IEICE TRANSACTIONS

on Information and Systems

VOL. E102-D NO. 8
AUGUST 2019

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

LETTER

NVRAM-Aware Mapping Table Management for Flash Storage Devices[*]

Yongju SONG[†], **Sungkyun LEE**[†], **Dong Hyun KANG**[††], ***Nonmembers*, *and* Young Ik EOM**[†a)], ***Member***

SUMMARY Flash storage suffers from severe performance degradation due to its inherent internal synchronization overhead. Especially, flushing an L2P (logical address to physical address) mapping table significantly contributes to the performance degradation. To relieve the problem, we propose an efficient L2P mapping table management scheme on the flash storage, which works along with a small-sized NVRAM. It flushes L2P mapping table from DRAM to NVRAM or flash memory selectively. In our experiments, the proposed scheme shows up to 9.37× better performance than conventional schemes.

key words: *flash storage, L2P mapping table, synchronization command, non-volatile RAM*

1. Introduction

Nowadays, flash memory-based storage devices have become the mainstream storage in various fields due to its higher performance and less power consumption. However, flash storage devices have an inherent constraint that it cannot support in-place update. When a write request updates data pages, flash storage has to reflect them into new physical pages and then invalidate old pages. Thus, the logical and physical addresses of flash pages might be different. To solve this constraint, flash storage equips with special embedded software, called Flash Translation Layer (FTL), where it translates a logical page address (LPA) to its physical page address (PPA). A single <LPA, PPA> pair is called a mapping entry and FTL manages all the mapping entries in the form of a mapping table [1]. Meanwhile, FTL caches the mapping table into the internal DRAM (write buffer) of the flash storage device in order to reduce I/O latency. However, when a sudden power-off occurs, the cached data in DRAM which is volatile can be lost, and thus FTL necessarily needs to ensure the durability of data in DRAM through synchronization commands. Unfortunately, this operation brings negative impact on the write performance of the flash storage devices.

To relieve the aforementioned problem, non-volatile RAM (NVRAM) devices (e.g., PCM or STT-MRAM) can be used as a write buffer of the flash storage devices. NVRAM has opened up opportunities to improve the write performance with its attractive features such as non-volatility, byte-addressability, and DRAM-like fast read/write latency. By the way, there have been several studies focusing on how to employ NVRAM in the storage-level. To improve the write performance and lifetime of flash storage, Wei and Kang [2], [3] transform a pattern of incoming I/O requests into a sequential pattern by using NVRAM as a write buffer. Lee [4] also considers NVRAM as a write buffer and a read cache, thereby avoiding useless copy operations of valid pages during garbage collection.

In this paper, we introduce a flash storage architecture equipped with a small-sized NVRAM in the write buffer layer of the storage, and then propose how to manage mapping entries to improve the write performance by employing the NVRAM. To efficiently handle the mapping entries, we first group multiple contiguous mapping entries into a single mapping segment. Typically, when a host calls a synchronization command, FTL directly flushes all dirty mapping entries into the flash memory. However, our scheme selectively copies them into NVRAM depending on how many dirty mapping entries exist on a mapping segment. Therefore, since NVRAM has lower latency than flash memory and also offers non-volatility, this strategy can guarantee the enhanced write performance and data durability. Our scheme is implemented on a solid-state drive development board, called Jasmine OpenSSD [5]. Experimental results show that the proposed scheme outperforms the conventional scheme by up to 9.37× in terms of throughput.

2. Motivation

To ensure the durability of mapping table inside flash storage devices, FTL has to reflect it into the flash memory with synchronization commands. Upon synchronization commands, FTL writes data pages first and also flushes their mapping entries related to the written pages into the flash memory. Unfortunately, this operation spends considerable time, thereby leading to the write performance degradation of the flash storage devices. In this operation, we particularly concentrated on the overhead of guaranteeing the durability of the mapping table.

Manuscript received January 25, 2019.
Manuscript publicized April 26, 2019.

[†]The authors are with the Department of Electronic, Electrical and Computer Engineering, Sungkyunkwan University, Suwon, Korea.
[††]The author is with the Department of Computer Engineering, Dongguk University-Gyeongju, Gyeongju, Korea.

[*]This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (No. NRF-2015M3C4A7065696) and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (No. NRF-2017R1A2B3004660).

a) E-mail: yieom@skku.edu (Corresponding author)

DOI: 10.1587/transinf.2019EDL8022

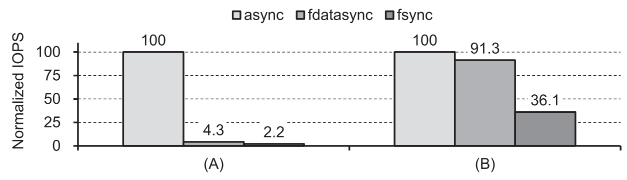


Fig. 1 IOPS during synchronization command execution, for two cases: (A) with flushing the mapping table, and (B) without flushing it.

Table 1 Comparison of memory technologies [6]–[9].

	STT-MRAM	PCM	DRAM	Flash
Non-volatility	Yes	Yes	No	Yes
Read latency	10 ns	20 ns	10 ns	25 μs
Write latency	10 ns	100~150 ns	10 ns	200 μs
Cost ($/Gb)	30~70	0.16	0.6	0.03
Density (Gb/cm^2)	0.36	13.5	9.1	185.8
Endurance (cycles)	10^{12}	10^{12}	10^{15}	10^3

As shown in Fig. 1, we conducted motivation experiments, measuring the impact of the aforementioned problem. To this end, we implemented two types of flash storage devices by modifying the internal FTL layer of the OpenSSD. One flushes the mapping table from DRAM to flash memory when synchronization commands are triggered (Experiment A), whereas the other does not perform the flushing operation (Experiment B). We used `fsync` and `fdatasync` system calls for synchronization commands. As a result, the write performance of experiment B is up to 16.4× and 21.2× higher compared to the experiment A, when a host issues `fsync` and `fdatasync`, respectively. Consequently, we confirmed that an operation of guaranteeing the durability of the mapping table incurs considerable performance overheads.

Meanwhile, NVRAM technologies can mitigate the aforementioned overhead with their advantages such as non-volatility, byte-addressability, and high performance. Table 1 presents properties of the emerging memory technologies briefly. As shown in Table 1, NVRAM can offer potential benefits than flash memory in terms of much lower read/write latency and higher endurance, but it has relatively lower density and expensive cost. Additionally, unlike DRAM, all NVRAM technologies prevent data loss against power failure or system crash due to their non-volatility. More specifically, there are slightly different properties among their types. In the case of STT-MRAM, it can meet the performance of DRAM, but its low-density limits its capacity. On the contrary, PCM is denser than DRAM and may enable the use of large capacity NVRAM, whereas it shows somewhat higher read/write latency. In this paper, we focused on how FTL can efficiently manage the mapping table by exploiting the characteristics of NVRAM devices.

3. Design

In this paper, we aim to efficiently manage the mapping table on the flash storage device. Our scheme includes NVRAM in the write buffer layer of the flash storage device and we exploit it as a dedicated in-storage buffer for the mapping table, as depicted in Fig. 2.

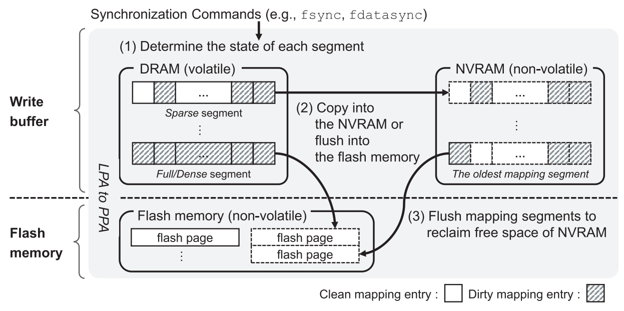

Synchronization Commands (e.g., fsync, fdatasync)
(1) Determine the state of each segment
Write buffer
DRAM (volatile)
Sparse segment
Full/Dense segment
(2) Copy into the NVRAM or flush into the flash memory
NVRAM (non-volatile)
The oldest mapping segment
LPA to PPA
Flash memory
Flash memory (non-volatile)
flash page
(3) Flush mapping segments to reclaim free space of NVRAM
Clean mapping entry : □ Dirty mapping entry : ▨

Fig. 2 Overall design of our scheme.

3.1 Categorizing Mapping Segments

Generally, commercial flash storage devices have 4–16 KB sized pages, and each mapping entry is a few bytes in size (e.g., 4 or 8 bytes). Thus, upon synchronization commands, the way that flushes a few bytes of mapping entries from DRAM to flash memory would be extremely inefficient, because the flash memory has to perform read/write operations at a page granularity even though the mapping entries are as small as a few bytes. In order to alleviate this inefficiency, we first attempt to handle several contiguous mapping entries in a unit of a group, called a mapping segment, and categorize them into four states: *clean*, *sparse*, *dense*, and *full*.

An initial state of a mapping segment is *clean*, which means that there is no dirty entry. Once any mapping entry in the *clean* segment is updated, its state is changed from *clean* to *sparse*. Then, as the number of dirty entries in the segment is increased gradually, the segment becomes *dense*. At last, if all the entries of the segment turn into dirty, this segment goes to the *full* state. After a dirty segment is copied to non-volatile media (i.e., NVRAM or flash memory) by synchronization commands, the state of the segment turns into *clean* again.

3.2 Handling Mapping Segments

When the host calls a synchronization command (e.g., `fsync`, `fdatasync`), our scheme identifies the state of each mapping segment which resides in DRAM and runs accordingly. Algorithm 1 explains how we handle mapping segments upon synchronization commands.

First, in case of *full* segment whose all entries are dirtied, we flush this type of segment into the flash memory, because the FTL can minimize the inefficiency of page writes on flash memory, as described in the previous subsection. Therefore, a mapping segment whose state is *full* might be the most preferred segment type to store it directly into the flash memory among the four kinds of mapping segments (Line 3–5). Similarly, since a *dense* segment also has many dirty entries, it can be a good candidate to flush into flash

Algorithm 1: Handling mapping segments.

```
   /* D_SEGMENT: a mapping segment in DRAM,
      NV_SEGMENT: a mapping segment in NVRAM      */
1  foreach D_SEGMENT do
2      switch state of the D_SEGMENT do
3          case full or dense do
4              flush the D_SEGMENT into flash memory
5          end
6          case sparse do
7              if there is no free space in NVRAM then
8                  find the oldest NV_SEGMENT;
9                  flush it into flash memory;
10             end
11             copy the D_SEGMENT into NVRAM;
12         case clean do nothing;
13     end
14 end
```

memory (Line 3–5). However, the case of *sparse* segment is different, where there are small number of dirty entries relatively. If the FTL flushes a *sparse* segment into the flash memory as in the case of *full* or *dense* segment, it inevitably causes the inefficiency of flushing a large amount of unnecessary data into the flash memory. The reason is that the size of dirty mapping entries in a *sparse* segment is very small when we compare it with the flash page size. Thus, we decided to copy the *sparse* segment into the NVRAM upon synchronization commands (Line 6–11). At this point, there may be insufficient free space in NVRAM for saving the incoming mapping segments (Line 7). In that case, we need to reclaim the free space of NVRAM by evicting some mapping segments, which is described in the next subsection. Lastly, for *clean* mapping segment, we do not perform any extra action because there is no new data for durability (Line 12). Consequently, when the host triggers a synchronization command, our scheme flushes the *full* or *dense* segments into the flash memory, whereas the *sparse* segments are copied into the NVRAM.

3.3 Reclaiming Free Space of NVRAM

As mentioned earlier, if the NVRAM does not have enough free space when the host invokes synchronization commands, we try to secure the free space for incoming mapping segments (Line 7–10). To select a mapping segment in the NVRAM to be evicted into the flash memory, we apply the least-recently-used (LRU) replacement policy. In our implementation, we provide a *segment age* value for each mapping segment, which refers to how long it has been maintained in the NVRAM. The *segment age* of the mapping segment is set to zero when it is copied into the NVRAM for the first time, and it is increased by one whenever a synchronization command is invoked. Once a mapping segment is re-accessed in NVRAM during runtime, we reset its *segment age* to zero based on the LRU policy. Using the *segment age* value, the FTL can secure the NVRAM space by evicting the mapping segment with the highest *segment age* in NVRAM into the flash memory.

Table 2 The parameters of Jasmine OpenSSD.

Parameter	Value
SSD capacity	8 GB
Page / Block size	8 KB / 1024 KB (128 pages)
Mapping entry / segment / table size	4 bytes / 8 KB / 4 MB
Flash page read / write latency	400 μs / 1300 μs
DRAM access latency	400 ns

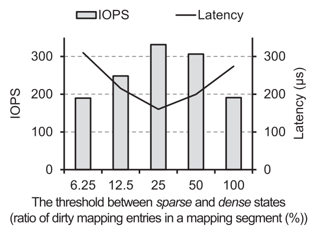


Fig. 3 Impact of different thresholds between *sparse* and *dense* states.

4. Evaluation

To measure the performance benefits of our scheme, we used an SSD development platform, called Jasmine OpenSSD. It consists of SATA 2.0 host interface, 64 MB DRAM, and Samsung K9LCG08U1M 8GB MLC NAND flash package [5]. The detailed configurations of Jasmine OpenSSD are listed in Table 2. Also, we assume that the type of employed NVRAM is STT-MRAM because its performance is comparable with DRAM. All experiments were performed on Linux kernel 4.4.0 and Ext4 file system with ordered mode journaling.

First of all, we conducted an experiment to get the appropriate threshold value between the *sparse* and *dense* states. For this, we evaluated IOPS and I/O latency of the flash storage by adjusting the threshold value from 6.25% to 100% under the synchronization-intensive workload. As mentioned earlier, if the ratio of dirty mapping entries in a *sparse* segment exceeds a certain threshold, its state becomes *dense*. As shown in Fig. 3, when the threshold between *sparse* and *dense* states is set to 25%, IOPS of the flash storage device can be increased by up to 1.7× compared to the case of 6.25%, achieving the peak performance. Based on this result, we decided the threshold to be set to 25%; that is, the state of a mapping segment whose ratio of dirty mapping entries is greater than 25% goes to *dense* state. Following experiments were conducted with this threshold value. In addition, since the size of NVRAM can affect the write performance of our scheme, we performed the experiments varying it from 25% (1 MB) to 100% (4 MB) relative to the size of whole mapping table.

Synthetic Workload We first evaluated the write performance of the proposed scheme by using FIO benchmark, which generates a synthetic workload that issues synchronization commands frequently. Figure 4 shows IOPS normalized to the case of asynchronous writes. Our scheme improves IOPS by up to 4.22× and 4.08× for `fsync` and `fdatasync`, respectively, compared to the conventional

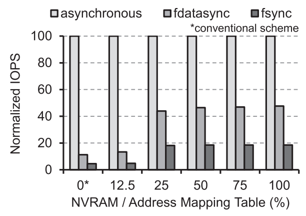
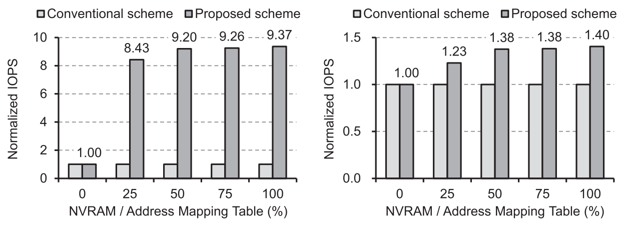


Fig. 4 Experimental results with FIO benchmark.

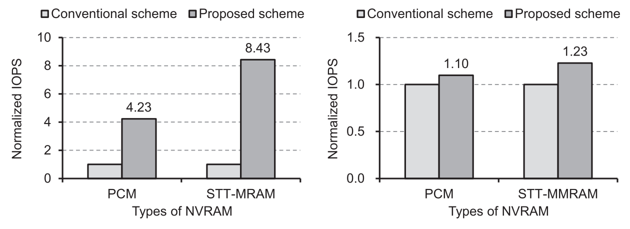


(a) Varmail workload (b) OLTP workload

Fig. 5 Experimental results with Filebench benchmark.

Table 3 Characteristics of workloads.

Workloads	Avg. number of written mapping entries per synchronization command	Number of synchronization commands
Varmail	869	2251
OLTP	153	2904



(a) Varmail workload (b) OLTP workload

Fig. 6 Experimental results of the proposed scheme with different types of NVRAM.

scheme, when the size of the NVRAM is large enough to accommodate all the mapping entries. Even if the NVRAM capacity is only a quarter of the total mapping table size, the proposed scheme can increase the write performance by up to 3.90× and 4.00× for `fsync` and `fdatasync`, respectively. It shows that the write performance improves with the higher capacity of NVRAM because we can handle greater number of mapping segments in the high-performance NVRAM as the capacity of NVRAM increases. However, there is no notable performance gap between them, whether the proposed scheme uses small- or large-sized NVRAM. Therefore, we believe that adding small-sized NVRAM (about a 25% of the mapping table size) with our scheme will be the best choice in terms of cost-efficiency.

Real-world Workloads We also measured IOPS of our scheme using Varmail and OLTP workloads from Filebench suite which behaves as real-world workload patterns. Varmail mimics an email server and OLTP emulates a transaction processing workload with a large number of small files. It is well known that both workloads include heavy synchronization commands. Figure 5 shows the performance gain of our scheme under these two real workloads. Compared to the conventional scheme, our scheme improves I/O throughput from 8.43× to 9.37× for Varmail workload with varying NVRAM size. Likewise, in the case of OLTP workload, the proposed scheme achieves from 1.23× to 1.40× higher IOPS than that of conventional scheme. The reason is that our scheme selectively copies mapping entries into the high performance NVRAM instead of flushing into flash memory, whereas the conventional scheme always flushes them into the flash memory, when synchronization commands are issued. Another noticeable result is that our scheme shows more enhancements for Varmail workload than for OLTP workload. This is because Varmail workload leads to more significant overhead on synchronization commands than the OLTP workload, as shown in Table 3. In other words, with larger synchronization overhead, our scheme can be more highly effective.

Employing Different NVRAM In the previous experiments, we assumed that the proposed scheme employed STT-MRAM as the type of NVRAM. If the flash storage device requires large-size mapping table, STT-MRAM may be inappropriate for our scheme due to its expensive cost and low density. In this case, PCM can be an eligible substitute because it offers high density at a lower cost and also shows improved endurance as much as that of STT-MRAM as a wide variety of materials have been developed (e.g., PVD $Ge_2Sb_2Te_5$), as shown in Table 1 [9]. So, we measured IOPS assuming that our scheme adopts PCM as the type of NVRAM device instead of STT-MRAM. In fact, the write latency of PCM is reported as 100~150 ns, while that of STT-MRAM is 10 ns [6], [8]. Since this latency gap can give a big impact on the write performance of our scheme, we added a delay of about 100 ns to every NVRAM write operation. Also, we set the capacity of NVRAM to 25% (1 MB) of the whole address mapping table size and used Varmail and OLTP workloads of Filebench. Figure 6 presents the normalized IOPS of our scheme for each type of NVRAM. Even though we impose a delay of around 100 ns to the NVRAM access for emulating PCM, our scheme achieved up to 4.23× and 1.10× more IOPS than the conventional scheme for Varmail and OLTP workloads, respectively. Even with considering the several characteristics of NVRAM types, our scheme is still practical and effective.

5. Conclusion

We found that the operation of ensuring the durability of the mapping table significantly degrades the write performance of the flash storage devices. To mitigate this overhead, we introduced a scheme for mapping table management on the flash storage architecture equipped with a small-sized

NVRAM. When a synchronization command is issued, our scheme tries to selectively copy dirty mapping segments into the NVRAM instead of the flash memory. Our experimental evaluations show that the write performance of the proposed scheme is increased by up to 9.37×, compared to the conventional scheme.

References

[1] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices," Proc. USENIX Conference on File and Storage Technologies, pp.49–66, 2018.

[2] Q. Wei, C. Chen, and J. Yang, "CBM: A cooperative buffer management for SSD," Proc. 30th IEEE Symposium on Mass Storage Systems and Technologies, pp.1–12, 2014.

[3] D.H. Kang, S.J. Han, Y.-C. Kim, and Y.I. Eom, "CLOCK-DNV: A write buffer algorithm for flash storage devices of consumer electronics," IEEE Trans. Consum. Electron., vol.63, no.1, pp.85–91, 2017.

[4] E. Lee, J. Kim, H. Bahn, S. Lee, and S.H. Noh, "Reducing write amplification of flash storage through cooperative data management with NVM," ACM Trans. Storage, vol.13, no.2, pp.1–13, 2017.

[5] "The OpenSSD Project," http://www.openssd-project.org/

[6] M.H. Kryder and C.S. Kim, "After hard drives—What comes next?," IEEE Trans. Magn., vol.45, no.10, pp.3406–3413, 2009.

[7] K. Suzuki and S. Swanson, "A survey of trends in non-volatile memory technologies: 2000–2014," Proc. IEEE International Memory Workshop, pp.1–4, 2015.

[8] "Analysts Weigh in on Persistent Memory." https://www.snia.org/sites/default/files/PM-Summit/2017/presentations/Coughlin_Handy_Analysts_Weigh_in_on_PM.pdf/

[9] S.W. Fong, C.M. Neumann, and H.-S.P. Wong, "Phase-change memory–Towards a storage-class memory," IEEE Trans. Electron Devices, vol.64, no.11, pp.4374–4385, 2017.