International Journal of Internet, Broadcasting and Communication Vol.13 No.2 27-35 (2021)
http://dx.doi.org/10.7236/IJIBC.2021.13.2.27

IJIBC 21-2-5

Optimizing Garbage Collection Overhead of Host-level Flash Translation Layer for Journaling Filesystems

Sehee Son[1] and Sungyong Ahn[2]

[1]Master Student, School of Computer Science and Engineering, Pusan National University, Korea
[2]Assistant Professor, School of Computer Science and Engineering, Pusan National University, Korea
[1]ight13751@pusan.ac.kr, [2]sungyong.ahn@pusan.ac.kr

Abstract

NAND flash memory-based SSD needs an internal software, Flash Translation Layer(FTL) to provide traditional block device interface to the host because of its physical constraints, such as erase-before-write and large erase block. However, because useful host-side information cannot be delivered to FTL through the narrow block device interface, SSDs suffer from a variety of problems such as increasing garbage collection overhead, large tail-latency, and unpredictable I/O latency. Otherwise, the new type of SSD, open-channel SSD exposes the internal structure of SSD to the host so that underlying NAND flash memory can be managed directly by the host-level FTL. Especially, I/O data classification by using host-side information can achieve the reduction of garbage collection overhead. In this paper, we propose a new scheme to reduce garbage collection overhead of open-channel SSD by separating the journal from other file data for the journaling filesystem. Because journal has different lifespan with other file data, the Write Amplification Factor (WAF) caused by garbage collection can be reduced. The proposed scheme is implemented by modifying the host-level FTL of Linux and evaluated with both Fio and Filebench. According to the experiment results, the proposed scheme improves I/O performance by 46%~50% while reducing the WAF of open-channel SSDs by more than 33% compared to the previous one.

Keywords: *Solid-State Disk, Flash Translation Layer, Open-channel SSD, Journaling Filesystem, EXT4 Filesystem*

1. Introduction

Today, in the storage device market, the NAND flash memory-based Solid-State Disk (SSD) is rapidly displacing the traditional storage, Hard Disk Drive (HDD) because of its various advantages such as low latency, high density, low power consumption and shock resistance. On the other hand, because NAND flash memory cannot overwrite data in-place, SSD needs internal software called Flash Translation Layer (FTL) to provide block device interface to host. The FTL updates the data with an out-place update fashion where obsolete data is invalidated and new data is written to other location. The invalidated data is periodically

Manuscript Received: January. 27, 2021 / Revised: February. 3, 2021 / Accepted: February. 7, 2021
Corresponding Author: sungyong.ahn@pusan.ac.kr
Tel: +82-51-510-2422, Fax: +82-51-517-2431
Assistant Professor, School of Computer Science and Engineering, Pusan National University, Korea

deleted by garbage collection, another major feature of FTL, to reserves free space. However, because the erase unit of NAND flash memory (NAND block) is much larger than write unit, the valid pages in a victim block must be copied to another location before deleting the NAND block. In other words, garbage collection of FTL causes additional writes, which causes performance degradation and shortened lifespan of the SSD.

Moreover, garbage collection overhead increases when short-lived and long-lived data are mixed in same NAND block. However, because host-side information which is useful for data classification is lost due to the narrow block device interface, SSDs suffer from a variety of problems such as garbage collection overhead, large tail-latency, and unpredictable I/O latency[1, 2].

Therefore, many studies have been conducted to overcome the limitations of the legacy block interface and deliver useful host-side information to the FTL inside the SSD. The TRIM[3] command which is one of the earliest studies to overcome the limitations of the legacy block interface has already been standardized and is supported successfully in most operating systems and SSDs. It is a newly proposed command which helps to perform garbage collection more effectively by notifying the SSD that certain data has been deleted by the host. Another notable study is the multi-stream SSD[4] which reduces garbage collection overhead by allocating I/O streams with different write patterns to a separate NAND flash area by transferring I/O stream information to the SSD. The multi-stream technique is also included in the latest NVM Express (NVMe) [5] standard. However, the existing studies have a limitation in that they can still provide very limited information. Therefore, the host-level FTL which directly manages the NAND flash memory of a SSD attracts attention because it can utilize host side information.

Open-channel SSD[6] is a new type of SSD that can fundamentally overcome the limitations of SSDs employing an internal FTL by revealing the internal structure of the SSD to the host. In open-channel SSD, the host directly performs FTL features such as data placement and garbage collection, so that the performance of SSDs can be optimized through utilizing host-side information such as I/O data classification using file system information and performance isolation through workload classification. However, existing the I/O stack of the operating system has not yet utilized the host-side information. Therefore, in this paper, we propose a new scheme to reduce garbage collection overhead of open-channel SSD by separating the journal from other file data for the journaling file system. The proposed scheme is implemented by modifying the host-level FTL of Linux and evaluated with Fio and Filebench. According to the experiment results, the proposed scheme improves I/O performance by 46%~50% while decreasing garbage collection overhead of SSD,WAF by more than 33% in comparison to the previous one.

The remainder of this paper is organized as follows. Section 2 introduces the open-channel SSD and journaling system of Ext4 filesystem. Then, Section 3 briefly shows the previous studies related to open-channel SSDs. Section 4 describes the design and implementation of proposed scheme. Experimental results are presented in Section 5. The conclusion is given in Section 6.

2. Background

2.1 Open-channel SSD

Open-channel SSD is proposed as a new class of SSDs to overcome shortcomings of conventional block device interface SSDs [6]. As shown in Figure 1(a), in the case of a conventional SSDs, FTL is implemented in the form of firmware inside the SSD, so that it can support a block device interface to the host. As a result, the host can deliver I/O requests with logical address as if SSDs can update data in-place. On the other hand, as shown in Figure 1(b), the open-channel SSD exposes the internal structure of the SSD to the host, so that the host can directly manage underlying NAND flash memory for open-channel SSDs.

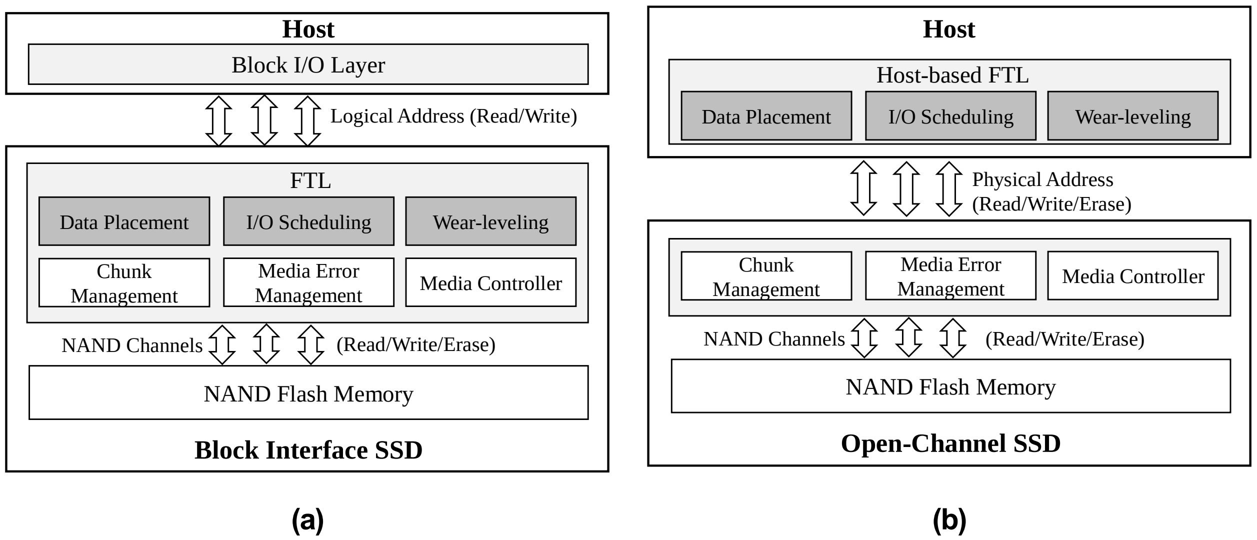


Figure 1. (a) Conventional block interface SSD vs. (b) Open-channel SSD[7]

The LightNVM[7], Linux subsystem, has been provided to support open-channel SSD since kernel 4.4. The LightNVM has host-level FTL called Pblk (i.e., Physical Block Device) which provides a block device interface to user programs, while internally performing data placement and garbage collection for underlying NAND flash memory. As a result, open-channel SSDs can take the following advantages [8].

- **I/O Isolation:** Since the host knows the internal structure of the SSD, it is able to allocate physically separated NAND flash memory region for each application program. As a result, I/O performance interference between different applications can be minimized.
- **Predictable latency**: In the conventional SSDs, long-tail latency is mainly caused by SSD internal operations such as garbage collection and wear-leveling. However, in the open-channel SSD, because the host can check the status of the internal operations, the completion time of I/O requests can be predicted. Moreover, if necessary, it is possible to delay garbage collection to achieve the desired I/O completion time.
- **Software-Defined Non-Volatile Memory**: Host-level FTL can be optimized for specified application by integrating FTL functions into the application.

2.2 Ext4 Journaling Filesystem

Ext4 filesystem [9, 10], the default filesystem of Linux, employs a journaling system to maintain the consistency and stability of the filesystem. In journaling systems, when a write operation is requested, the information about the write operation (called as journal) should be recorded in a separated journal region of storage before writing the actual file data into the storage device. Therefore, even if a system failure occurs during processing the write operation and causes the inconsistency of the filesystem, the filesystem can be quickly restored by using the journal written in the journal region.

In the Linux, the Journaling Block Device2 (JBD2) daemon performs journaling for Ext4 filesystem periodically or by the *fsync()* system call. JBD2 performs journaling in three steps: At first, the journal for file write operation is recorded in the journal region separated from the file data region of the filesystem. Second step is *Commit* indicating that the journal is successfully recorded. After *Commit*, the file data is finally written in the file data area of the storage device in *Checkpoint* step. Note that the journal record can be deleted after file data writing is completed.

As mentioned above, a journal of Ext4 filesystem is deleted in most cases after the corresponding file data is completely written to file data region of the storage. However, since the SSD performs an out-place update, the deleted journal is not actually deleted from the NAND flash memory but remains as an invalid page that will be deleted during the next garbage collection. As shown in Figure 2, if the NAND pages including general file data (*Data page*) and the NAND pages including journal (*Journal page*) are recorded together in same NAND block, the number of valid pages copied during garbage collection may increase.

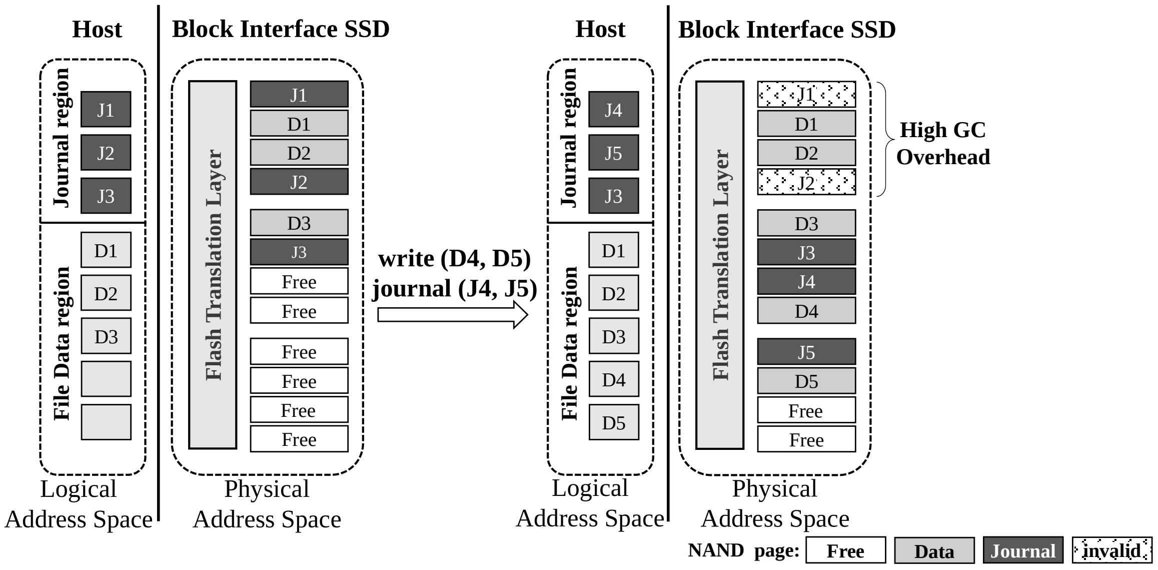


Figure 2. Garbage collection overhead without journal separation

The reason is that file data and journal have different lifespan. However, in the existing block interface SSDs, information other than the logical address cannot be transferred from the host to the SSDs, so FTL firmware cannot distinguish file data and journal. Therefore, in this paper, we propose a scheme which separates journal from file data in journaling filesystem to reduce garbage collection overhead of open-channel SSDs.

3. Related Work

There have been several studies on optimizing I/O stack by using open-channel SSDs. The one of the earliest studies is optimizing Key-Value(KV) store to the open-channel SSDs. The KV stores based on Log-Structured Merge Trees (LSM-tree)[11] such as LevelDB[12] and RocksDB[13] is attracting attention because of its good read speed and sequential write property that is suitable for SSD. However, LSM-tree should periodically merge key-value pairs which have same keys. This operation, called *compaction*, is the main reason of deterioration of the LSM-tree performance as it generates many read/write operations.

In the previous study, it is proposed to classify the files constituting the LSM-tree according to their characteristics and allocate them to a separate NAND flash memory region. The proposed method can increase the throughput of KV store by more than four times while reducing garbage collection overhead[14]. Also, RocksDB have been optimized for open-channel SSDs by integrating garbage collection of FTL to the compaction process of LSM-tree[15]. In the past, the compaction process and garbage collection were performed separately because the KV store could not know the internal structure of the SSD. However, previous studies show that the garbage collection overhead of open-channel SSD can be reduced by integrating both operations.

Another studies are about the performance isolation technique by using open-channel SSDs in the multi-tenant environment. In a cloud environment, several tenants sharing SSDs are suffered from severe I/O performance interference and large tail-latency. Therefore, it is proposed to allocate an independent channel or die to each tenant using an open-channel SSD to make the most of the internal parallelism of the SSD while minimizing the performance interference between tenants[16]. According to the evaluation results, proposed storage system increase throughput by up to 1.6 times and provides 3.1 times lower tail latency compared to legacy SSDs.

4. Design and Implementation

4.1 Design Overview

As mentioned in Section 2, journal has different lifespan from file data in the journaling filesystem such as Ext4. Therefore, it is better to write he *journal pages* and *data pages* in separate NAND blocks to reduce garbage collection overhead. However, the FTL firmware of conventional block interface SSD cannot distinguish journal from file data because it has no information about the filesystem. However, in the case of open-channel SSDs, the host-level FTL can achieve the useful information from filesystems. So, in this section, we propose host-level FTL that separates journal from file data to reduce garbage collection overhead by utilizing the information delivered from filesystem.

Figure 3 shows the basic idea and effectiveness of proposed scheme. As you can see in the figure, the proposed scheme writes *journal pages* to a separate NAND block(*Journal Block*).

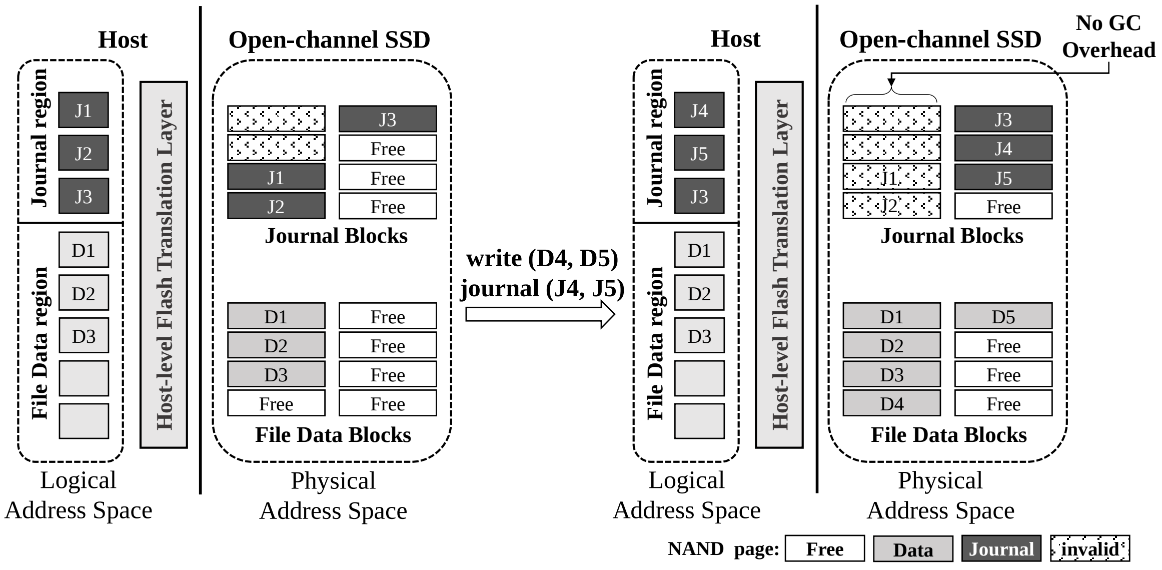


Figure 3. Garbage collection overhead with separating journal on open-channel SSDs

Therefore, *Journal block* are highly likely to be erased without valid page copy during garbage collection because journal is sequentially deleted after *Checkpoint* step of journaling. As a result, the Write Amplification Factor (WAF) which directly affects the lifespan of the SSDs can be reduced by using proposed scheme. Note that WAF represents the ratio of the amount of additional writes generated by garbage collection, and the larger this value, the higher the overhead of garbage collection and the shorter the lifespan of the SSD.

4.2 Implementation

In this section, we describe the operation of host-level FTL supporting Ext4 journal separation for open-channel SSDs as can be seen in Figure 4. The proposed scheme is implemented by using following two features. First, a method is needed to deliver the information that can distinguish journal from file data to host-level

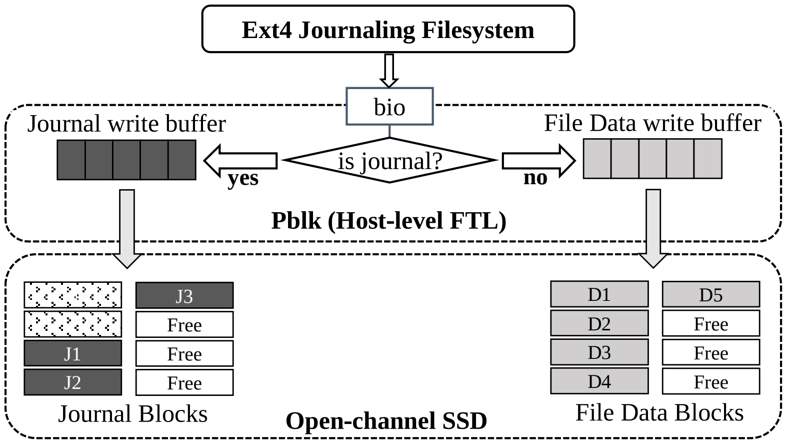


Figure 4. Design for journal separating host-level FTL

FTL from filesystem. Second, we need a way to manage journals separate from file data in host level FTL.

At first, to deliver journal identification information to the host-level FTL, we employ a new flag indicating journal in the *struct bio*. Here, *struct bio* is Linux kernel structure describing block I/O request. The journal flag is set to inform that the write request is for journal when the journal is committed by JBD2. As a result, the host-level FTL can allocates a *struct bio* of which journal flag is set to the *journal block*.

Second, the proposed scheme uses two circular buffers to handle journal writes and file data writes separately. Write requests are first written to the write buffer inside host-level FTL. When the free space in the buffer is exhausted, they are requested to the SSD at once by host-level FTL. At this time, if journal and file data are mixed in same buffer, it is difficult to write them separately. Therefore, we employ two separate write buffers for journal and file data, respectively. Although additional memory space should be consumed, it is negligible compared to system memory size.

5. Experimental Results

5.1 Experimental Environments

The proposed scheme is implemented in Pblk, host-level FTL of Linux, and Ext4 filesystem of Linux kernel 4.16 and evaluated on Ubuntu 18.04 emulated on QEMU virtual machine. The detailed experimental environments are described in Table 1. As can be seen in the table, open-channel SSD is emulated on the

Table 1. Experimental environments

	Component	Specification
Host Hardware	CPU	Intel Xeon CPU E5-2620 v4 @ 2.10GHz x 2
	Memory	64GB DDR4
	SSD	Intel DC P4610 1.6TB NVMe SSD
QEMU virtual machine	CPU	8 cores
	Memory	4GB
	Operating system	Ubuntu 18.04
Virtual OCSSD	# of Group	2
	# of PU per Group	4
	# of Chunk per PU	60
	# of Sectors per Chunk	4096
	Sector Size	4KB
	Total Capacity	7.5GB

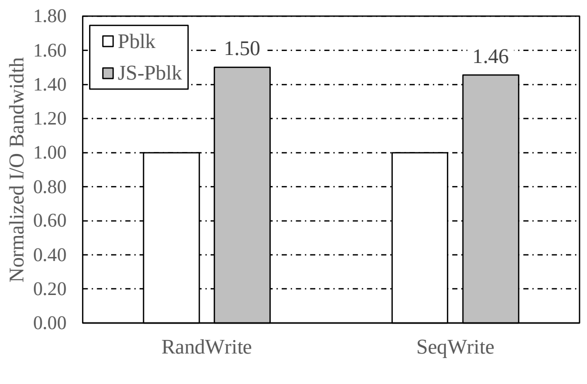
backend storage device, Intel DC P4610 NVMe SSD by using Virtual Open-channel SSD[17] because real open-channel SSD devices are not officially released yet. Moreover, Fio[18] and Filebench[19] are used for performance evaluation with the configuration described in Table 2.

Table 1. Workload characteristics

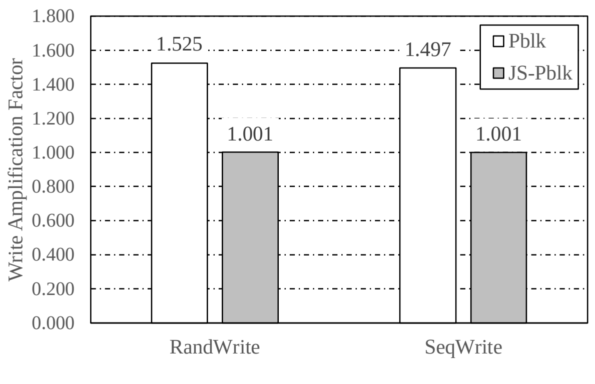
Benchmark		Parameters	Values
Fio		I/O size (rand/seq)	4KB / 1MB
		I/O depth	32
		Total I/O size	2GB
Filebench	Fileserver	Mean file size	128KB
		I/O size	1MB
		Mean append size	16KB
	Webserver	Mean file size	16KB
		I/O size	1MB
		Mean append size	16KB
	MongoDB	File size	16KB
		Mean I/O size	16KB
		Read I/O size	1MB

5.2 Evaluation Results

Figure 5(a) shows normalized I/O bandwidth of the proposed journal separation FTL (*JS-Pblk*) compared to the original host-level FTL (*Pblk*) for Fio workload. As you can see the figure, *JS-Pblk* increases the I/O bandwidth by 50% for both random and sequential writes. Moreover, Figure 5(b) shows that the performance improvement of *JS-Pblk* is due to reduced garbage collection overhead. According to the figure, WAF of *JS-Pblk* is only up to 67% compared to previous one. It indicates that the proposed journal separation policy dramatically decreases the garbage collection overhead of host-level FTL.



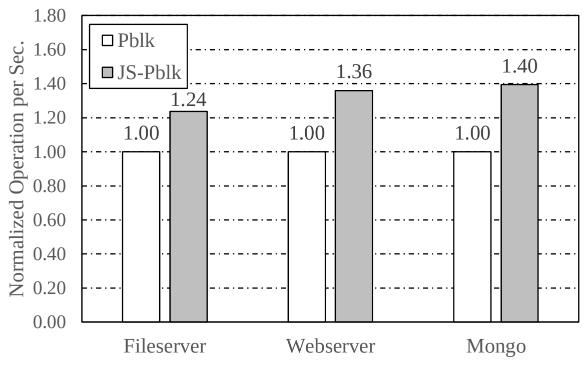
(a) Normalized I/O bandwidth

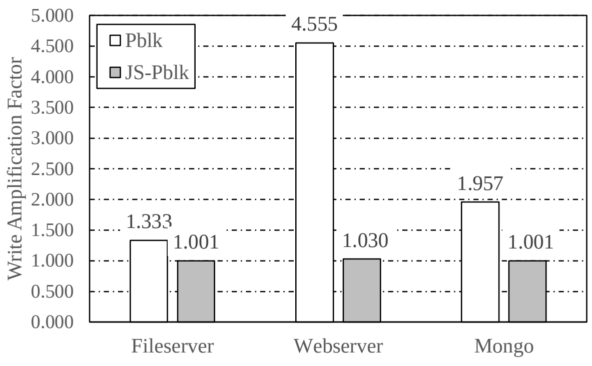


(b) Write amplification factor

Figure 5. Garbage collection overhead for Fio workloads

Similar results are found in the performance evaluation using Filebench emulating server workloads. Figure 6(a) shows normalized number of operations per second(OPS) for three different Filebench workloads. As you can see the figure, *JS-Pblk* delivered up to 40% improved performance. Moreover, in the Figure 6(b), JS-Pblk shows a WAF close to 1 while Pblk shows WAF above 4.5 at the maximum. increases WAF decreased by up to 41% and the number of VPC pages decreased from 57% to 100%. In the Filebench performance evaluation results, it is confirmed that WAF is dramatically reduced for the real-world workloads with *JS-Pblk*.





(a) Normalized OPS **(b) Write amplification factor**

Figure 6. Garbage collection overhead for Filebench workloads

6. Conclusion

Unlike conventional block interface SSDs, the open channel SSD is a new type of SSD that exposes the internal structure of the SSD to the host and allows the host directly to manage underlying the NAND flash memory. Therefore, host-level FTL of open-channel SSD can utilize host-side information to optimize the performance of SSDs. By classifying hot and cold data using host information, data placement in NAND flash memory can be optimized to reduce garbage collection overhead. In this paper, we have proposed a scheme to reduce the garbage collection overhead of open-channel SSD by separating the journal of the journaling file system from the file data. The proposed scheme is implemented by modifying the host-level FTL of Linux for open-channel SSD and evaluated using an emulated virtual open-channel SSD. According to the evaluation results, the proposed scheme improves I/O bandwidth by 46%~50% while reducing the WAF of open-channel SSDs by more than 33% compared to the previous one. The results means that by recording the journal of filesystem in a separate NAND block, the SSD can enjoy the extended lifespan with small garbage collection overhead.

Acknowledgement

This work was supported by a 2-Year Research Grant of Pusan National University.

References

[1] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi, “The tail at store: a revelation from millions of hours of disk and SSD deployments,” *in Proc. 14th USENIX Conference on File and Storage Technologies (FAST ‘16)*, pp. 263–276, Feb. 22-25, 2016.
DOI: https://dl.acm.org/doi/10.5555/2930583.2930603

[2] F. Chen, T. Luo, and X. Zhang, “CAFTL : A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives,” *in Proc. 9th USENIX Conference on File and Storage Technologies (FAST ‘11)*, Feb. 15-17, 2011.
DOI: https://dl.acm.org/doi/10.5555/1960475.1960481

[3] J. Kim, H. Kim, S. Lee, and Y. Won, “FTL design for TRIM command,” *in Proc. 5th International Workshop on Software Support for Portable Storage (IWSSPS 2010)*, pp. 7-12, Oct. 28, 2010.

[4] J. Kang, J. Hyun, H. Maeng, and S. Cho, “The Multi-streamed Solid-State Drive,” *in Proc. 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage ‘14)*, June 17-18, 2014.
DOI: https://dl.acm.org/doi/abs/10.5555/2696578.2696591

[5] NVMe overview. https://www.nvmexpress.org/wpcontent/uploads/NVMe_Overview.pdf.

[6] Open-channel Solid State Drives. https://openchannelssd.readthedocs.io/en/latest/.

[7] M. Bjørling, C. Labs, J. Gonzalez, F. March, and S. Clara, “LightNVM: The Linux Open-channel SSD Subsystem,” *in Proc. 15th USENIX Conference on File Storage Technologies (FAST ‘17)*, pp. 359–374, Feb. 27-March 2, 2017.
DOI: https://dl.acm.org/doi/abs/10.5555/3129633.3129666

[8] I. L. Picoli, N. Hedam, P. Bonnet, and P. Tözün, “Open-channel SSD (What is it Good For),” *in Proc. 10th Annual Conference on Innovative Data Systems Research (CIDR ‘20)*, Jan. 12-15, 2020.

[9] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, “The New Ext4 Filesystem: Current Status and Future Plans,” *in Proc. Linux Symposium*, Vol. 2, pp. 21-33, 2007.

[10] 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(JIIBC)*, Vol. 16, No. 6, pp. 177-181, Dec. 2016.

[11] P. O’Neil, E. Cheng, D. Gawlick, and E O'Neil, “The log-structured merge-tree (LSM-tree)”, *Acta Informatica*, Vol. 33, No. 4, pp. 351-385, June 1996.
DOI: https://doi.org/10.1007/s002360050048

[12] LevelDB. https://github.com/google/leveldb.

[13] RocksDB. https://github.com/facebook/rocksdb.

[14] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. “An efficient design and implementation of LSM-tree based key-value store on open-channel SSD,” *in Proc. of the 9th European Conference on Computer Systems (EuroSys ‘14)*, pp. 1-14, April 2014.
DOI: https://doi.org/10.1145/2592798.2592804

[15] RocksDB on Open-Channel SSDs. https://javigongon.files.wordpress.com/2011/12/rocksdbmeetup.pdf.

[16] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi “Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds,” *in Proc. 15th USENIX Conference on File and Storage Technologies (FAST ‘17)*, pp. 375-390, Feb. 27-March 2, 2017.
DOI: https://dl.acm.org/doi/10.5555/3129633.3129667

[17] QEMU Open-channel SSD 2.0. https://github.com/OpenChannelSSD/qemu-nvme.

[18] Fio - Flexible I/O tester rev. 3.23. https://fio.readthedocs.io/en/latest/fio_doc.html.

[19] V. Tarasov, E. Zadok, and S. Shepler, “Filebench: A Flexible Framework for File System Benchmarking,” *USENIX ;login*, Vol. 41, No. 1, pp. 6–12, April 2016.