Optimizing Host-level Flash Translation Layer with Considering Storage Stack of Host Systems

Jhuyeong Jhin
Sungkyunkwan University
Suwon, Korea
jjysienna@gmail.com

Hyukjoong Kim
Sungkyunkwan University
Suwon, Korea
wangmir@gmail.com

Dongkun Shin
Sungkyunkwan University
Suwon, Korea
dongkun@skku.edu

ABSTRACT

Many researchers have studied and proposed various kinds of host-level flash translation layer (FTL). Host-level FTL allows the host system to handle the internal architecture of solid state drives (SSDs), and thus tries to overcome the performance or the functionality failure of traditional SSDs which only handle the I/O requests with the block I/O interface. However, existing studies only focused on the functionalities of the host-level FTL while less researches have been done on how the FTL should work on the operating system (OS) rather than bare-metal firmware. From the observation, we found that existing host-level FTL suffers from the performance bottleneck caused by unscalable software design of FTL. Therefore, we propose an optimizing scheme which efficiently processes the I/O operations requested by multiple users and guarantees the scalability of the storage stack. Our experimental results show that the performance of our scheme improves the performance of software stack in twice or more compared to the existing host-level FTL.

CCS CONCEPTS

• **Information systems → Flash memory**; • **Software and its engineering → Secondary storage**; *Multithreading*;

KEYWORDS

Solid State Drives, Flash Translation Layer, Open-Channel SSDs

ACM Reference Format:
Jhuyeong Jhin, Hyukjoong Kim, and Dongkun Shin. 2018. Optimizing Host-level Flash Translation Layer with Considering Storage Stack of Host Systems. In *IMCOM '18: The 12th International Conference on Ubiquitous Information Management and Communication, January 5–7, 2018, Langkawi, Malaysia.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3164541.3164556

1 INTRODUCTION

Recently, the flash storage devices such as SSD have been used by various fields from mobile systems to server systems, and their performance and capacity have also been significantly improved. SSDs require a system software layer, called the flash translation layer (FTL), to manage the NAND flash memory of the SSDs. FTL is responsible for address translation, garbage collection (GC), and wear-leveling.

FTL has been implemented as internal firmware of the SSD device, exposing SSDs as a block device to provide the same block I/O interface to the host system. Therefore, the host system has no information on the internal architecture of the SSDs, and thus cannot exploit stack-wise optimization for the storage systems such as data placement or I/O scheduling. This limitation eliminates the possibilities for the predictable performance of I/O operations on SSDs even though the performance of flash memory operations–read, program, and erase–is predictable.

Recent studies have attempted to implement the FTL at the host system rather than inside of the SSDs [3, 5]. The host-level FTL manages the SSDs at the top of the storage interface. In this system, the SSDs expose their internals such as the physical architecture and the performance of the flash memory operations. Therefore, the SSD device can share the management of the storage with the host system such as address translation, GC and operation scheduling while the devices still are responsible for the low-level functionalities of the storage such as error correction code (ECC), wear-leveling, and bad block management.

In large-scale server systems, multiple users simultaneously request a lot of I/O operations to the storage. Therefore, the recent high-performance SSDs support NVMe interface [4] that can process lots of I/O commands in parallel with maximum 64K hardware queues. The I/O schedulers in the host systems also are advanced to multi-queue architecture such as blk-mq [2] to speed up the command processing of storage I/O. In addition, the SSDs have multiple parallel units, providing the scalability upon I/O operations. However the FTL, which tends to be a bare-metal firmware inside SSDs, is failed to scale up over multi-core system with OS. The host-level FTLs are often implemented as a single-threaded system daemon at the host system, thus the system with these FTL shows bad scalability upon multiple I/O operations. We analyzed the problems of the host-level FTL through the experiments using lightnvm [3] on Linux kernel, and found that its performance is degraded because it cannot process concurrent accesses of multiple users in parallel.

In this paper, we propose a noble host-level FTL called multi-Threaded FTL (MT-FTL) which services concurrent accesses of multiple users in parallel and provides system scalability. The experimental results show that the MT-FTL improves the scalability of existing host-level FTL.

2 BACKGROUND AND MOTIVATION

Recent storage interface supports multiple hardware queues to take full advantage of the high IOPS of the SSDs. The NVMe interface [4]

IMCOM '18, January 5–7, 2018, Langkawi, Malaysia

ACM ISBN 978-1-4503-6385-3/18/01…$15.00
https://doi.org/10.1145/3164541.3164556

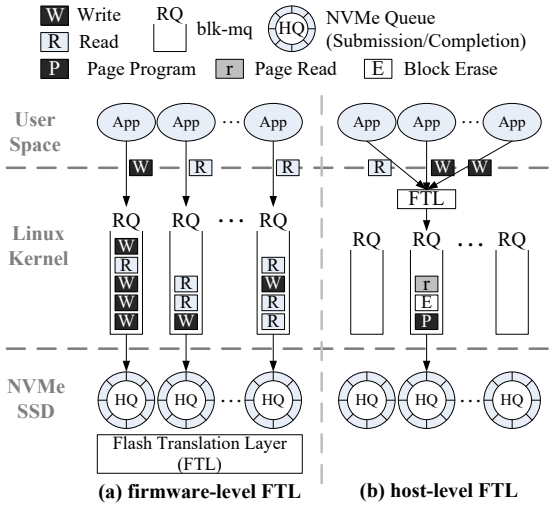

Figure 1: Comparison between firmware-level FTL and host-level FTL.

provides up to 64K hardware queues contrasting to SATA interface which provides a single queue. Using the NVMe interface, an user exclusively inserts the I/O requests into the one or more queues, which are dedicated to a core. At this time, the core executes the context of the user. Thus, the SSDs can simultaneously receive multiple I/O requests which are issued by multiple users at the same time. In addition, blk-mq separates the storage I/O path per each core, eliminating the bottlenecks caused by the contention between multiple users running on multiple cores.

2.1 Firmware-level FTL vs. Host-level FTL

Figure 1 shows the I/O processing in the system adopting firmware-level FTL and host-level FTL in the environment using blk-mq and NVMe interface. In Figure 1(a), since the user inserts an I/O request into the queue of the core which is running the context of the user (blk-mq), multiple I/O requests can be simultaneously pushed into the different cores without the contention to the same queue. These I/O requests are forwarded to the SSDs through multiple hardware queues of the NVMe interface in parallel. On the other hand, in Figure 1(b), the I/O requests has to go through the host FTL, before they are inserted into the per-core queue of blk-mq. Therefore, the host FTL causes the bottleneck while multiple users access the global data structure such as mapping table or buffers in the FTL.

2.2 LightNVM and pblk

LightNVM is the subsystem for Open-Channel SSD (OC-SSD) [3]. OC-SSD is the one of the new-class SSDs which expose their performance and internal architecture, and share their responsibilities to manage the storage space with the host FTL. LightNVM represents an abstraction of the partial functionalities of existing FTL as a target and the target is implemented in the Linux Kernel. We denote target as host FTL.

In LightNVM, the basic host FTL is pblk. pblk maintains page-level mapping table for address translation, and performs the write buffering. The write buffering is required because of the difference between the size of the sector on the host side and the size of the

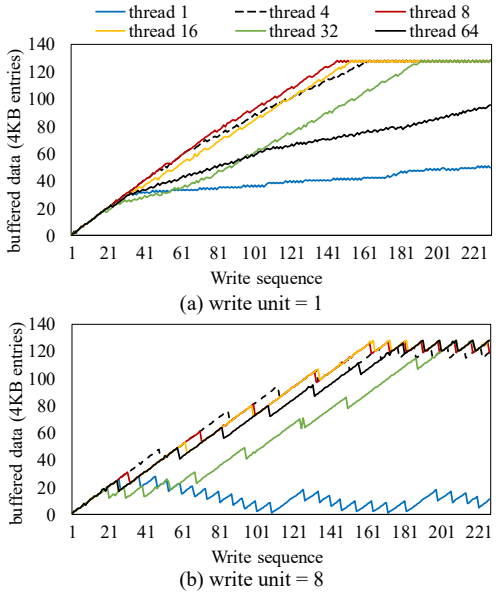

Figure 2: The consuming of buffered data with single writer thread.

physical page of NAND flash memory. pblk is also responsible to garbage collection (GC), wear-leveling, and bad block management. When users or write-back threads such as pdflush write their data, or the valid sectors are copied during GC, pblk divides all the data into 4KB entries and inserts them into a single buffer. At the time enough data is gathered in the buffer, a single thread, called writer thread, flushes the entries of the buffer into the storage.

2.3 Single-threaded Host-level FTL

In this paper, we investigate the problems of the current host FTL architecture through the experiments in LightNVM. We also observe the possibilities of the performance improvement.

2.3.1 Environment of Observational Experiment. The experiment was conducted on the server based on Intel Xeon processor (2.40GHz, 16 cores). For this experiment, we implemented a SSD Emulator based on LightNVM and allocated 4GB of DRAM memory for the storage emulator. The SSD Emulator exposes DRAM memory as a flash memory storage to LightNVM at the device driver, and uses pblk in the upper layer. We used fio benchmark and the workload consisted of random writes of 4KB unit. The write requests was performed as direct I/O and the number of the user threads which request the write operations varies 1 to 64. The number of the entries, which the buffer of pblk holds up to, is set 128.

2.3.2 Experimental results. We have observed how the I/O requests are accumulated in the single buffer managed by the single writer thread with the increasing number of the user threads. Figure 2 shows the amount of the I/O requested accumulated in the buffer as the I/O requests progress. In Figure 2(a), the number of the entries processed at a time (WU, Write Unit) is 1. In Figure 2(b), we increase WU to 8 in order to increase the consuming speed, at which the writer thread services the accumulated data in the buffer.

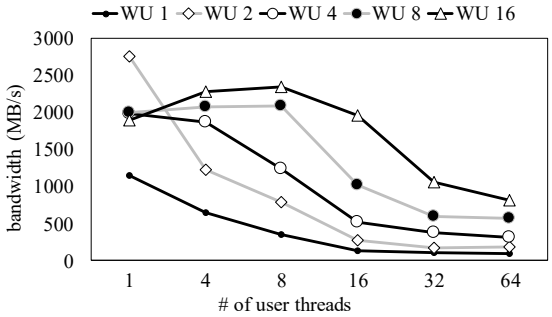

Figure 3: Performance of single threaded host-level FTL with various number of write units.

As shown in the Figure 2(a), when the single user makes the write requests, the producing speed, at which the I/O requests are accumulated in the buffer, is slower than when the multiple users make the write requests. In the other words, the producing speed is accelerated with the increasing number of the user threads. Even though the DRAM memory can sufficiently support the corresponding bandwidth, the bottlenecks occur in the buffer because of the architecture of pblk, which handles the write requests with the single buffer and the single writer thread. In the Figure 2(b), the entries do accumulate not too much in the buffer, as the consuming speed is faster than the producing speed of a single user thread. However, the I/O requests are still accumulated in the buffer as the number of the user threads increases even if the consuming speed is accelerated.

WU and the number of the user threads affect the actual performance. Figure 3 shows the bandwidth variation according to WU and the number of the user threads. In the most cases, WU is proportional to the I/O bandwidth regardless of the number of the use threads. This is because the user threads have less time to wait for the free space in the buffer because of the faster consuming speed. However, the I/O bandwidth significantly decreases with the more users even if WU increases. The cause of this degradation can be found in the architecture of pblk, a single buffer and a single writer thread. Even if multiple users request the write operations at the same time, only one user thread having the lock of the buffer can access at a time. Moreover, the I/O requests accumulated in the buffer can be consumed by only one writer thread. The architecture of pblk causes bottlenecks.

3 MULTI-THREADED HOST-LEVEL FTL

In this paper, we propose Multi-Threaded Host FTL(MT-FTL) to provide the scalability of the system and handle the I/O operations in parallel, which are requested by the multiple users at the same time. Figure 4 shows the current host FTL and MT-FTL, respectively. The current host FTL has a single-threaded architecture in which the multiple users cannot access the buffer of the host FTL in parallel. MT-FTL manages the per-core buffers and the per-core writer threads as many as the number of the CPU cores. A user thread inserts data to write into the buffer which is dedicated to the CPU core which executes the user thread. Even if the multiple users simultaneously send the write request, the bottleneck caused by the single buffer of the existing host FTL can be eliminated because the

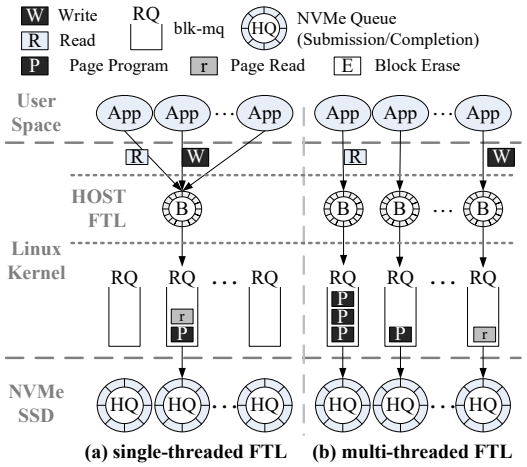

Figure 4: Comparison between single-threaded and multi-threaded FTL.

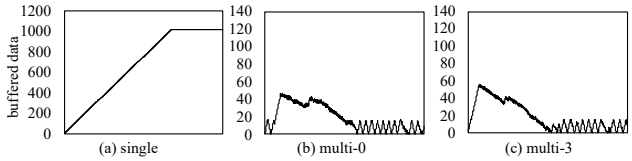

Figure 5: The consuming of buffered data on single writer (a) and multi writers (b-c).

user threads insert their request into the different buffers. In addition, since the number of the writer threads is equal to the number of the CPU cores, the scalability of the system can be improved.

Figure 5 shows how data is accumulated in the buffer(s) of pblk and MT-FTL when the 8 user threads write simultaneously. Only some section of the whole experiment time is shown. In both pblk and MT-FTL, WU is set to 1 and the total number of the buffer entries is 1024. Each per-core buffer in MT-FTL has a total of 128 entries because we experiment with 8 CPU cores.

Comparing Figure 5(a) with (b) and (c), the buffer fills up faster in the pblk and this phenomenon lasts until the 8 user threads no longer request the write operations. In MT-FTL, each buffer does not fill up, and the accumulated data shrinks faster than in pblk.

In pblk, multiple user threads access a single buffer at the same time and insert the entries at a high speed. Since there is only one writer thread servicing the buffer, the consuming speed is relatively slower than the producing speed. After the buffer is full, the user who wants to insert an entry into the buffer has to wait until there is the free space in the buffer. On the other hand, MT-FTL has the per-core buffers and the per-core writer threads. A user thread inserts the entries into the buffer of the core which is executing the user's context. When the unique writer thread of that buffer wakes up on the core, it services the entries of the buffer.

That is, when there are 8 cores and 8 user threads are writing some data at the same time, the ratio of user thread, buffer, and writer thread is 8:1:1 in pblk, and 1:1:1 in MT-FTL. Since the producing speed to each buffer and the consuming speed does not differ much in MT-FTL, there are usually free space in each buffers as shown in Figure 5(b, c). Thus, the user threads do not have to waste time waiting for the free space of the buffer. They also do

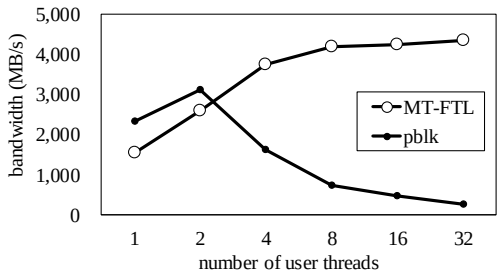

Figure 6: Performance of pblk and MT-FTL.

not share the data structures related to the buffers with the user threads running on different cores.

4 EXPERIMENT

We evaluate and compare the performance of pblk of LightNVM and MT-FTL, and analyze the user and writer thread(s) behavior per core.

4.1 Experimental Setup

The experiments are conducted on an Intel Core^{TM} I7-7700 processor (3.60GHz, 8 cores). We use our SSD Emulator and fio benchmark [1]. The workload consists of random writes of 4KB unit. The write requests are performed as direct I/O, and the number of the user threads varies 1 to 32. The total number of the buffer entries is 1024. MT-FTL manages 8 buffers and 8 writer threads, and each buffer has up to 128 entries.

4.2 Experimental results

Figure 6 shows the I/O bandwidth of pblk and MT-FTL according to the number of the user threads. In pblk, the performance drops sharply as the number of the user threads increases. As shown in Figure 3, the performance degradation occurs even if the consuming speed is accelerated by increasing WU. In contrast, MT-FTL significantly improves the performance with the increasing number of the user threads.

There are two main reasons for this result. The first is the difference of the method in which the users access the buffer. In pblk, multiple user threads access a single buffer at the same time. On the other hand, MT-FTL has the per-core buffers as many as the number of the cores. Therefore, the user threads running on different cores exclusively access the buffer, which is dedicated to the cores. The second is the difference in the consuming speed. In pblk, the producing speed becomes faster with the increasing number of the user threads, even if the consuming speed is almost constant. As the difference of the speed of consuming and producing increases, the free space remained in the buffer is quickly reduced. The user threads spend time to wait until there is free space in the buffer. In MT-FTL, since only one user thread inserts entries into the buffer of one core executing the context of the user, there is no difference between the producing and consuming speed even if the number of the user threads increases. Therefore, each buffer usually has free space, reducing the amount of time the users wait for free space in the buffer.

This difference can be seen more clearly in Figure 7. In this experiment, pblk is full when the experiment is over 0.004 seconds from

(a) single

(b) multi

Figure 7: The behavior of the single-threaded FTL e.g., pblk (a) and the multi-threaded FTL (b).

the start. Figure 7 shows that the entries are produced/consumed per core in pblk and MT-FTL from 0.004 seconds to 0.008 seconds, respectively. In the Figure 7(a), the user threads cannot insert an entry continuously in pblk. Even though the writer thread continues to consume the buffer entries on core #2, this phenomenon lasts. This is because there is not enough space to insert new entry into the buffer. In the Figure 7(b), the user threads continue to insert more entries even after 0.004 seconds in MT-FTL, because there is enough free space in each buffer to insert the entries.

5 CONCLUSION & FUTURE WORK

We observed that existing host-level FTL cannot efficiently handle concurrent accesses of the multiple users. We optimize the architecture of the host-level FTL to support multiple I/O operations requested by multiple users, and provide the scalability of the system. The performance of host-level FTL is improved by mitigating the competitive accesses of multiple users to the buffer in contrast to existing host FTL.

Our experimental results show the performance is slightly lower than that of existing host FTL when the number of the user threads is one or two. To resolve this problem, we will design and apply better multi-threaded architecture to improve the performance and the scalability.

ACKNOWLEDGEMENTS

This research was supported by the MSIT(Ministry of Science and ICT), Korea, under the SW Starlab support program(IITP-2017-0-00914) supervised by the IITP(Institute for Information & communications Technology Promotion)

REFERENCES

[1] Jens Axboe. 2015. Flexible io tester. *hups://github. comlaxboe/fio* (2015).
[2] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*. ACM, 22.
[3] Matias Bjørling, Javier González, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem.. In *FAST*. 359–374.
[4] NVMe. 2011. NVMe: Non-volatile Memory Express Interface. http://www.nvmexpress.org/. (2011).
[5] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined Flash for Web-scale Internet Storage Systems. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014), 471–484. https://doi.org/10.1145/2654822.2541959