Exploiting Exclusive Higher Resolution to Enhance Response Time of Embedded Flash Storage

Jung-Hoon Kim[a] and Young-Sik Lee[b]
Device Solution Research, Samsung Electronics Co., Hwasung, Republic of Korea

Keywords: NAND Flash Memory, Embedded Flash Storage, Flash Translation Layer, Internal Overhead, Response Time.

Abstract: NAND flash-based embedded storage may spend a long time on responding to a host storage system. Most of the flash translation layers (FTLs) of the embedded flash storage utilize a granular page-mapping level. However, they did not pay heed to page mapping management that causes the internal overhead of the page-level FTL. This overhead might damage the response time, especially after the random writes to the embedded flash storage. In this paper, we propose a novel method to reduce the internal overhead related to the page mapping write. This method exploits a virtually-shrunk segment exclusively to the page mapping table, which is implemented by our mapping-segmented flash translation layer (MSFTL). One mapping segment is intrinsically composed of consecutive page mappings smaller in size than a logical page of the host system. As a result, MSFTL drastically reduces the amount of page mapping data written and therefore improves both the average and worst response time compared with the fine-granularity page-level FTLs.

1 INTRODUCTION

Over the past decade, an embedded flash storage device (e.g., UFS (Standard, 2018) or eMMC (Whitaker, 2015b)) has been able to adopt NAND flash memory by leveraging many mapping schemes (Ban, 1995; Gupta et al., 2009; Kim et al., 2002; Lee et al., 2006; Lee et al., 2007). Among the mapping schemes, a flash translation layer (FTL) of the embedded flash storage arguably considers a page mapping as the best mapping scheme, which serves storage space in smartphones, autonomous vehicles, and Internet-of-Things (IoT) devices. Because the page mapping is a granular component, the page-level FTL can easily combine the other techniques improving the performance and lifetime of NAND flash-based storage. However, the granular page details give birth to a large page mapping table.

To manage the large page mapping table containing all the page mappings, the page-level FTL uses both volatile and non-volatile memory in the embedded flash storage. When preparing for translating the page mapping locations, the page-level FTL should first locate the page mappings in the volatile memory. And, if the page mappings are for the user data written by the host system, the page-level FTL should record the corresponding page mappings into the volatile memory. Besides, to keep the consistency (Moon et al., 2010) of those up-to-date page mappings, the page-level FTL has to store the page mappings into the non-volatile memory, such as NAND flash memory.

The data amount of the page mappings to be written may rise in the embedded flash storage as one of the internal overheads. When storing the page mappings into NAND flash memory, the page-level FTL should use NAND page programming soliciting data as much as a NAND page (Grupp et al., 2009). In this constraint, the page-level FTL needs to maximize the throughput of storing the page mappings updated into one NAND page. However, even when storing a few page mappings updated, the page-level FTL saves them by filling the insufficient amount with data, such as other page mappings unchanged. Furthermore, since the NAND page size has increased (Kim et al., 2015b; Takeuchi, 2009) continually by NAND fabrication evolution, the NAND page enlarged might exacerbate the insufficient amount to store the page mappings and degrade the performance of the embedded flash storage.

Many researchers have studied several techniques (Lee et al., 2017; Lee et al., 2013; Ma et al., 2011; Park and Kim, 2011) based on the page-level

[a] https://orcid.org/0000-0002-4369-866X
[b] https://orcid.org/0000-0003-3095-3729

Kim, J. and Lee, Y.
Exploiting Exclusive Higher Resolution to Enhance Response Time of Embedded Flash Storage.
DOI: 10.5220/0009859804700477
In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT 2020)*, pages 470-477
ISBN: 978-989-758-443-5

FTL to reduce the internal overhead. They have inspected the behavior of the data, such as the data locality and I/O request size, and tried to improve the garbage collection overhead of the page-level FTL. However, as well as conducting the garbage collection, the page-level FTL should govern the page mappings and preserve them in NAND flash memory. To improve the internal overhead of the page mapping writes, the page-level FTL could set the granularity to the smaller mini-page size (Lv et al., 2018). However, if the granularity decreases by half, even twice the resources (e.g., volatile and non-volatile memory) should be needed for the page mapping table. As another technique, the page-level FTL might employ the sub-page programming (Kim et al., 2015a) storing data with a smaller size than one NAND page. But, this technique decreases the NAND page size physically and focuses on improving the lifetime of NAND flash memory.

In this paper, we state a new mapping segmentation method that is to exploit a virtually-shrunk segment. Exposing this small segment exclusively to the page mapping table makes a higher resolution. Therefore, the page-level FTL can adjust the page mappings more precisely without being more granular to its page-mapping level. Our mapping-segmented flash translation layer (MSFTL) integrates this novel method into the page-level FTL. Because the data amount of the segment is less than the NAND page, MSFTL collects it as much as the necessary data amount for the NAND page programming. The evaluation results show that MSFTL drastically reduces the data amount of page mappings written and therefore improves both the average and worst response time compared with the fine-granularity page-level FTLs.

2 BACKGROUND AND MOTIVATION

Because of the advantage of the small page granularity, the page-level FTL conducts the data transactions lightly. This flexibility also gives many chances to improve the performance and lifetime of NAND flash-based storage (Gupta et al., 2009; Kim et al., 2002; Lee et al., 2017; Lee et al., 2013; Lee et al., 2006; Lee et al., 2007; Lv et al., 2018; Ma et al., 2011; Park and Kim, 2011). By referring to the host system, such as an operating or a file system, the page-level FTL defines the page mapping granularity as the 4-KiB size nowadays. However, this small page granularity gives birth to a large page mapping table in NAND flash-based storage. For example, 1-TiB NAND flash-based storage needs the 1-GiB space if an entry of the page mapping table is four bytes. Consequently, the page-level FTL manages this entire page mapping table to operate the page mapping-level scheme correctly.

The embedded flash storage, as one of the NAND flash-based storages, employs both volatile and non-volatile memory to keep the page mappings consistently. The volatile memory works as the mapping cache and holds only a fraction of the entire page mapping table. When the host system or the internal operation, such as a garbage collection (GC) of the page-level FTL, writes its data to NAND flash memory of the embedded flash storage, the page-level FTL updates the corresponding page mappings into the mapping cache first. After that, to prevent the data loss of the page mappings from the event of a power cycle, the page-level FTL intentionally stores the page mappings into the non-volatile NAND flash memory. We define this writing activity for the page mappings as a *mapping flush*. The mapping flush is typically related to the booting time (Whitaker, 2015a; Zhang et al., 2014) of the embedded flash storage. Besides, because of the small amount of the volatile memory in the embedded flash storage, the page-level FTL usually writes the page mappings when they are evicted from the mapping cache by the cache policy.

When conducting the mapping flush, the page-level FTL writes the page mappings by using the NAND page programming of NAND flash memory. Because NAND flash memory operates with its peculiarities (Grupp et al., 2009), such as erase-before-program constraint, unbalanced operation latency, and different fixed unit sizes of operations, the page-level FTL ensures these characteristics of NAND flash memory. For these reasons, when performing the NAND page programming, the page-level FTL fills the necessary minimum amount with the page mappings, which is commonly called a *translation page* since the demand-based flash translation layer (DFTL) (Gupta et al., 2009). The translation page consists of continuous page mappings of the page mapping table (PMT) similar to the simple linear addressing organization (Wei et al., 2015). Based on the NAND page size, the page-level FTL determines the number of the page mappings in the translation page. For instance, Figure 1 depicts PMT0 that starts from a logical page number 0 (LPN0) and ends up in each last LPN (i.e., LPN4095, LPN2047, or LPN1023).

Utilizing the latest three-dimensional (3D) NAND flash memory with the 16-KiB NAND page size, we have implemented the baseline page-level FTL, which is called Base-DFTL. Base-DFTL manages the entire page mapping table with the translation pages written in the translation blocks. The mapping flush of Base-

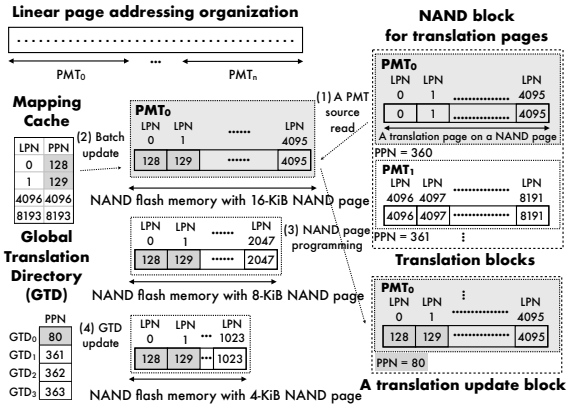

Figure 1: A page mapping write used in typical page-level FTLs with translation pages.

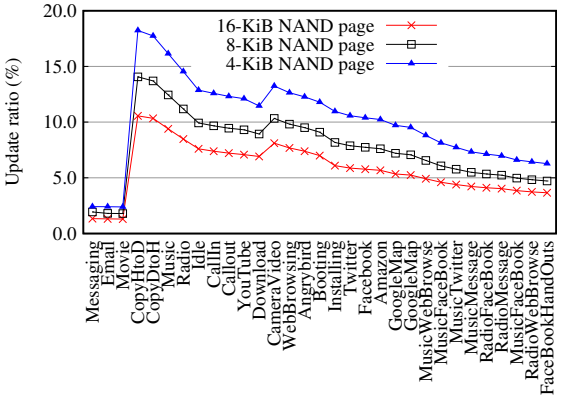

Figure 2: Update ratios measured by the smartphone I/O traces on various NAND page sizes.

DFTL starts from (1) and ends to (4) described in Figure 1. Because LPN0 and LPN1 belong to PMT0 are updated in the mapping cache, the PMT0 as the source for the NAND page programming is read from the translation block first (1). Base-DFTL refers to the global translation directory table (GTD) to look up the PMT0 location in the translation blocks. After the read of the PMT0 source, since two LPNs were updated, Base-DFTL transfers two physical page numbers (PPNs) of the LPNs into the NAND page buffer (2), which is called a *batch update*. Lastly, Base-DFTL creates the new translation page of PMT0 by using the NAND page programming (3-4). Because Base-DFTL copied the PMT0 source, we can notice that almost every page mapping of the new translation page is the same as the one of the PMT0 source. However, this process satisfies the minimum data amount with the page mappings of PMT0 to complete the NAND page programming. Actually, during the mapping flush, Base-DFTL may have many page mappings updated among the different PMTs. Therefore, the page mappings updated merely from the one PMT source might increase the internal overhead.

To analyze the data amount of the page mappings updated in the new translation page, we have evaluated the Base-DFTL with 31 application I/O traces of the smartphone. Section 4 explains the detail parameters of this experiment. As a result of the experiment shown in Figure 2, the ratio of the page mappings updated in the 16-KiB NAND page programming averages a 5.8% ratio. Because this updated ratio was meager, we realized that the larger size of the translation page did not benefit from the mapping flush. As another experiment, we replaced the 16-KiB NAND page with either the small 8-KiB or 4-KiB NAND page and re-evaluated it with the same parameters. These results still show that the updated average ratios are 7.7% and 10.1%, however. This reason has motivated us to reduce this internal overhead caused by storing the page mappings into NAND flash memory.

3 MSFTL ARCHITECTURE

3.1 Virtually-shrunk Mapping Segment

MSFTL is based initially on Base-DFTL to manage the same page mapping table. However, to mitigate the data amount relevant to one PMT source, MSFTL exploits a virtually-shrunk segment exclusively to the PMT. Because this segment is smaller in size than a logical page (i.e., 4 KiB), MSFTL lowers the data amount of the page mappings written by using this segment, which is called a *mapping segment*. Through the small mapping segment, MSFTL makes a higher resolution expanding the range of choice in the page mapping table. In our experiments, the mapping segment has virtually shrunk as several multiples of 1 KiB (i.e., 1 KiB and 2 KiB). Note that the 1-KiB is a minimum data size protected by the error correction code (ECC) of the internal hardware controller on the embedded flash storage.

Figure 3 compares the PMT to a mapping segment table (MST) separated by the virtually-shrunk segment. As we know it, there was no segment (NS) in the Base-DFTL. However, the two-segment (2S) case by MSFTL cuts a NAND page into halves holding the mapping segment each. Likewise, the 16-segment (16S) case separates the NAND page into sixteen regions. By being more granular with the segment, MSFTL can detail the MST grasping the page mappings. By the way, whenever MSFTL stores the MST into NAND flash memory, there is a situation to fill the minimum data amount for conducting the NAND page programming. The following section describes

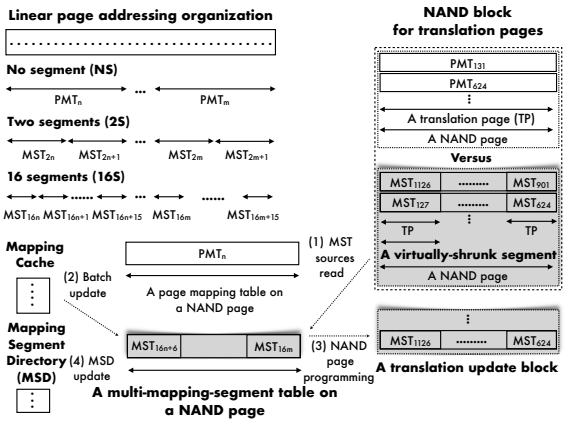

Figure 3: Virtual-Shrunk segmentation and the collection of the mapping segments for the page mapping write.

how to gather the small mapping segments in detail.

3.2 Mapping Flush for Mapping Segments

MSFTL aims to store all the MSTs containing any page mappings updated. During the mapping flush, whenever the condition of the minimum data amount accepts, MSFTL conducts the NAND page programming. Because this collection technique merges the different MSTs into one NAND page, the number of the page mappings updated increases naturally on the NAND page programming. Therefore, MSFTL takes advantage of the collection technique to improve the updated ratio commented in Section 2.

We depict the collection of the MSTs in Figure 3. Instead of using one PMT source of Base-DFTL, MSFTL gathers several MST sources for one NAND page programming. First, MSFTL fetches the corresponding MST sources by looking up the mapping segment directory table (MSD). Then, all the page mappings belong to the MSTs are updated from the mapping cache. If one NAND page buffer is full of the MSTs, the NAND page programming executes. This collection of the MSTs increases the number of the updated page mappings on one NAND page programming. We also describe this processing sequence from (1) of Figure 3 to (4).

When gathering several MSTs, MSFTL could raise the additional overhead not happened in Base-DFTL. During the batch update of one PMT source, Base-DFTL spends one NAND page read timing on loading the translation page from NAND flash memory. However, MSFTL should collect the different MSTs until one NAND page buffer is full. If the MSTs have spread to several different NAND pages, MSFTL consumes more NAND page read timing than Base-DFTL. By the way, in addition to the NAND page read timing, the mapping flush contains other remaining processes to be done. In Table 1, we can notice that a NAND page programming timing is ten times longer than the NAND page read. Because MSFTL reduces the number of the NAND page programmings as many as the number of the MSTs collected, MSFTL lowers the entire operation time of the mapping flush.

3.3 Mapping Segment Directory Table

MSFTL uses the mapping segment directory (MSD) table to direct the locations of the MSTs written in NAND flash memory. Because the number of the MSTs is more than that of the PMTs, the MSD table size increases, and it correlates to one MST size. As the MST size becomes small, the MSD table size increases linearly. And the higher storage capacity enlarges the MSD table size as well. If we apply a 128-GiB storage capacity, NS uses 32-KiB space and, 16S needs (i.e., 1-KiB MST) 512-KiB space sixteen times of NS. However, this space is worth bring a higher resolution to the page mapping table. So then, MSFTL caches the entire MSD table to the volatile memory. Similar to this approach, we can consider the fine-granularity FTL, such as the mini-page level FTL (Lv et al., 2018). However, this smaller page-mapping granularity needs 512-MiB space to adopt the same 1-KiB level.

To keep the MSD table consistency after a power cycle, MSFTL stores the MSD table to the translation block along with the MST. After writing the MSTs to the translation block, MSFTL records the locations of the MSTs in the MSD table of the volatile memory first. Whenever this MSD table changes, MSFTL does not store the MSD table immediately into the translation block, however. Because MSFTL can recover the MST locations by scanning the translation block, MSFTL stores the entire MSD table sparsely before conducting the GC of the translation blocks.

3.4 Garbage Collection for Translation Blocks

MSFTL utilizes the update block scheme to store the MST and MSD table. If there is no free space in the translation update block, MSFTL conducts the GC of the translation blocks with the traditional greedy policy (Rosenblum and Ousterhout, 1992). As the victim block for the GC, MSFTL chooses one of the translation blocks. To retain the free space maximum after conducting the GC, MSFTL considers the number of up-to-date MSTs held in the translation block. There-

fore, among all the translation blocks, the number of up-to-date MSTs in the victim block is the smallest.

We realize that the number of MSTs increases by shortening the PMT. In the 16S case, the total number of MSTs is sixteen times as large as that of NS. During the GC of the translation block, MSFTL relocates all the up-to-date MSTs of the victim block into a free block. However, the up-to-date MSTs could exceed the free page limit, such as the number of free pages in a free block.

To resolve this problem, MSFTL assembles the up-to-date MSTs of the victim block during the GC of the translation block. This process is similar to the MST collection of the MSFTL mapping flush. The more granular MSFTL raises the number of MSTs. However, MSFTL minimizes the internal overhead by employing this kind of MST collection method.

4 EXPERIMENTS

4.1 Evaluation Methodology

To investigate the detail of the page mapping writes in the page-level FTLs, we have implemented a trace-driven FTL simulator that processes block-level traces of the host system. The first page-level FTL is the baseline Base-DFTL that ordinarily allocates one NAND page for the translation page, which is identical to the DFTL scheme (Gupta et al., 2009). The second one is the mini-page FTL (Lv et al., 2018) using the logical small page-level granularity as the translation page (i.e., 4 KiB). Lastly, we evaluate MSFTL exploiting two types of the segment, such as 2 KiB and 1 KiB. They are called as MSFTL-2KiB and MSFTL-1KiB, respectively. To conduct the mapping flush along with the user writes, we nominate 9 MiB that accounts for the data amount written to the embedded flash storage as much as one NAND block size. And, the logical page-mapping granularity is the same as 4 KiB in all the FTLs.

We have simulated 128-GiB NAND flash-based storage by using four 3D NAND flash chips (Samsung, 2014). Table 1 describes the features of the NAND flash chip. Because the response time of the embedded flash storage is highly relevant to the performance of NAND flash memory, we calculate the FTL operation time, lastly, with the NAND performance parameters such as the NAND access time of Table 1. Note that the over-provisioning space (Smith, 2012) is about 5.0% of the entire capacity made from four NAND flash chips. The block layout of the over-provisioning space is composed of the translation update block, the user update block, the translation blocks, and the free blocks. The remaining blocks become the user data space visible as the embedded flash storage capacity to the host system.

Table 1: Characteristics of 256-Gb TLC 3D NAND flash.

NAND Structure	Page Size	16 KiB
	Block Size	9 MiB
	Block Count	3776
NAND Access Time	Page Read	49 μs
	Page Program	0.6 ms
	Block Erase	4.0 ms
	Byte Transfer	1.25 ns

First, we evaluate two kinds of synthesized data patterns to analyze the aspect of page mappings written by the three different page-level FTLs. These synthesized patterns are composed of sequential and random address patterns. And then, in the end, we experiment with the data patterns of the real-device smartphone. Because the smartphone workloads logged from various kinds of applications (Zhou et al., 2015), the data patterns of the smartphone show us mixed address patterns with sequential and random data patterns. Using these data patterns, we measure the data amount of page mappings written, which reflects the update ratio mentioned in Section 2.

4.2 Analysis with Synthesized Data Pattern

We measure the data amount of page mappings written by using the sequential data pattern. The trace-driven FTL simulator writes 126.14-GiB with a 16-KiB chunk consecutively to the FTL, which starts from the clean-slate state. In our evaluation, the clean-slate state means that the FTL has never written the data after the FTL simulator initialized.

In the result of the sequential pattern, Base-DFTL stored 339.9-MiB page mappings to NAND flash memory. This data amount is only 0.3% of the entire data written, however. Because the batch update by the sequential patterns has updated almost every page mapping in the translation page, this data amount of the page mappings written should be small in quantity. Though, both the mini-page FTL and MSFTL reduced that amount. Since they have collected the page mappings spread between two adjacent PMTs, the data amounts of the page mappings written reduced to 224.3 MiB. Therefore, the mini-page FTL and MSFTL drop the page mappings written by 33.3% of Base-DFTL. As we expected, the data amount of the directory and the MSD table written has increased slightly. Consequently, the mini-page FTL lowers them totally by 32.6% of Base-DFTL. Because of the enlarged MSD table, MSFTL reduces them by either 32.2% or 28.1%.

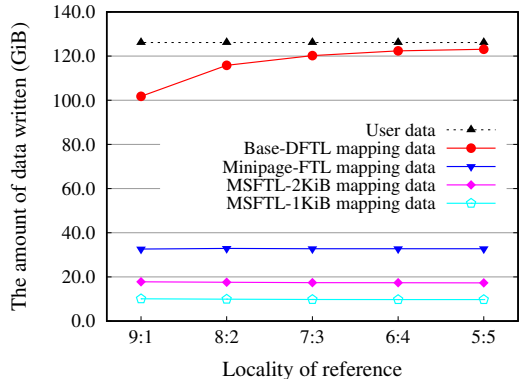

Figure 4: A comparison of every data amount written with various spatial localities.

To start over with the random address pattern, we access data with several degrees of the spatial locality. The degree of the spatial locality is different in the size of the user space and the data amount written to that space. For example, a high spatial locality (i.e., 9:1) represents that many user data write to the small user data space. And, a uniform spatial locality (i.e., 5:5) means that we write data evenly to the entire user data space. Similar to the sequential data pattern, the data write operates with the 16-KiB chunk. And, the total amount of data written is 126.14 GiB as the whole user data space.

Figure 4 shows every data amount written, which consists of the user and page mapping data. Contrary to the sequential results, the data amount of page mappings written to NAND flash memory is considerable. Because the batch update by the random patterns has only updated a few page mappings in the translation page, the mapping flush should write the page mappings updated in many different translation pages by using NAND page programming. Especially in the result of Base-DFTL, the data amount of page mappings written reached to 97.6% of the amount of user data written. Thanks to the small translation page, the mini-page FTL dropped the page mappings written by 48.5% of Base-DFTL.

Differently, MSFTL adopts the virtually-shrunk segment without extending resources for the page mapping table, such as the mini-page FTL scheme. The smaller segment of MSFTL has more advantage of raising the resolution to the specific page mapping area. Therefore, MSFTL-1KiB reduced the data amount of page mappings written by up to 92.1% in the random patterns. The data amounts of the GTD and MSD table written were quite small, by the way. Their averages were less than 1.0% of the total page mapping data written. After this section, the data amount of page mappings written includes the GTD and MSD table data.

4.3 Analysis with Real-device Workloads

Finally, we evaluate three different page-level FTLs by using the smartphone workloads of embedded flash storage. These workloads consist of 31 block-level I/O traces logged from 18 typical applications (e.g., Email and Twitter) of a Nexus-5 smartphone (Zhou et al., 2015). The I/O traces in (Zhou et al., 2015) show a mixed pattern with the sequential and random address. However, most of the data patterns are close to the randomness, and the amounts of data are not sufficient to provoke the GC in the clean-slate state. So, the FTL simulator writes a certain amount of data with the synthesized random pattern in advance. Through this precondition, the FTL can conduct the GC when the FTL simulator injects the smartphone workloads into the FTL. The amount of data written is a quarter, half, or full of the entire user storage space. The x-axis of Figure 5 differentiates the precondition. Lastly, we evaluate the FTLs with the smartphone workloads in these preconditions.

In the results of the smartphone workloads shown by Figure 5, three different FTLs show the same amount of user data written by the smartphone workloads. However, the precondition escalates the data amount of the GC data written. This GC worsens the data amount of page mappings written along with the random patterns of the I/O traces. Consequently, it reaches up to 177.9% of the user data written, especially in Base-DFTL.

By utilizing the small logical page granularity of the host system to the translation page, the mini-page FTL reduces the data amounts of the page mappings written. However, MSFTL brings more enhanced results than the mini-page FTL by adopting the virtually-shrunk segment that details the granularity, especially for the page mappings. Figure 5 compares the results evaluated by Base-DFTL, the mini-page FTL, and MSFTL. MSFTL-1KiB drops the entire amount of data written by 31.4%, 48.2%, and 58.8% of Base-DFTL in the quarter, half, and full case, respectively. Moreover, MSFTL-1KiB improves by up to 8.5%, 16.0%, and 23.0% of the mini-page FTL.

To clarify the performance gain occurred by lowering the data amount of page mappings written, we judge the operation time with the performance parameters of NAND flash memory, such as the NAND access time of Table 1. And we assume the data I/O transaction sharing the dual-channel NAND flash controller (SiliconMotion, 2019). Because the embedded flash storage spends most of the time in the data I/O transaction to NAND flash memory, we ex-

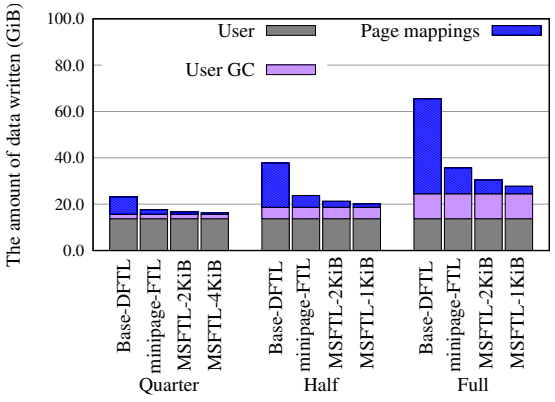

Figure 5: Total data amounts of page mappings written by the mapping flush.

amine every response time of a request, which is about 270K requests from all the I/O traces.

Figure 6 presents the cumulative distribution result for the response time. Both the mini-page FTL and MSFTL reduce all the response times of the smartphone workloads. The mini-page FTL improves the average response times by up to 28.9%, 37.1%, and 44.3% of Base-DFTL in the quarter, half, and full cases, respectively. However, MSFTL-1KiB improves them by up to 36.3%, 43.4%, and 56.1% by exploiting the virtually-shrunk segment. Also, MSFTL benefits from a higher resolution to decrease the worst response time. MSFTL-1KiB drops them by up to 85.3%, 71.6%, and 65.1% of Base-DFTL. As a result, MSFTL-1KiB improves by up to 41.9%, 26.8%, and 27.7% of the mini-page FTL.

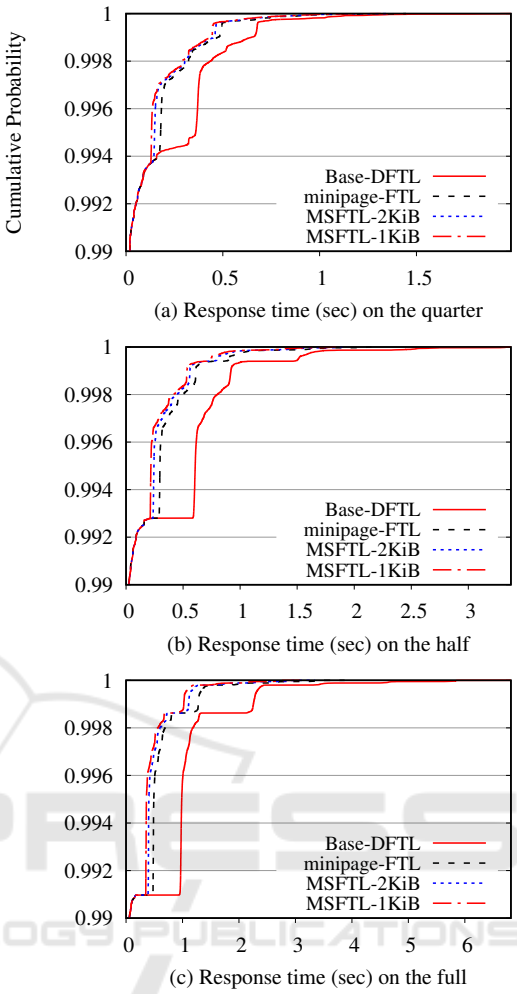

Figure 6: Cumulative distribution result for the response times spent by NAND flash memory.

5 RELATED WORK

There have been many studies improving the performance and lifetime of NAND flash-based storage by using the page-level FTL. These studies (Gupta et al., 2009; Kim et al., 2002; Lee et al., 2017; Lee et al., 2013; Lee et al., 2006; Lee et al., 2007; Lv et al., 2018; Ma et al., 2011; Park and Kim, 2011) utilized the user data behaviors, such as the temporal data locality, the data preemption, the data compression, and the data caching. For example, DFTL (Gupta et al., 2009) exploits the significant temporal locality and finally reduces the amount of the GC data written. A semi-preemptible GC (PGC) scheme (Lee et al., 2013) allows the GC preemption to hide the GC cost. And, the new NVM cache technique (Lee et al., 2017) mitigates the GC overhead. For a data reduction, zFTL (Park and Kim, 2011) utilized the data compression with the prediction scheme that differentiates incompressible data in advance. However, these kinds of proposals focused on the user data but did not pay attention to the page mappings that they should manage.

As the aspect of the page mapping-level granularity, the sub-page programming (Kim et al., 2015a) used a fraction of the NAND page to improve the lifetime of NAND flash memory. Since this method has limited the amount of data written, at least two NAND page programmings should be needed to write the data as much as one NAND page. To overcome the enlarged physical NAND page, the new PM-FTL (Lv et al., 2018) was developed by utilizing the smaller mini-page granularity. However, this kind of smaller mapping scheme needs vast amounts of resources to handle the entire page mapping table.

6 CONCLUSION

To enhance the response time of the embedded flash storage, we exploit a virtually-shrunk segment exclusively to the page mapping table. Our novel mapping-segmented flash translation layer (MSFTL) implements the page-level FTL combined with the new mapping segmentation method. The mapping segment of MSFTL is composed of consecutive page mappings with a smaller size than the logical page of the host system. When storing the mapping segments, MSFTL gathers every mapping segment that has any updated page mappings. As a result, MSFTL reduces the data amount of page mappings written by up to 58.8% compared with the fine-granularity page-level FTLs. Finally, MSFTL improves both the average and worst response time by up to 56.1% and 85.3% in the real-device smartphone workloads.

REFERENCES

Ban, A. (1995). Flash file system. Patent No. 5,404,485, Filed Mar. 8th., 1993, Issued Apr. 4th., 1995.

Grupp, L. M., Caulfield, A. M., Coburn, J., Swanson, S., Yaakobi, E., Siegel, P. H., and Wolf, J. K. (2009). Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, pages 24–33. IEEE.

Gupta, A., Kim, Y., and Urgaonkar, B. (2009). Dftl: A flash translation layer employing demand-based selective caching of page-level address mappings. *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*.

Kim, J., Kim, J. M., Noh, S. H., Min, S. L., and Cho, Y. (2002). A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375.

Kim, J.-H., Kim, S.-H., and Kim, J.-S. (2015a). Subpage programming for extending the lifetime of nand flash memory. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 555–560. IEEE.

Kim, J.-Y., Park, S.-H., Seo, H., Song, K.-W., Yoon, S., and Chung, E.-Y. (2015b). Nand flash memory with multiple page sizes for high-performance storage devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):764–768.

Lee, E., Kim, J., Bahn, H., Lee, S., and Noh, S. H. (2017). Reducing write amplification of flash storage through cooperative data management with nvm. *ACM Transactions on Storage (TOS)*, 13(2):1–13.

Lee, J., Kim, Y., Shipman, G. M., Oral, S., and Kim, J. (2013). Preemptible i/o scheduling of garbage collection for solid state drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(2):247–260.

Lee, S.-W., Choi, W.-K., and Park, D.-J. (2006). Fast: An efficient flash translation layer for flash memory. In *International Conference on Embedded and Ubiquitous Computing*, pages 879–887. Springer.

Lee, S.-W., Park, D.-J., Chung, T.-S., Lee, D.-H., Park, S., and Song, H.-J. (2007). A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18.

Lv, H., Zhou, Y., Wu, F., Xiao, W., He, X., Lu, Z., and Xie, C. (2018). Exploiting minipage-level mapping to improve write efficiency of nand flash. In *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–10. IEEE.

Ma, D., Feng, J., and Li, G. (2011). Lazyftl: a page-level flash translation layer optimized for nand flash memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1–12. ACM.

Moon, S., Lim, S.-P., Park, D.-J., and Lee, S.-W. (2010). Crash recovery in fast ftl. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 13–22. Springer.

Park, Y. and Kim, J.-S. (2011). zftl: Power-efficient data compression support for nand flash-based consumer electronics devices. *IEEE Transactions on Consumer Electronics*, 57(3):1148–1156.

Rosenblum, M. and Ousterhout, J. K. (1992). The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52.

Samsung (2014). Samsung V-NAND technology.

SiliconMotion (2019). Ufs and emmc controllers.

Smith, K. (2012). Understanding ssd over-provisioning. *Flash Memory Summit*.

Standard, J. (2018). Univeral flash storage (ufs) card extension starndard.

Takeuchi, K. (2009). Novel co-design of nand flash memory and nand flash controller circuits for sub-30 nm low-power high-speed solid-state drives (ssd). *IEEE Journal of Solid-State Circuits*, 44(4):1227–1234.

Wei, B., Cheok, S. T., Chng, Y., and Toh, C. (2015). Logical block address mapping. Patent No. 9,146,683, Filed Apr. 20th., 2011, Issued Sep. 29th., 2015.

Whitaker, K. (2015a). A comparative study of flash storage technologies for embedded devices.

Whitaker, K. (2015b). Embedded multimediacard (eMMC) eMMC/card product standard, high capacity, including reliable write, boot, and sleep modes.

Zhang, C., Wang, Y., Wang, T., Chen, R., Liu, D., and Shao, Z. (2014). Deterministic crash recovery for nand flash based storage systems. In *Proceedings of the 51st Annual Design Automation Conference*, DAC'14, pages 1–6, New York, NY, USA. ACM.

Zhou, D., Pan, W., Wang, W., and Xie, T. (2015). I/o characteristics of smartphone applications and their implications for eMMC design. In *2015 IEEE International Symposium on Workload Characterization (IISWC)*, pages 12–21. IEEE.