusting the number of streams according to workload, noting that existing stream-related studies only considered the case of efficient use of fixed streams. By using the system resources saved through this as the mapping table cache of SSDs, a method of improving the performance of SSDs was used.

3.2. Open-Channel SSDs

Open-Channel SSD [13] is a storage platform that can make internal data structures completely public to the hosts. Because the physical address space is completely opened, there are advantages which is the host can leverage all resource of SSDs, but the disadvantage is that all resources has to be managed by hosts. For example, SSDs are composed of multiple flash memory chips and flash memory chips are connected to controller in parallel. Therefore, this parallelism must be ensured in order to fully utilize the performance of Open-Channel SSDs.

LightNVM [14] is a sub system of Linux kernels for Open-Channel SSD, which is minimizing resource management overhead on Open-Channel SSDs. LightNVM contains Host-level FTL, 'pblk', which is mapping logical addresses onto physical address on Open-Channel SSDs. The Open-Channel SSDs may be used like conventional block device through pblk.

SSW [15] proposed a method that prioritized sequential writes suitable for SSDs in an Open-Channel SSD environment. By checking the metadata of the file system, The SSW distinguish between sequential writes and random read/write tasks, and use a method of reducing garbage collection by appropriately assigning sequential write tasks to multiple channels.

IV. SEPARATING JOURNAL PLACEMENT

In this section, we describe design changes of dm-zoned to prevent unnecessary copying of journal data and to reduce garbage collection overhead by separating journal data into journal-only zone on the virtual block devices emulated through dm-zoned.

4.1. Dm-Zoned I/O Routine

The dm-zoned combines Conventional SSD and ZNS SSD to create a virtual block device. Since the Ext4 file system is mounted on a virtual device generated by dm-zoned, I/O request of the Ext4 file system are submitted to the virtual device. And at that time, I/O request is submitted through Block Layer I/O function *submit_bio()*.

I/O requests submitted to the virtual device are redirected to the physical device to which the physical data is stored. In this case, the function determining the physical device and zone where I/O requests is processed is *dmz_ map()*.

Fig. 5 illustrates how *dmz_map()* works during original

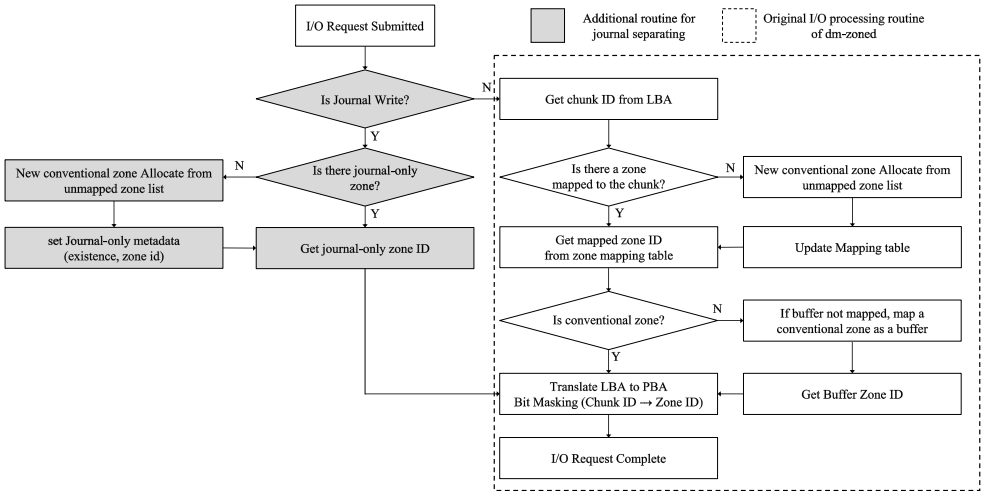


Fig. 5. Dm-zoneI/O routine for separating journal placement.

I/O routine in the dm-zoned. The first zone allocated is the conventional zone, and the write request to the sequential zone is redirected to the conventional zone mapped to the sequential zone as a buffer. That is, most of the write requests are transmitted to the conventional zone. Also, about I/O routine, noting that when the I/O request is submitted, there is a process of checking whether there is a mapped zone and then obtaining mapping information.

4.2. Separating Journal from File Data

In the Ext4 file system, when I/O request is occurred at user space, it is submitted to block layer first. At this step, block layer transmits the information of I/O request through struct bio. This structure is created and used from a function that processes I/O requests in the File system at first and is maintained until the I/O request is processed at the end.

Therefore, declare a member variable into struct bio to store metadata indicating the request file write request of ext4 file system, and metadata indicating the request journal write request. These metadata are used in the process of mapping Zones during the I/O Routine discussed in Section 4.1.

As you can see in Fig. 5, if the flag indicating a journal is enabled in the *struct bio* of certain write request, a separated process for the journal write request is executed. For the first journal write request, a new conventional zone is allocated, then the request is processed. And after then, all journal write requests are then redirected to that zone.

4.3. Allocation of Journal-Only Zone

In dm-zoned, all random write request are redirected to conventional zones. Moreover, an empty conventional zone is allocated upon first write request for a chunk to which the zone is not mapped. This is same for the journal data. Thus, when a conventional zone is allocated for the first journal write, an information indicating that it is a journal-only zone is stored in the metadata of the zone additionally. Actually, At time 1 in the Fig. 6-❶, the first journal write request is submitted. For handle this, Zone 2 is allocated as journal-only zone.

4.4. Garbage Collection Policy for Journal-Only Zone

In dm-zoned, the zones are reclaimed with Least Recently Used (LRU) Algorithm. That is, the mapped conventional zone list is iterated, and the most previously used zone is determined as a reclaimed target. However, in the proposed design, if the metadata of zone indicating a journal-only zone, the corresponding zone is never selected as reclaimed target. That is, if the reclaim operation occurs, reclaim target will be selected from among the mapped conventional zone list exclude journal-only zone.

Fig. 6-❷ illustrates this. At Time 7, there is two mapped conventional zone in the list. But one of them is the journal-only zone, therefore the other zone is selected to the victim zone to reclaim. Thus, Zone 4 is reclaimed, and then the valid data remained in Zone 4 is copied to sequential zone 6.

That is, journal-only zone is never reclaimed. Furthermore, because journal-only zone exists in the mapped conventional zone list, journal-only zone is never allocated for other write requests. As a result, there is no write requests can access the journal-only zone except for journal write request.

Additionally, the data copied to the sequential zone due to reclaim cannot be directly modified by sequential write

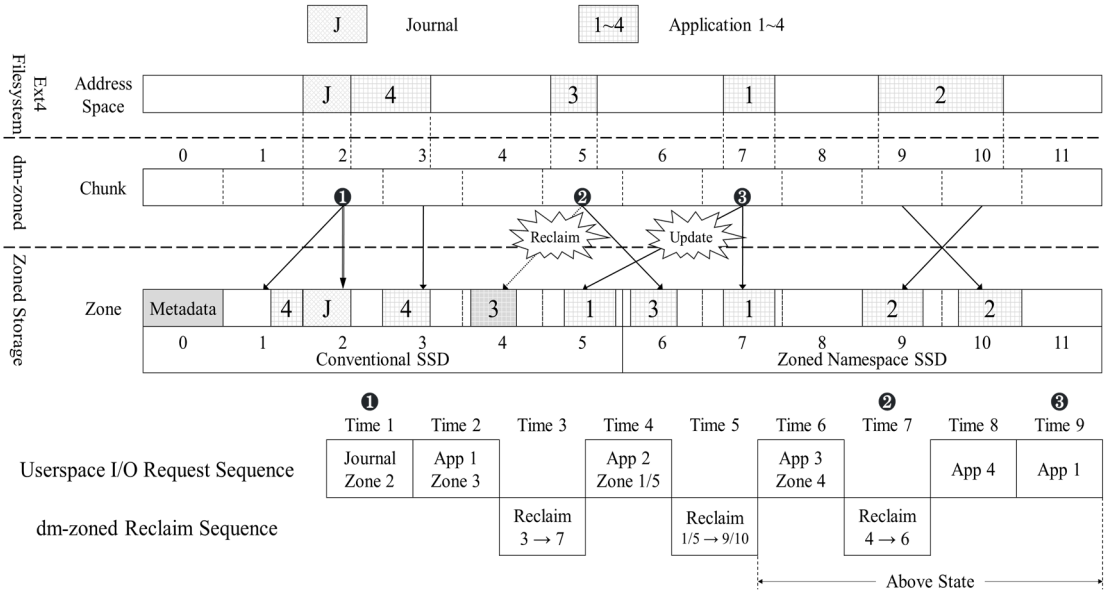


Fig. 6. Overview of allocation and garbage collection policy for journal-only zone.

constraints. Therefore, as we explained above, conventional zone is mapped to sequential zone as a buffer for handling random write request.

At time 9 in Fig. 6-❸, Write request of App 1 is submitted. Especially, App 1 is mapped to sequential zone 7, It cannot update directly. To handle this, conventional zone 5 is mapped to sequential zone 7 as a buffer, and then updated data is redirected to buffer zone 5.

V. EXPERIMENTAL RESULTS

The experimental environment is described in Table 1. As you can see in Table 1, the proposed methods are implemented in Linux kernel 5.10.136 and evaluated by using 4GB ZNS SSD emulated by using qemu-based SSD emulator, FEMU 7.0 (Flash EMUlator) [16]. Note that FEMU emulates SSD device in the virtual Linux machine. Also, you can see both hardware specifications of the host system and emulated virtual Linux system in Table 1. For evaluation, the dm-zoned device mapper emulated block device consisting of 1GB block storage device and 4GB ZNS SSD.

The FIO [17] benchmark (version 3.33) is used to generates various synthetic I/O workloads such as sequential read/write, random read/write to evaluate the I/O performance of storage devices. In the Ext4 filesystem, journal write requests are incurred by file system write requests, so we executed only sequential and random write workloads to evaluate the proposed journal separating scheme.

Table 1. Experimental setup.

Host system	
CPU	Intel (R) Xeon (R) Gold 5218 CPU 2.30 GHz× 2 (64 Threads)
Memory	128 GB
Virtual Linux system emulated by FEMU	
CPU	48 Cores
Memory	100 GB
OS	Linux Kernel 5.10.136
Storage	1 GB conventional SSD 4 GB ZNS SSD
FIO configuration	
I/O engine	Libaio, psync
Direct I/O	Enable
I/O Depth	32
Block size	4 K
Workloads	Random write (Uniform, Zipf, Pareto, Gaussian dist.) Sequential write

5.1. Write Amplification Due to Garbage Collection

Fig. 7 shows the amount of journal data copied in current dm-zoned during garbage collection. According to the figure, journal data copying accounts for up to 48.5% of additional write incurred by garbage collection. It means that unnecessary journal data copying is exacerbating the garbage collection overhead.

Therefore, we can see that the proposed journal separation scheme can reduce the amount of data copied during garbage collection by at least 14% and at most 26.8% in Fig. 8. According to the experimental results, the proposed scheme has an effect of reducing write amplification incurred by garbage collection regardless of journal mode and workload.

The result represented in Fig. 8 may not seem to have a significant effect compared to the previous experiment for the result represented to Fig. 7. It is because copied amount of journal data in Fig. 7 is around 50% at most. This result is due to the hided internal behavior of the dm-zoned. In the dm-zoned, the reclaim operation is triggered with special conditions which is few remaining conventional zones. But this behavior is invisible to the Ext4 file system. In sight from Ext4 file system, there is only logical chunks in the emulated block storage device, which is Ext4 file system

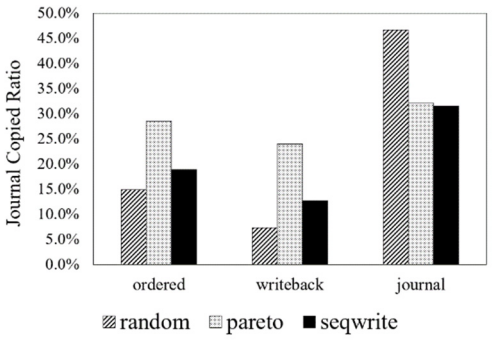


Fig. 7. The amount of journal data copied during garbage collection in the current dm-zoned.

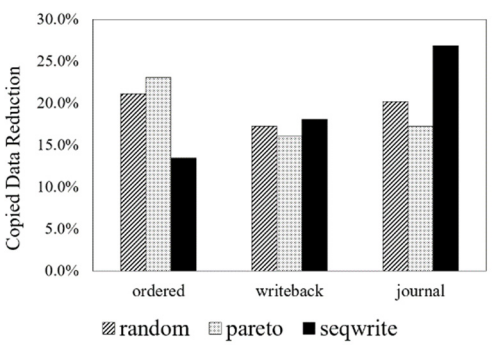


Fig. 8. The reduction ratio of data copied during garbage collection in the proposed methods.

mounted, and the chunks remained fixed however the physical zones mapped to the corresponding chunk are remapped continuously. Thus, there was a deviation in measuring the amount of data copying each time, so the average of the results measured five times is shown in the Fig. 8.

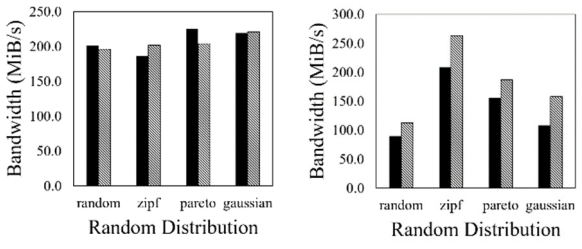
Summarizing the results, as shown in Fig. 7, before the introduction of journal-only zone, journal data copied accounted for about 10 to 30% of the total data copy amount, except for the highest ratio value. Also, looking at Fig. 8, the after introducing the journal-only zone, the total data copy amount decreased by 13%−27%, similar to the ratio of journal data copied in Fig. 7. This means that journal data copied can be effectively removed with our proposed method.

5.2. I/O Performance

Moreover, Fig. 9 compares the I/O performance of current dm-zoned and the proposed scheme. Note that Fig. 9(a) and Fig. 9(b) shows I/O performance of Ext4 file system with different journaling modes, respectively. In the case of the ordered mode (in Fig. 9(a)), where journal data is generated only for metadata update, there is no significant I/O performance improvement due to the small amount of journal data. In the contrast, in the case of the journal mode (in Fig. 9(b)), where journal data is generated for every write request, we can see that I/O performance has improved. These improvement of I/O performance is because the proposed journal separation scheme reduces garbage collection overhead. As a result, our proposed scheme not only reduces write amplification but also improves I/O performance of ZNS SSDs.

VI. CONCLUSION

The NAND flash memory based SSD is most popular storage device because of its outstanding performance. However, SSDs suffer from garbage collection overhead due to the out-place update scheme that addresses the erase-before-write constraint of NAND flash memory. To reduce write amplification incurred by garbage collection, data having similar update pattern should be placed in the same NAND block. The ZNS SSD, a new type of SSD, can minimize the garbage collection overhead by allowing the host to directly manages data placement. However, legacy journaling file systems still cannot use the advantages of ZNS SSDs. Therefore, in this paper, we proposed the journal separating scheme which allocates journal-only zone for journal data to separate journal data from file data. The proposed scheme is implemented in Linux kernel and carefully evaluated. The experimental results show that, in various journaling modes, the proposed journal separating scheme can reduce the write amplification of the ZNS SSD by up to 26.8% while improving I/O performance.



(a) Ordered (b) Journal

Fig. 9. I/O performance evaluation results.

ACKNOWLEDGMENT

This work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-01201, Convergence security core talent training business (Pusan National University)) and ITRC (Information Technology Research Center) support program (IITP-2022-2020-0-01797) supervised by the IITP.

REFERENCES

[1] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Proceedings of the Linux Symposium*, Ottawa, Canada, Jun. 2007.

[2] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of the 2005 USENIX Annual Technical Conference*, Anaheim, CA, Apr. 2005, pp. 105-120.

[3] S. Kim and E. Lee, "Analysis and improvement of I/O performance degradation by journaling in a virtualized environment," *The Journal of the Institute of Internet, Broadcasting and Communication*, vol. 16, no. 6, pp. 177-181, Dec. 2016.

[4] S. Son and S. Ahn, "Optimizing garbage collection overhead of host-level flash translation layer for journaling file systems," *International Journal of Internet, Broadcasting and Communication*, vol. 13, no. 2, pp. 27-35, May 2021.

[5] M. Bjørling, A. Aghayev, H. Holmberg, A. Ramesh, D. L. Moal, G. R. Ganger, et al., "ZNS: Avoiding the block interface tax for flash-based SSDs," in *Proceedings of the 2021 USENIX Annual Technical Conference*, Jul. 2021, pp. 689-703.

[6] dm-zoned-tools, https://docs.kernel.org/admin-guide/device-mapper/dm-zoned.html.

[7] J. Bhimani, J. Yang, Z. Yang, N. Mi, N. H. V. Krishna Giri, and R. Pandurangan, et al., "Enhancing SSDs with multi-stream: What? why? how?," in *Proceedings of the 36th International Performance Computing and Communications Conference*, San Diego, CA, Dec. 2017.

[8] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "AutoStream: Automatic stream management for multi-streamed SSDs," in *Proceedings of the 10th ACM International Systems and Storage Conference*, Haifa Israel, May 2017, pp. 1-11.

[9] E. Rho, K. Joshi, S. Shin, N. J. Shetty, J. Hwang, and S. Cho, D. et al., "FStream: Managing flash streams in the file system," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, Oakland, CA, Feb. 2018, pp. 257-263.

[10] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, and J. Hwang, et al., "PCStream: Automatic stream allocation using program contexts," in *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems*, Boston, MA, Jul. 2018.

[11] H. Yong, K. Jeong, J. Lee, and J. Kim, "vStream: Virtual stream management for multi-streamed SSDs," in *Proceedings of 10th USENIX Workshop on Hot Topics in Storage and File Systems*, Boston, MA, Jul. 2018.

[12] S. Lim and D. Shin, "DStream: Dynamic memory resizing for multi-streamed SSDs," in *Proceedings of the 2019 34th International Technical Conference on Circuits/Systems, Computers and Communications*, JeJu, Korea, Aug. 2019, pp. 1-4.

[13] I. L. Picoli, N. Hedam, P. Bonnet, and P. Tözün, "Open-channel SSD (What is it Good For)," in *Proceedings of 10th Annual Conference on Innovative Data Systems Research*, Amsterdam, Netherlands, Jan. 2020.

[14] M. Bjørling, J. Gonzalez, and P. Bonnet, "LightNVM: The Linux open-channel SSD Subsystem," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, Santa Clara, CA, Feb. 2017, pp. 359-373.

[15] Y. Du, J. Gu, Z. Xiao, and M. Huang, "SSW: A strictly sequential writing method for open-channel SSD," *Journal of Systems Architecture*, vol. 109, p. 101828, Oct. 2020.

[16] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator," in *Proceedings of the 16th USENIX Conferenceon File and Storage Technologies*, Oakland, CA, Feb. 2018, pp. 83-90.

[17] J. Axboe, 2021. Flexible I/O Tester, https://github.com/axboe/fio.

[18] S. Lee, D. Shin, Y. Kim, and J. Kim, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 36-42, Oct. 2008.

[19] G. Wu and X. He, "Delta-FTL: Improving SSD lifetime via exploiting content locality," in *Proceedings of the 7th ACM European Conference on Computer Systems*, Bern, Switzerland, Apr. 2012, pp. 253-266.

[20] NVM Express Base Specification 2.0c, https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf.

AUTHORS

Young-in Choi received the B.S. degree in the Department of Physics from Pusan National University, Busan, South Korea, in 2020. He is studying for a M.S. degree in the School of Computer Science and Engineering, Pusan National University, Busan, South Korea. His interests include storage technologies, I/O stack of operating system and cloud computing.

Sungyong Ahn received the B.S. and Ph.D. degree in the Department of Computer Science and Engineering from Seoul National University, Korea, in 2003 and 2012, respectively. He worked at Samsung Electronics as a senior engineer from 2012 to 2017. In 2017, he joined the School of Computer Science and Engineering, Pusan National University, Busan, South Korea, where he is currently a professor. His interests include operating systems, solid-state disk, and emerging memory technologies.