International Journal of Advanced Smart Convergence Vol.10 No.1 142-150 (2021)
http://dx.doi.org/10.7236/IJASC.2021.10.1.142

IJASC 21-1-15

Implementation of Memory Efficient Flash Translation Layer for Open-channel SSDs

Gijun Oh[1], Sungyong Ahn[2]

[1]*Master, School of Computer Science and Engineering, Pusan National University, Korea*
[2]*Assistant Professor, School of Computer Science and Engineering, Pusan National University, Korea*

[1]*kijunking@pusan.ac.kr*, [2]*sungyong.ahn@pusan.ac.kr*

Abstract

Open-channel SSD is a new type of Solid-State Disk (SSD) that improves the garbage collection overhead and write amplification due to physical constraints of NAND flash memory by exposing the internal structure of the SSD to the host. However, the host-level Flash Translation Layer (FTL) provided for open-channel SSDs in the current Linux kernel consumes host memory excessively because it use page-level mapping table to translate logical address to physical address. Therefore, in this paper, we implemente a selective mapping table loading scheme that loads only a currently required part of the mapping table to the mapping table cache from SSD instead of entire mapping table. In addition, to increase the hit ratio of the mapping table cache, filesystem information and mapping table access history are utilized for cache replacement policy. The proposed scheme is implemented in the host-level FTL of the Linux kernel and evaluated using open-channel SSD emulator. According to the evaluation results, we can achieve 80% of I/O performance using the only 32% of memory usage compared to the previous host-level FTL.

Keywords: *Open-channel SSD, Memory Management, Caching, Journaling Filesystem, Flash Translation Layer, Host-level Flash Translation Layer*

1. Introduction

Solid-State Drives (SSDs), a storage device based on NAND flash memory, are rapidly replacing the existing Hard-Disk Drives (HDDs) due to its various advantages such as fast I/O performance, low power consumption, and shock resistance. However, because of physical constraints of NAND flash memory such as erase-before-write , limited erase count and large size of erase block, Flash Translation Layer (FTL) should be employed to provide conventional block device interface. The main role of FTL is to hide the characteristics of NAND flash memory by performing out-place-update, Logical-to-Physical (L2P) mapping, and garbage collection, so that the host can access the SSD through the existing block interface [1, 2]. However, because useful host information cannot transfer to the FTL in firmware form inside the SSDs, the existing FTL

Manuscript Received: February. 11, 2021 / Revised: February. 15, 2021 / Accepted: February. 18, 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

firmware cannot fundamentally solve the reduction in lifespan of SSDs due to the garbage collection and write amplification.

Open-channel SSD [3-5] is a new type of SSD that exposes the internal structure of the SSD to the host. Therefore, unlike conventional SSDs, FTL functions can be directly performed at the host rather than firmware level. Note that this kind of FTL is called host-level FTL compared to conventional FTL in the form of the firmware. Moreover, because the host-level FTL is performed at the host, it can utilize various useful information of the host to optimize FTL [7-9].

In Linux, open-channel SSD is supported by the host-level FTL called Pblk (i.e. Physical Block Device) [6] which manage L2P address mapping information with page-level mapping table rather than block-level mapping table. However, as shown in Figure 1, as the capacity of open-channel SSD increases, the host memory is excessively used for storing L2P mapping table. Therefore, memory efficient host-level FTL is required for employing extremely large capacity open-channel SSDs.

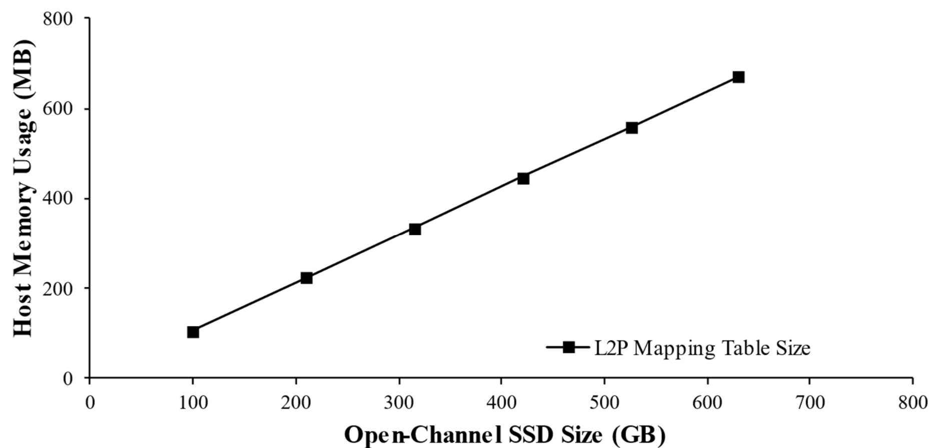


Figure 1. Host memory usage for mapping table of Open-channel SSD

In this paper, we propose a selectively mapping table loading scheme to reduce memory footprint of host-level FTL for open-channel SSDs. The proposed scheme loads only a currently required part of the mapping table into the mapping table cache in the host memory instead of loading the entire mapping table. In addition, by using the filesystem information for the replacement policy of the mapping table cache, its miss ratio is minimized. The proposed scheme is implemented by modifying the host-level FTL for open-channel SSDs of Linux and evaluated with Fio and Filebench. According to the evaluation results, the proposed scheme can achieve 80% of I/O performance with just 32% of memory usage in comparison with existing one. Moreover, the additional optimization utilizing host information for cache replacement policy increases I/O performance of the proposed scheme by 10%.

The remainder of this paper organizes as follows. Section 2 introduces open-channel SSDs and host-level FTL of Linux. Section 3 describes the design overview and detailed implementation of the proposed scheme. The experiment results are presented in Section 4. At last, Section 5 gives the conclusion of this paper.

2. Background

2.1 Open-channel SSDs

Open-channel SSD is proposed as a new class of SSDs to overcome shortcomings of conventional block

device interface SSDs. 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 2 briefly shows the internal structure of open-channel SSD. As shown in the figure, an open-channel SSD is composed of multiple channels, each of which has multiple Parallel Units (PUs). The PU is also represented as a Logical Unit Number (LUN) in a host-level FTL and used as the basic unit of parallel I/O processing in open-channel SSDs. In other words, each PU can process I/O requests independently from other PUs. Each PU consists of multiple chunks, each of which has multiple blocks. The chunks in the same location of each PU are grouped together to form a *line*. That is, the total number of *lines* of the open-channel SSD is the same as the number of chunks of each PU. A *line* is the basic unit that manages meta information in open-channel SSD.

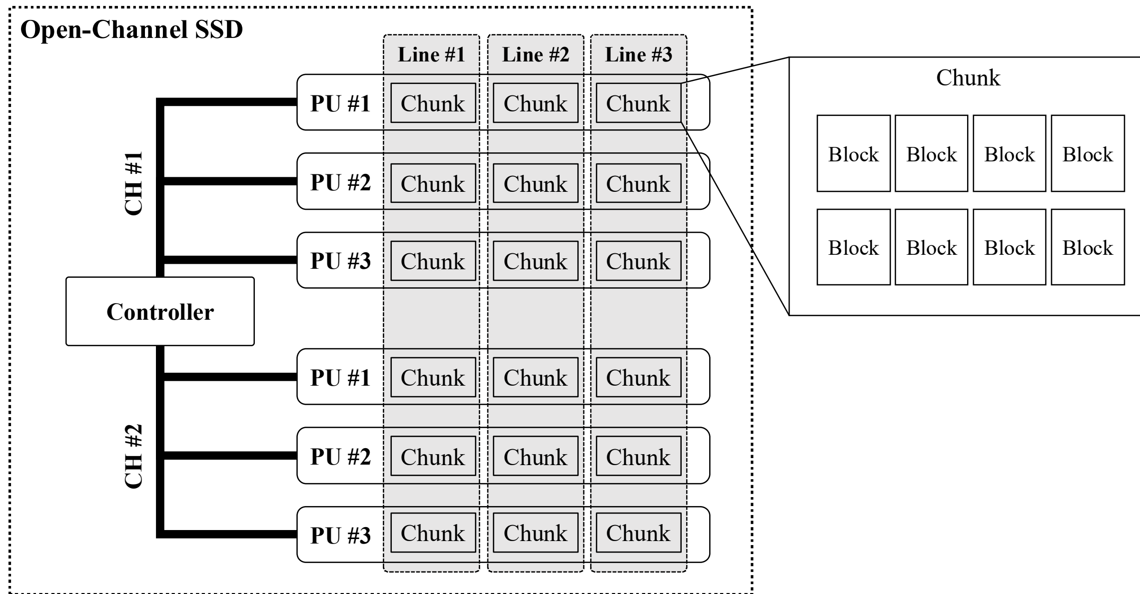


Figure 2. Internal structure of Open-channel SSD

In Linux, LightNVM subsystem [5] supports a host-level FTL called Pblk [6] for open-channel SSDs. The filesystem can transfer I/O requests to the open-channel SSD through Pblk while using the conventional block device interface. The Pblk handles page-level L2P mapping table management, address translation, and garbage collection. Since the entire page-level mapping table of the open-channel SSD should be loaded to host memory at the booting time, the memory usage increases linearly as the capacity of SSD increases. Therefore, in this paper, we implemented a memory efficient host-level FTL for open-channel SSDs.

3. Design and Implementation

3.1 Design Overview

In this section, we introduce overview of the proposed selective mapping table loading scheme. As you can see in Figure 3, the proposed scheme manages page-level mapping table of the open-channel SSD with three structures as follows: *mapping blocks* managed inside SSD, a *global directory* and *mapping table cache* on the host memory. The entire page-level L2P mapping table of open-channel SSD is divided into certain size of the block, *mapping block*, then stored into the SSD. Note that the size of *mapping block* is the same as the cache block size (e.g. 64KB) of the *mapping table cache*.

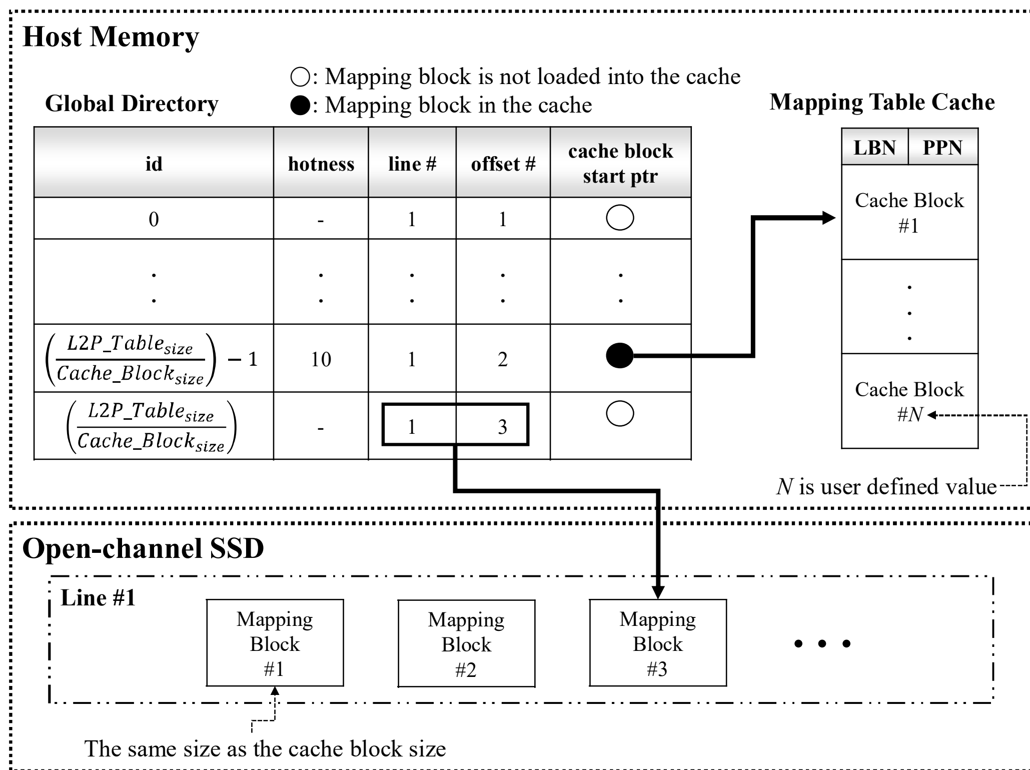


Figure 3. Overview of the selective mapping block loading scheme

The *mapping table cache* exists on the host memory and consists of multiple cache blocks. The total size of the *mapping table cache* is specified as a multiple of the cache block size defined by the user. The main role of the *mapping table cache* is to load *mapping blocks* currently required for translate the logical addresses of I/O requests into physical addresses of NAND flash memory. Moreover, as cache space runs out over time, the *mapping table cache* reserves free space in the cache by reclaiming low-priority cache blocks. The cache block replacement policy will be described in the next section.

The *global directory*, as the core of the proposed scheme, has various metadata for managing *mapping blocks* and the *mapping table cache*. Each entry of the *global directory* represents the information which consists of five metadata about each *mapping block* as follows: identifier of the *mapping block* (*id*), the popularity of each mapping block (*hotness*), the line where the *mapping block* exists (*line* #), the offset of the *mapping block* within the line (*offset* #), and a pointer to the corresponding cache block in the *mapping table cache*.

An identifier (*id*) of each *mapping block* is a unique value used to indicate the corresponding entry in a *global directory*. Because an identifier of *mapping block* is assigned sequentially, host can easily find identifier of *mapping block* including certain logical block number.

$$The\ number\ of\ mapping\ blocks = \left(\frac{L2P_Table_{size}}{Cache_Block_{size}}\right) \quad (1)$$

The number of *mapping blocks* follows Equation 1. Here, $L2P_Table_{size}$ and $Cache_Block_{size}$ indicates total size of page-level L2P mapping table of the open-channel SSD and user defined size of cache block, respectively. For example, in the case of a 1TB open-channel SSD, the total size of the page-level mapping

table is 1GB for a 4 bytes entry size and 4KB page size. Therefore, if user defined size of cache block is 64KB, the total number of *mapping blocks* is 16,384. By using this, we can get the size of the *global directory* required for 1TB open-channel SSD. For convenience of calculation, assuming the size of each entry of *global directory* is 40 Bytes, the amount of host memory required to build the *global directory* is 640KB, which is negligible.

The *hotness* refers to how often the host uses a particular cached *mapping block*. Note that the higher *hotness*, the more often the *mapping block* is used to translate LBN(Logical Block Number) to PPN(Physical Page Number). In the proposed scheme, the *hotness* of the *mapping block* is used as the priority of the cache block. Therefore, a *mapping block* with low *hotness* value has a higher probability of being reclaimed to the SSD compared to other cached *mapping blocks*.

In addition, the line number (*line* #) and the offset (*offset* #) of *global directory* is used to specify the location of the *mapping block* on the open-channel SSD. The line number refers to the physical line number of open-channel SSD indicating a set of chunks having consecutive physical blocks. On the other hand, the offset refers the relative offset of *mapping block* within the line. For example, "*line#=5, offset=3*" means that the location of the *mapping block* is 3rd one within the line 5.

Next, the pointer value in the *global directory* points to the cache block of *mapping table cache.* If the pointer value is not *null*, it means that the corresponding *mapping block* has been loaded to the *mapping table cache*. Therefore, we can directly translate the LBN to PPN without loading the *mapping block* including required mapping information. Otherwise, if the pointer value is *null*, the corresponding *mapping block* should be loaded into *mapping table cache*. At last, after required *mapping block* is loaded, the bitmap managing cache block status is updated. The next section introduces how to handle *mapping table cache* misses.

3.2 Handling Mapping Table Cache Miss

When certain LBN is requested from the block layer, the overall operation of the proposed scheme is shown in Figure 4. First, the entry of *global directory* for the *mapping block* corresponded to requested LBN is found by using the identifier of *mapping block*. The identifier of the *mapping block* can be obtained by dividing the LBN by the number of entries within single *mapping block*. For example, in Figure 4, because single *mapping block* has 4 entries, the identifier of the *mapping block* corresponded to the requested LBN (=7) is one. Then, the pointer value of the *global directory* should be checked to find if the mapping table has been loaded into

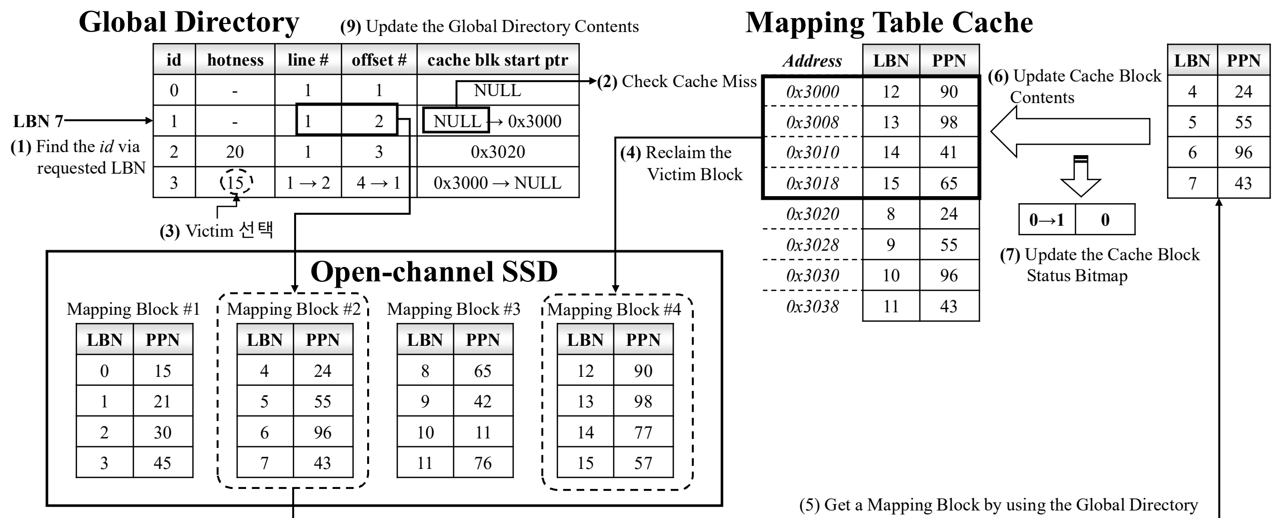


Figure 4. Cache miss handling process for mapping table cache

mapping table cache. If the pointer value is not *null*, cache hit has occurred. The LBN can be translated into PPN using the *mapping block* of the *mapping table cache*.

On the other hand, if the pointer value is *null*, a cache miss has occurred. Therefore, the required *mapping block* should be loaded to an empty cache block in the *mapping table cache* from SSD by using the line number and offset of the *global directory*. However, if there is no free cache block in the *mapping table cache*, the *mapping block* with the lowest hotness among the cached *mapping blocks* is reclaimed to the SSD to free up cache space. Note that the *hotness* allocation policy and the cache replacement policy will be described in the next section. At last, the *mapping block* including the requested LBN is retrieved and loaded into the *mapping table cache* from the SSD.

3.3 Hotness Allocation Policy

As mentioned above, since the proposed scheme stores only a part of the mapping table in the *mapping table cache*, I/O performance can be reduced due to *mapping table cache* miss. Therefore, miss ratio of *mapping table cache* should be minimized as much as possible. In the case of the existing cache management policies, the cache miss ratio can be decreased by determining the priority of cache blocks according to the temporal and spatial locality [10]. The proposed method allocates the *hotness* of each *mapping block* by utilizing host-side information.

In this paper, we noted that the access pattern of the data block and journal block of the journaling filesystem are different. Figure 5 briefly shows journaling process of Ext4 journaling filesystems [11]. Here, we assumes that Ext4 is set to *Journal* mode, not *Ordered* mode. As you can see the figure, the journal is written into separate journal space in storage device, not the entire block device. When a write request occurs, the journal (i.e. log for write request) should be written to the journal area before the requested file data is written to the storage device. Therefore, logical blocks including journal area are frequently overwritten because journal area is used as a circular buffer. In other words, LBN of journal area is more frequently accessed than data area. Moreover, journal writes take a large portion of the total write volume [12]. Consequently, if the mapping information for the journal area does not remain in the *mapping table cache*, the I/O performance is reduced because of excessive *mapping table cache misses*. Therefore, in this paper, mapping information for journal area is priorly kept in the cache as much as possible by using journal information of the host.

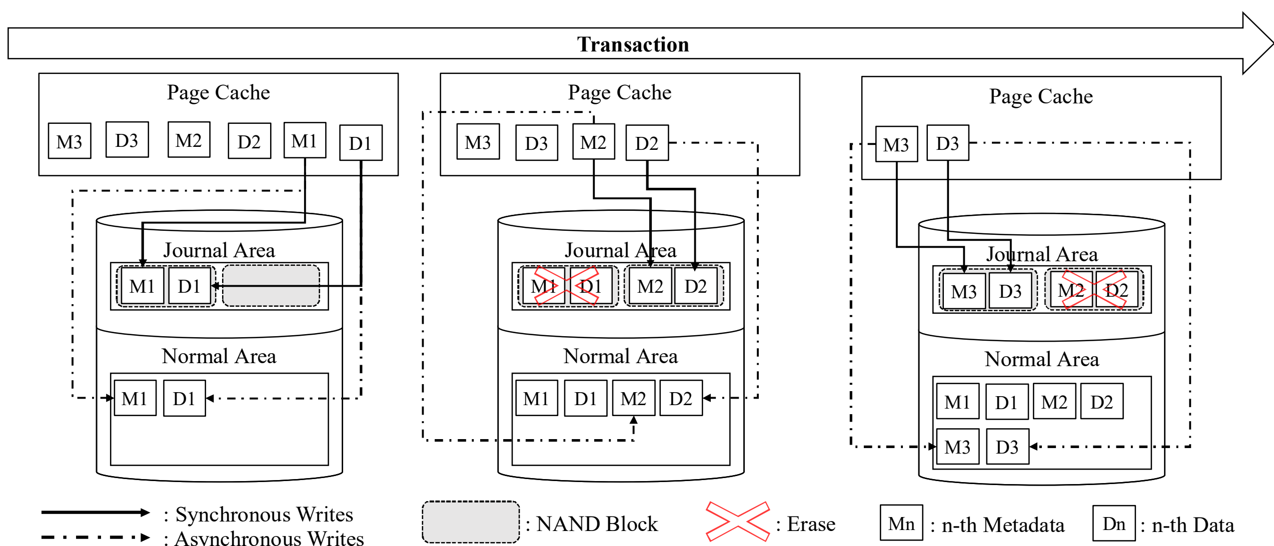


Figure 5. Journaling process of Ext4 filesystem

Moreover, to prevent thrashing and optimize cache replacement policy, when a *mapping block* is placed in the *mapping table cache,* a positive weight is given to the *hotness* to prevent the cache block from being selected as a victim block as soon as it is loaded. Otherwise, if the temporal locality of the *mapping block* is too low, a negative weight is given to the *hotness* because the *mapping block* is expected to not be used soon.

4. Experimental Results

For the performance evaluation, a virtual open-channel SSD is emulated in the QEMU virtual machine [13]. The host and the virtual machine environments used in the experiment are described in Table 1 and Table 2, respectively. As can be seen in Table 1, the host machine for running the QEMU virtual machine was equipped with two Intel Xeon CPUs and 32GB DRAM, and a high-performance NVMe SSD, Intel DC P4500, was used as the backend storage for the virtual open-channel SSD.

Table 1. Hardware and software specification

Components	Specification
CPU	Intel Xeon CPU E5-2620 v4 @ 2.10 GHz
Memory	32GB
Storage	Intel SSD DC P4500 1.0TB
OS (Kernel)	Ubuntu 18.04.2 LTS (Linux Kernel 5.3)

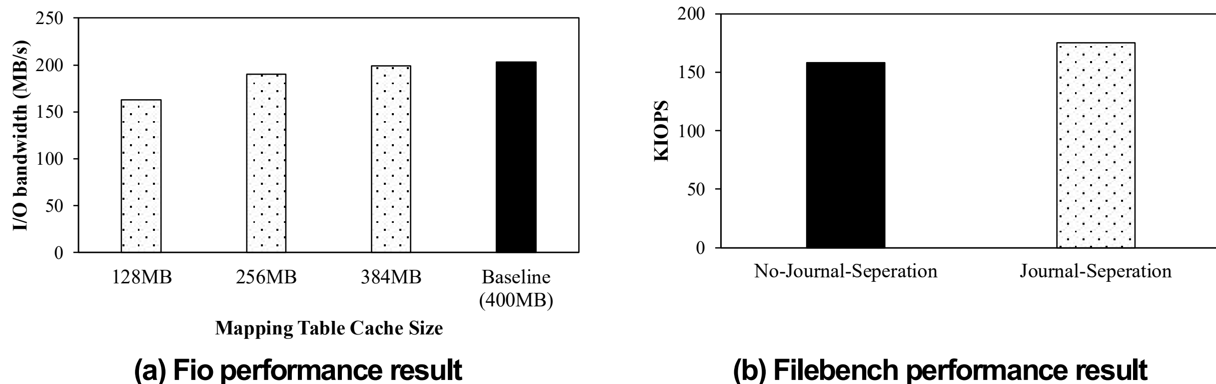
Table 2 shows that the QEMU virtual machine generated on the host machine has 16 CPU cores and 16GB DRAM. Moreover, the virtual open-channel SSD is created in a size of 400GB with a single channel, 1000 lines, and 16MB size of chunk. Therefore, the existing host-level FTL of Linux requires 400MB host memory to store page-level mapping table of the virtual open-channel SSD.

Table 2. QEMU virtual machine specification

Components		Specification
# of Cores		16
Memory		16GB
OCSSD	Capacity	400GB
	# of Channels	1
	# of Lines	1000
	Size of Chunk	16MB
OS (Kernel)		Ubuntu 16.04.5 LTS (Linux Kernel 4.16)

The proposed scheme is implemented on Pblk, host-level FTL of Linux for open-channel SSD. The performance evaluation uses both Fio [14] and Filebench [15] benchmarks. Figure 6(a) shows I/O performance of Fio random write workload according to pareto distribution with various size of *mapping table cache*. Pareto distribution is used because it can generate access pattern with locality.

As you can see in the figure, the proposed scheme achieves 80% I/O performance compared to existing Pblk while using only 32% memory usage. Figure 6(b) shows the I/O performance of Filebench OLTP workload with Ext4 filesystem. The experiment was performed with fixed the *mapping table cache* size of 64MB to verify the effectiveness of proposed *mapping table cache* management policy using journal information. Note that *mapping table cache* size is reduced because Filebench OLTP workload size is relatively small compared to Fio. As a result, the I/O performance of open-channel SSD is improved by 10%



(a) Fio performance result **(b) Filebench performance result**

Figure 6. I/O Performance evaluation results

by giving higher *hotness* to the journal data.

5. Conclusion

In this paper, we implemented a selective mapping table loading scheme that can reduce the host memory usage of open-channel SSDs. The proposed scheme loads only a currently required part of the *mapping blocks* to the *mapping table cache* in the host memory, so that memory overhead can be dramatically reduced. Moreover, to decrease *mapping table cache* miss ratio, cache replacement policy utilizes filesystem information and access history of *mapping blocks*. The proposed scheme is implemented on host-level FTL of Linux for open-channel SSD and evaluated with Fio and Filebench. According to the evaluation results, proposed selective mapping table loading scheme can achieve 80% I/O performance with just 32% memory usage in comparison with existing one. Moreover, cache replacement policy increases 10% of I/O performance of the proposed scheme by utilizing journal information and access history.

Acknowledgement

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2018R1D1A3B07050034) and the Basic Research Program through the National Research Foundation of Korea(NRF) funded by the MSIT(No. 2020R1A4A407985911)..

References

[1] M. Cornwell, “Anatomy of a Solid-state Drive: While the ubiquitous SSD shares many features with the hard-disk drive, under the surface they are completely different,” *Queue*, Vol. 10, No. 10, pp. 30-36, Oct. 2012. DOI: https://doi.org/10.1145/2381996.2385276

[2] J. Kim and S. Chung, “A Study on Flash Memory Management Techniques,” *The Journal of the Institute of Internet, Broadcasting and Communication(JIIBC)*, Vol. 17, No. 4, pp. 143-148, Aug. 2017. DOI: https://doi.org/10.7236/JIIBC.2017.17.4.143

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

[4] 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.

[5] 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

[6] Pblk: Host-based FTL for Open-channel SSDs. http://lightnvm.io/pblk-tools/.
[7] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, “SDF: Software-Defined Flash for Web-Scale Internet Storage Systems,” *ACM SIGPLAN Notices*, Vol. 49, No. 4, pp. 471–484, Feb. 2014.
DOI: https://doi.org/10.1145/2644865.2541959
[8] I. L. Picoli, C. V. Pasco, B. P. Jónsson, L. Bouganim, and P. Bonnet, “UFLIP-OC: Understanding Flash I/O Patterns on Open-channel Solid-State Drives,” in *Proc. of the 8th Asia-Pacific Workshop on Systems (APSys ‘17)*, pp. 1-7, Sep. 2-3, 2017.
DOI: https://doi.org/10.1145/3124680.3124741
[9] J. González and M. Bjørling, “Multi-Tenant I/O Isolation with Open-channel SSDs,” in *Proc. 8th Annual Non-Volatile Memories Workshop (NVMW ‘17)*, March 12-14, 2017.
[10] A. Gupta, Y. Kim, and B. Urgaonkar, “DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings,” *ACM SIGPLAN Notices*, Vol. 44, No. 3, pp. 229–240, 2009.
DOI: https://doi.org/10.1145/1508284.1508271
[11] 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.
[12] 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.
DOI: https://doi.org/10.7236/JIIBC.2016.16.6.177
[13] QEMU Open-channel SSD 2.0. https://github.com/OpenChannelSSD/qemu-nvme.
[14] Fio - Flexible I/O tester rev. 3.23. https://fio.readthedocs.io/en/latest/fio_doc.html.
[15] V. Tarasov, E. Zadok, and S. Shepler, “Filebench: A Flexible Framework for File System Benchmarking,” *USENIX ;login*, Vol. 41, No. 1, pp. 6–12, Apr. 2016.