Utilizing Subpage Programming to Prolong the Lifetime of Embedded NAND Flash-Based Storage

Jung-Hoon Kim, Sang-Hoon Kim, and Jin-Soo Kim, *Member, IEEE*

***Abstract*—The flash translation layer (FTL) plays an important role in achieving high performance and reliability for nand flash-based storage devices. To ensure metadata consistency for storage requests, FTL is forced to save its own metadata to nand flash memory. However, a large number of FTL metadata writes can impair the lifetime of nand flash-based storage devices. Even though a small fraction of metadata is updated in the internal memory, FTL should write the whole flash page with the redundant data. In this paper, we utilize the subpage programming (SP) technique to decrease the amount of FTL metadata written to the nand flash memory. We also propose a novel FTL called subpage FTL based on the SP technique to prolong the lifetime of embedded nand flash-based storage. The evaluation results show that SPFTL reduces the written amount of FTL metadata by up to 45.4%, and therefore enhances the lifetime of a storage device by up to 19.3% in real workloads.**

***Index Terms*—Flash memory, flash translation layer (FTL), mobile devices, reliability, subpage programming (SP).**

I. Introduction

DURING the last decade, nand flash memory became the most popular and important medium for modern storage in many consumer electronics devices. Unlike traditional storage media, nand flash memory is read and written (or *programmed*) by *pages*, and written pages cannot be overwritten until they are erased. Due to the idiosyncrasy of nand flash memory, nand flash-based storage devices require a mechanism to deal with these characteristics so that hosts can access them over the traditional block I/O interface. Such a mechanism is generally referred to as the flash translation layer (FTL). Many FTL designs have been proposed to efficiently manage the address space on nand flash memory [1] and to maximize performance and/or lifetime [2]–[4].

To efficiently and reliably store data on the nand flash memory, FTLs must not only manage the mapping information for the address space but also perform several maintenance tasks, such as garbage collection (GC) and wear-leveling. These requirements force FTLs to maintain their own FTL metadata. For instance, FTLs are required to maintain the mapping information between the logical block addresses (LBAs) and the physical flash addresses. FTLs should also keep track of the erase counts for all blocks to level wear among them. In general, it is impractical to keep the entire metadata in the internal memory of storage devices because these devices are usually equipped with only a small amount of SRAM and the metadata should be persistent across power cycles. Thus, recent FTLs [5], [6] usually cache a part of the metadata in the internal memory, whereas the original metadata is stored in the nonvolatile flash memory.

Because the reliability and consistency of FTL metadata are critical, FTLs should periodically flush or write back the FTL metadata to nand flash memory. This is especially true for the most popular flash-based storage devices for mobile consumer devices such as embedded multimedia card (eMMC) and universal flash storage (UFS) [7], [8], which have a very limited amount of internal memory due to resource constraints. Such additional FTL metadata writes significantly increase the number of writes to the nand flash memory in embedded storage devices. In the analysis with real I/O traces, we observe a large fraction of flash writes is due to FTL metadata writes. We also identify that FTL metadata updates are so small that most of these FTL metadata writes have identical contents to the previous writes except for a few bytes. However, FTLs cannot write less than an entire page for FTL metadata updates, even though the update changes only a few bytes of FTL metadata and the rest of the page is unchanged. This wastes the limited lifespan of nand flash-based storage devices by repeatedly writing almost identical FTL metadata.

This paper presents a novel subpage FTL (SPFTL) to reduce the *effective* amount of FTL metadata writes thereby extending the lifetime of nand flash-based storage devices. To exploit the small size of metadata updates, SPFTL adopts the subpage programming (SP) scheme [9] presented in our previous work. SP allows to program only a part of the page, while the rest of the page is left unprogrammed. This approach prevents the unprogrammed part from wearing out, thereby extending the overall lifetime of nand flash memory. By reorganizing the structures of FTL metadata on nand flash memory, SPFTL handles small metadata writes using the SP scheme. Our evaluation results using real smartphone usage traces indicate that SPFTL reduces the amount of metadata writes by up to 45.4% and increases the lifetime of nand flash-based storage devices by up to 19.3%, compared to a traditional page-mapping FTL.

The rest of this paper is organized as follows. Section II introduces the fundamentals of FTL and its own metadata for

Manuscript received December 28, 2017; revised February 10, 2018; accepted February 15, 2018. Date of publication March 7, 2018; date of current version March 29, 2018. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. NRF-2016R1A2A1A05005494). *(Corresponding author: Jin-Soo Kim.)*

J.-H. Kim and J.-S. Kim are with the College of Information and Communication Engineering, Sungkyunkwan University, Suwon 16419, South Korea (e-mail: jhdev.kim@samsung.com; jinsookim@skku.edu).

S.-H. Kim is with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24061 USA (e-mail: sanghoon@vt.edu).

Digital Object Identifier 10.1109/TCE.2018.2811638

typical mapping schemes, followed by our analysis on the amount of FTL metadata writes with real I/O traces. The SP scheme used to reduce the amount of data writes is explained in Section III. Section IV describes the implementation of the baseline FTL and SPFTL designs with SP. We compare the amount of FTL metadata writes and show the device lifetime enhancement by the SPFTL in Section V. Section VI describes related work, and Section VII concludes this paper.

II. Background and Motivation

A. Flash Translation Layer

The primary goal of FTL is to hide the idiosyncrasy of nand flash memory, allowing hosts to access storage devices made up of nand flash memory via the traditional block I/O interface. To this end, FTL maps a LBA to a physical page number (PPN). When a host requests for a write to an LBA, FTL divides the request into a logical page number (LPN) by its management unit and stores requested data in the flash memory. FTL maintains the mapping information between the LPN and the corresponding PPN. When the host later reads the storage device using the same LBA, FTL translates its LPN to corresponding PPN in flash chips, and serves the request from the PPN location. Because nand flash memory cannot be overwritten, FTL deals with overwrites via out-of-place updates. When the host requests for a write to an already-written LBA, FTL writes the requested data to a free page, marks the previously stored data invalid, and updates the LPN mapping for the LBA to the new PPN location.

The invalidated data is reclaimed by a procedure called GC. When the number of free blocks in the storage falls below a threshold, FTL initiates GC to secure enough free blocks to allow FTL to keep handling write requests from the host. To perform GC, FTL selects a victim block, migrates valid data in the victim block to other blocks, and then erases the victim block, which eventually converts the victim block to a free block. The free block can be used again by FTL to handle subsequent write requests. Typically, the victim block is selected by considering the number of pages containing valid data (called valid pages) to minimize the overhead of migrating them [10], [11].

B. FTL Metadata

FTLs can be classified into three categories according to their mapping granularity: 1) block-mapping FTL; 2) page-mapping FTL; and 3) hybrid-mapping FTL. We can consider the mapping information as one of FTL metadata because it describes the property (i.e., the PPN location) of the corresponding data stored in the storage device. In addition to the mapping information, FTL should maintain other types of metadata to provide various features to improve the performance and reliability of the storage device.

For example, FTL should distinguish free blocks from used blocks and keep track of how many free blocks are available in the storage. Usually, FTL manages these metadata using a bitmap and a counter. To perform GC, FTL should maintain the validity of pages to identify valid pages that need to be migrated to reclaim the block. To optimize the victim block selection for GC, FTL may trace the number of valid pages for each block, the age of erase blocks, and the hotness for each page [12].

FTL also should level the wear of blocks to maximize the lifetime of the flash-based storage device [13]. FTL performs so-called wear-leveling by switching a block erased many times with a block whose erase count is small. To perform wear-leveling, FTL should maintain a counter for each block which tells the number of erases the block has experienced. In addition, FTL may adopt various sophisticated features such as hot–cold data separation [11], wear-leveling index [4], wear-unleveling [15], and deduplication [14] to enhance the performance and reliability of the storage device. These schemes also require their own FTL metadata to keep track of the associated information during the operation of FTL.

Most of the aforementioned FTL metadata should be persistent and consistent across a power cycle. For this reason, they must be stored in the nonvolatile flash memory. The exact location of the metadata can vary according to the FTL designs. One approach is to divide the PPN address space into partitions, and reserve some partitions for storing metadata [16], [17]. This design simplifies block management while manipulating FTL metadata. However, the static partitioning can lead to storage underutilization when the amount of metadata is not static. The other approach is to store metadata along with regular data without reserving blocks for metadata [2]. This approach can resolve the storage underutilization issue; however, it complicates block management.

C. Motivation

The types and amount of FTL metadata depend on the design and implementation of its mapping scheme and the features the FTL supports. Nevertheless, FTL metadata stored in nand flash memory should be loaded into the internal memory of the storage device to be accessed or updated. However, the amount of internal memory is so small and limited that only a part of the FTL metadata can be loaded at a time. For instance, the sizes of internal memory for typical eMMC and UFS storage systems, which are currently popular in the commercial consumer market, are about several hundred kilobytes. When we consider an FTL that manages 32 GiB of flash memory in 4-KiB page granularity, the FTL requires at least 32 MiB for mapping information, which exceeds the size of the internal memory of such storage devices. Thus, many FTLs for embedded storage devices utilize the internal memory as a cache of FTL metadata, which is analogous to the CPU cache and the main memory in the memory hierarchy.

After being loaded into the internal memory, FTL metadata can be modified by write requests, GC, wear-leveling, and so on. The updated FTL metadata should be written back to the nand flash memory to make the update persistent across a power cycle. The FTL metadata updates occur frequently, as the internal memory is small compared to the entire FTL metadata and the FTL metadata reliability is critical for the reliability of the storage device. To mitigate the frequent metadata writes, one might utilize journaling and

recovery techniques [18]. These techniques attempt to skip writing noncritical metadata updates. Instead, they scan the metadata during the power-on sequence of the device to identify and recover possible metadata corruptions. However, this approach can be inappropriate for embedded storage devices, which need to guarantee a fast system boot time. Thus, embedded storage devices are apt to write back or flush FTL metadata to nand flash memory considering the time spent on the power-on sequence. For instance, eMMC storage systems flush dirty metadata periodically for the fast boot time.

The frequent metadata flush results in many metadata writes to nand flash memory. To better understand the characteristics of FTL metadata updates and the associated flash writes, we conducted various trace-driven studies. We collected I/O traces from actual smartphones used by customers, and replayed the traces on an FTL simulator that implements the page-mapping FTL designed for embedded storage devices. Please refer to Section IV for the FTL design and Section V for details about the I/O traces and the FTL simulator. Fig. 1 summarizes interesting results obtained from this paper.

We segregate the FTL metadata writes from total flash writes while replaying the various I/O traces. In the clean condition where no host write requests have been issued before, the FTL metadata updates account for 8.3%, 6.7%, and 11.0% of total flash writes for patterns A, B, and C, respectively. However, the percentage of FTL metadata updates are significantly increased to 88.8%, 87.4%, and 86.5% of total flash writes for patterns A, B, and C, respectively, in the aging condition after numerous host write requests. Fig. 1 shows the breakdown of the amount of flash writes. These write amounts for the FTL metadata updates are a significant fraction of the total flash writes. In addition to the simulator study, we also performed a similar evaluation inside commercial eMMC storage systems, and observed a similar result; even in the clean condition, approximately 24.8%–60.9% of the flash writes were originated from FTL metadata writes. Commercial embedded storage systems have large metadata overhead due to many features such as fast boot time and sudden power-off recovery on very restricted resources. This result of flash writes implies that a considerable fraction of nand flash wear-out comes from FTL metadata writes, and there is an optimization opportunity to extend the lifetime of nand flash-based storage further if these FTL metadata updates are managed properly.

Interestingly, we found the unique characteristics for FTL metadata writes. For each write for FTL metadata update, we identified the changes in pages by comparing the updated FTL metadata with the original FTL metadata stored in the corresponding page in the flash memory. We noticed that FTL metadata updates are made to only a few bytes in the metadata pages. In particular, the average amounts of changes are only 70.3, 48.4, and 66.0 bytes for patterns A, B, and C, respectively. From 86.5% to 88.8% of FTL metadata writes change less than 20 bytes, and the largest amount of changes is only 1 KiB out of a 4-KiB page. The small FTL metadata update is because FTL metadata updates for write requests or other operations usually touch only a few pages, altering a few bytes of the corresponding mapping information and other metadata. This means that FTL metadata updates are programming flash pages with the almost identical values to the original pages, except for the small updated part. From this observation, we are motivated to eliminate unnecessarily programmed data to extend the lifetime of embedded storage devices.

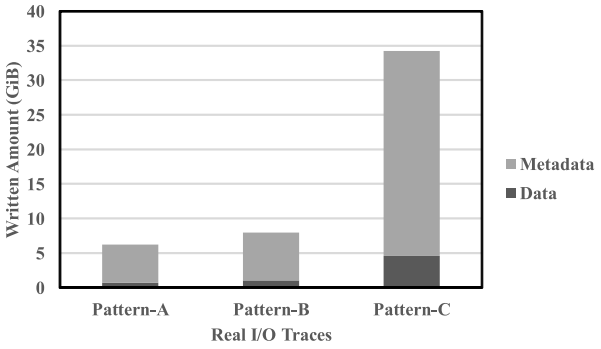

Fig. 1. Breakdowns of flash writes on the real I/O traces.

III. Subpage Programming

As we discussed in the previous section, metadata updates are small but comprise a large fraction of the total flash writes. This section introduces SP, which is our previous work to extend the lifetime of nand flash memory.

A. Subpage

We define a *subpage* as a programming unit that is smaller than the page size. Typically, we set the subpage size to a multiple of the sector size (e.g., 512 bytes). For instance, we can consider two 2-KiB subpages or four 1-KiB subpages for a 4-KiB page. When programming a subpage with data, the other subpages in the same page are filled with the predefined value, typically 1, that corresponds to the unprogrammed cell. Thus, the cells outside of the subpage being programmed are actually left unprogrammed, incurring less stress to the oxide layer in the cells. Thus, overall, SP inflicts less stress on cells than regular full-page programming.

The key idea behind the SP scheme is to program the subpage only when we need to write a small amount of data rather than to program the entire page padded with the original data. Let us consider a case when an FTL metadata update changes two 2-KiB ranges in two 4-KiB pages. It requires two full-page programming operations to write 8 KiB in the original case, whereas the update can be handled by two SP operations that write only 4 KiB. In this sense, we consider adopting the SP scheme in FTL designs to handle the small FTL metadata writes.

SP is similar to the partial-page programming [19] in that the pages are partially programmed. The main differences between these programming schemes come from the way in which the unprogrammed cells are handled. In case of partial-page programming, the remaining part of a page is programmed soon after by successive partial programming operations. Contrarily, in SP, the remaining subpages are not programmed by subsequent SP unless the block containing

the page is erased. Consequently, partial-page programming incurs more intrapage program disturbance, resulting in less endurance compared to SP. In our previous evaluation with real nand chips from three manufacturers, the endurance cycles of partial-page programming obtained from SLC-mode blocks were less than those of full-page programming by up to 24% [9]. In case of MLC-mode blocks, the partial-page programming corrupted the corresponding LSB/MSB page data in all nand chips.

B. Subpage Programming Mode

We first define a *subpage partition* as a set of subpages in a block which have the same offset within the pages in the block. For example, if the subpage size is half of the page size, each block has two subpage partitions. The subpage partition 0 consists of subpages that are on the lower part of pages whereas the subpage partition 1 on the upper part of pages. In the same manner, we can define four partitions when the subpage size is a quarter of the page. The number of subpages in a partition is the same as that of pages in a block.

We call a block an *SP block* when the block is dedicated for SP. We impose three restrictions in programming SP blocks. The first is that full-page programming is prohibited on SP blocks. The second is that only the subpages belonging to the same partition can be used at a time. When a partition is used for an SP block, the other partitions can be used only after the block is erased. The last restriction is that each subpage should be sequentially programmed from the lowest page number to the highest page number, as done in full-page programming.

Because there are two or more subpage partitions in SP blocks, we can consider a number of orders in using the subpage partitions. Referring to the best result of endurance cycle, we consider the pong-pong order [9] that all flash cells in a page get the minimal program disturbance stress. Fig. 2 shows an example use of the pong-pong order, where a partition is continuously used until an uncorrectable bit error occurs in the partition. Then, the next partition is used until another uncorrectable bit error occurs again in the partition. In the pong-pong order, the flash cells in unused partitions can stay in the program inhibited state until they are actually used.

IV. SPFTL

This section discusses how an FTL can utilize the SP scheme to alleviate the problem caused by excessive FTL metadata writes. We first present the baseline FTL design, and provide two different approaches to the FTL. We believe other FTLs can adopt the SP technique to their metadata management in a similar way.

A. Baseline FTL

The baseline FTL, which we refer to as DFTL_{por}, is based on DFTL [2], which utilizes the on-demand mapping technique [5], [6]. DFTL is one of the state-of-the-art FTLs which dynamically loads necessary metadata into the internal memory to maximize the random write performance. In order to clarify our idea and contributions, we only add the minimal

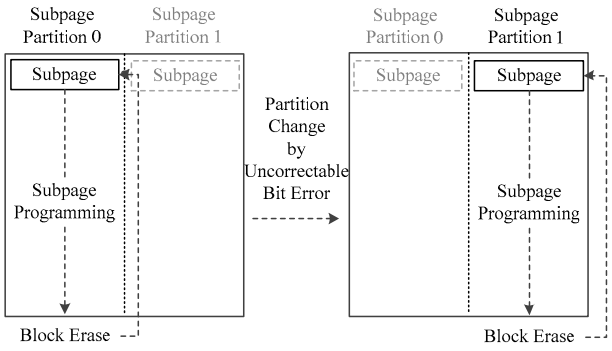

Fig. 2. Pong-pong programming order for SP.

features to DFTL: per-block metadata management and metadata recovery scheme for a power cycle. We anticipate that other sophisticated features that require more metadata, such as deduplication, hot/cold data separation, and so on, can be considered as well without any significant modification.

As depicted in Fig. 3, DFTL_{por} maintains the following metadata in the internal memory: cached page mapping table (CPMT), cached block information table (CBIT), and global context (GXT). Similar to DFTL's translation blocks [2], the entire page mapping table (PMT) is divided by the flash page size and stored in the dedicated flash blocks called *PMT blocks*. The active page mapping entries are cached in a region of the internal memory called CPMT. For per-block metadata management, DFTL_{por} maintains a metadata structure called the block information table (BIT). Each entry in the BIT keeps track of the number of valid pages and the erase count for a physical flash block. Because the size of the BIT is huge, it is also divided by the flash page size and stored in the flash blocks called *BIT blocks*. The BIT entries needed for victim block selection and wear leveling are cached in an area called CBIT of the internal memory. To track the locations of the most up-to-date PMT and BIT pages, DFTL_{por} maintains the page mapping directory (PMD) and the block information directory (BID) in the internal memory, respectively. The two directories, PMD and BID, constitute the GXT, which is the top-level metadata managed by DFTL_{por}. Other than PMD and BID, the GXT also has other metadata such as block summary information (BSI) and update/free-block pointers (UFP), which are used in GC for processing the victim block. Because the GXT is frequently updated and its space overhead is very low (e.g., 2047 bytes for a 16-GB device), DFTL_{por} keeps the full GXT metadata in the internal memory all the time.

DFTL_{por} writes the data into the *update block* which denotes the flash block where the incoming data are written. This updates the mapping entry in the CPMT and also changes the corresponding information in the CBIT. Note that there can be no space for the new entries in the CPMT or CBIT when we update them. In this case, one or more dirty entries can be flushed to the flash memory. Finally, DFTL_{por} updates the associated entry in the PMD and BID. When there is no space in the current update block, the GC process is invoked. As the update block is changed after GC, DFTL_{por} first flushes all dirty entries in CPMT/CBIT belonging to the current update

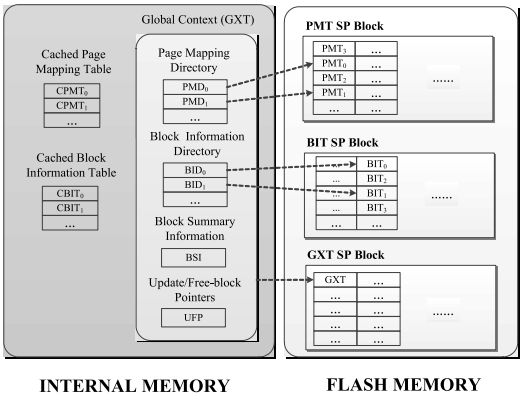

Fig. 3. DFTL_{por} and SPFTL metadata architecture.

block to the flash memory. The victim selection is based on the greedy policy which cleans the block with the least number of valid pages by referring to the BSI in the GXT. After choosing the victim, DFTL_{por} identifies all the valid pages in the victim block and migrates them to a free block. Afterwards, the free block becomes the new update block, while the victim block is erased and then converted to a free block. Finally, the GXT is updated and then flushed to the flash memory.

Similar to the update block for user blocks, metadata blocks are managed in a log-structured manner. Each type of metadata block has its own update block and the new metadata is written sequentially in the corresponding update block. If all the free blocks are exhausted in the metadata partition, a victim metadata block is chosen within the partition based on the greedy policy as done in user blocks. The information on the current update block in each partition is tracked by the UFP in the GXT. The UFP also manages free blocks in the user partitions and the metadata partitions.

To safely recover any metadata on a sudden power failure, the contents of the GXT should be flushed to the flash memory periodically. DFTL_{por} tries to minimize the amount of metadata writes by flushing the GXT to the dedicated GXT block only when one of the PMT, BIT, or user update blocks is changed. This is sufficient to guarantee the metadata consistency on a sudden power loss. Once we find the latest GXT information from the GXT block, the metadata can be made up-to-date by scanning pages in the PMT, BIT, and user update blocks which are written after the GXT flush.

B. SPFTL

We first present the basic design of SPFTL, which is a clean-slate approach that fully utilizes SP for metadata management. SPFTL stores all metadata in SP blocks, and metadata updates are directly written to their corresponding update blocks.

In SPFTL, PMT or BIT pages are written to the flash memory when they are evicted from the corresponding cache (CPMT or CBIT) due to the limited internal memory size. In this case, the new PMT (or BIT) page differs from the old one only by a few dirty entries. The rest of the metadata is written to the flash memory *redundantly* simply because the entire flash page should be programmed at once. However, the SP technique unlocks a new opportunity to write the data smaller than the flash page size. If we manage metadata pages (PMT, BIT, and GXT pages) in the subpage unit, we can reduce the *effective* amount of metadata writes by avoiding duplicated writes to the unmodified part of the metadata page.

The basic metadata architecture of SPFTL is same as that of DFTL_{por} as shown in Fig. 3. The main difference is that SPFTL reduces the metadata size to fit into subpages, and stores all the FTL metadata in SP blocks. For brevity, we only consider the case where the subpage size is half of the flash page size. Because the PMT and BIT pages are now stored in the subpage granularity, SPFTL reads or writes them by subpages. However, this does not require any change in managing CPMT and CBIT. When one or more dirty entries are evicted from CPMT or CBIT, SPFTL loads the corresponding subpage into the internal memory, updates the modified entries, and writes it back to the corresponding metadata update block.

Compared to DFTL_{por}, one downside of SPFTL is that the number of entries in PMD and BID is doubled due to the increased number of PMT and BIT pages. If the GXT size is increased beyond the subpage size due to the increase in PMD and BID, the benefits of managing PMT and BIT pages using the SP technique will be offset. However, this does not have a significant effect on the overall performance for the following reasons. First, our experimental results using the real workloads show that the ratio of the GXT writes to the total metadata writes is only 3.0%. Therefore, its impact will be negligible even though each GXT write requires two SP operations. Second, because the number and the location of PMT and BIT blocks are fixed, it is possible to reduce the size of GXT by storing only the block index instead of the block number in PMD or BID entry.

C. SPFTL With Metadata Logging

We present another variant of SPFTL called $\text{SPFTL}_{\text{log}}$, which employs the metadata logging approach. The design principle behind $\text{SPFTL}_{\text{log}}$ is to utilize SP in managing FTL metadata, while minimizing the efforts to modify DFTL_{por}'s metadata architecture. In this sense, $\text{SPFTL}_{\text{log}}$ is identical to DFTL_{por} except for handling metadata updates.

To handle small metadata updates with SP, $\text{SPFTL}_{\text{log}}$ simply logs only the modified metadata entries in a dedicated logging block as shown in Fig. 4. Each metadata type has its own logging block. Thus, there are three kinds of logging blocks in $\text{SPFTL}_{\text{log}}$, each for GXT, PMT, and BIT. Those logging blocks are set to the SP blocks so that they can be programmed only via SP.

When a metadata update occurs, $\text{SPFTL}_{\text{log}}$ creates a log of the update and writes it to the corresponding logging block via SP. For a PMT update, the log is comprised of an LBA and its updated PPN. The log for a BIT update is comprised of the block number, valid page count, and erase count of the updated block. Similarly, the log for a GXT update is composed of the type of updated metadata and its updated value.

Logs are coalesced into a subpage if multiple updates are performed to the same original metadata page. For instance, logs belonging to the same PMT page are coalesced in a subpage. The logs are merged into the original metadata when

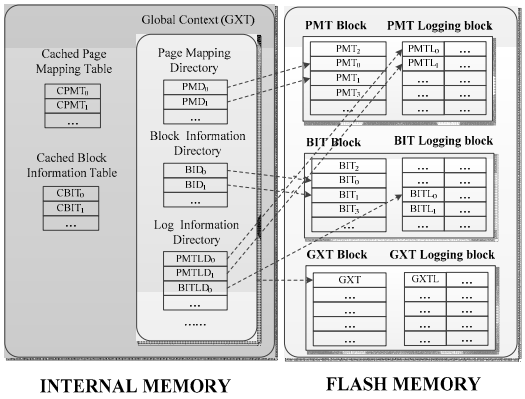

Fig. 4. $SPFTL_{log}$ metadata architecture with logging blocks.

the subpage of the corresponding logs becomes full. To merge logs, $SPFTL_{log}$ loads the corresponding metadata page into the internal memory, updates the metadata in the logs, and writes back the updated metadata to the update block according to the metadata type.

$SPFTL_{log}$ keeps tracks of the up-to-date logs using a data structure called log information directory (LID). Each entry in LID points to the location of the update in logging blocks for the corresponding metadata page. To access a metadata page, $SPFTL_{log}$ first looks up the LID to see whether the corresponding metadata page is updated and logged in the logging block. If there is an entry for the metadata page, $SPFTL_{log}$ reads the up-to-date metadata from the logs in the logging block pointed to by the LID entry. Otherwise, the metadata page does not have any associated logs, and $SPFTL_{log}$ accesses the metadata page stored in each update block.

V. Evaluation

A. Evaluation Setup

We have implemented a trace-driven FTL simulator for $DFTL_{por}$, SPFTL, and $SPFTL_{log}$. The FTL simulator accepts a block-level trace and processes each request in the trace according to the FTL implementation. We used the representative parameters of 20-nm-class MLC nand flash memory for simulating nand flash memory by referring to hardware specifications from major flash manufacturers. Specifically, the nand flash memory is comprised of 8-KiB pages which is the logical to physical mapping granularity in our evaluation, and each block is comprised of 128 pages. Thus, the size of a single block is 1 MiB. Table I summarizes the characteristic of 20-nm-class MLC nand flash memory. We set up the total 16 384 blocks which yield 16-GiB capacity as the total storage space. Note that 15 564 blocks are used for the user space, which is equivalent to a 15.20-GiB capacity and the remaining 5% of the blocks are reserved as an overprovisioning space [25].

We assume 26 KiB of internal memory to operate all three FTLs by referring to the common hardware specifications for eMMC and UFS storage devices. Among the internal memory, we set 2 KiB to cache PMT and BIT entries for one update block and allocate 4 KiB for GXT to manage two directories and other remaining metadata. The internal memory allows FTLs to cache the entire GXT, whereas only a part of PMT and BIT can be loaded into the given memory. We assume the remaining internal memory is used for an 8-KiB page copy-back and an 8-KiB buffer to manage PMT (or BIT) page, which are required to operate each FTL. An additional 4-KiB buffer is used in $SPFTL_{log}$ for merging logs.

We assume that metadata blocks are operated in the SLC-mode [20], [21] whereas data blocks are in the MLC-mode. Each block in MLC or TLC chip can be set to the SLC-mode, in which only LSB pages are used. Because FTL metadata occupies a small portion of storage space but has critical information, we utilize SLC-mode which provides higher reliability and a longer lifetime than MLC-mode. However, we should consider the different endurance cycles between metadata blocks and data blocks. We believe that the proposed scheme can be incorporated in any MLC or TLC chips without significant modifications.

We use both real I/O traces and synthetic random write patterns in our evaluation. The real I/O traces are collected from customers by setting up smartphones with a block-level tracing tool and letting consumers use smartphones in the usual way for 24 h. Table II summarizes the characteristics of three real I/O traces. We also perform various random writes with 8-KiB page requests to measure the number of FTL metadata writes. Because FTLs operate with limited internal memory, the random writes can generate a large amount of FTL metadata writes.

For the performance metric, all nand operations are gathered while every request is served (i.e., metadata write counts and the corresponding erase counts). Consequently, the operation time for a given trace is measured to compare FTL performance in the Flashsim simulator [22]. The operation time is based on the time parameters of 20-nm-class MLC nand memory shown in Table I.

TABLE I
Characteristic of 20-nm-Class MLC nand Flash Memory

Structure	Page Size	8 KiB
	Block Size	1 MiB (128 pages)
Access time	Page Read	80 µs
	Page Program	1.6 ms
	Block Erase	1.5 ms
	Byte Transfer	25 ns

TABLE II
Features of Real Workloads From Smartphones

I/O Traces	Write Amount (MiB)	% of 4KiB Requests	% of Sequential Requests	Description
Pattern A	710.5	66.8	23	Small Random
Pattern B	1,057.9	56.2	39	Large Sequential
Pattern C	4,749.5	57.4	28	Large Amount

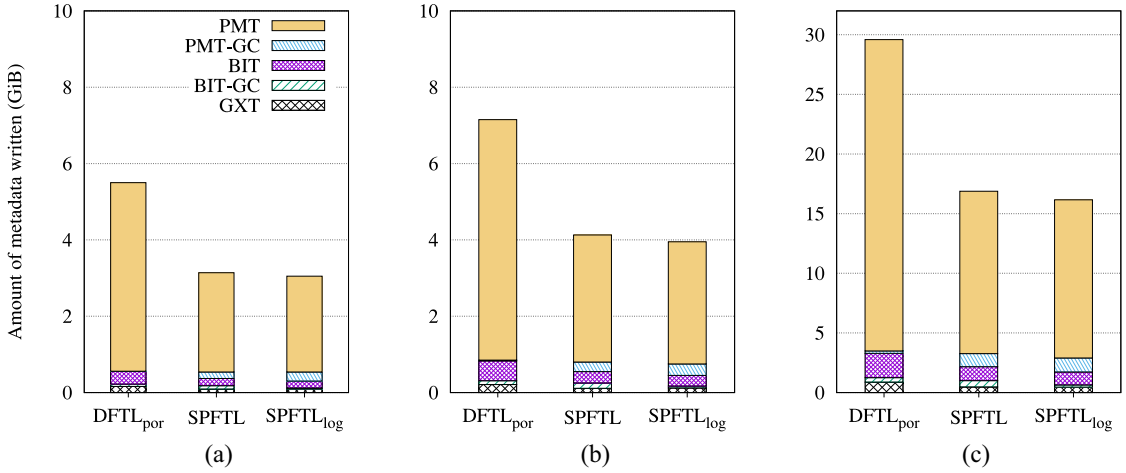

Fig. 5. Amount of FTL metadata writes in the real I/O traces. (a) Pattern-A. (b) Pattern-B. (c) Pattern-C.

B. Analysis of FTL Metadata

As we expected, a large amount of FTL metadata is written when FTLs perform 15.20 GiB of random writes with 8-KiB page requests. This is because random write patterns result in many changes of PMT/BIT entries in different PMDs. For DFTL$_{por}$, the total amount of FTL metadata written reaches 15.16 GiB, which is almost the same as the total amount of user writes.

We replayed three real I/O traces on a trace-driven FTL simulator and collected performance metrics on FTL metadata. First, we perform three real I/O traces on the clean condition in which all data and metadata blocks are empty. The amount of metadata written for DFTL$_{por}$ ranges from 6.7% to 11.0% of the total flash writes as mentioned in Section II. SPFTL decreases them to 6.3%, 3.9%, and 4.9% of the total flash writes for three real I/O traces, respectively. SPFTL$_{log}$ also decreases the written metadata amounts to 6.2%, 3.8%, and 4.7%.

Fig. 5 presents the amount of metadata writes incurred while running the real I/O traces in the aging condition. To make the aging condition, we fill up the entire user space with the random workload and replay those I/O traces, which incurs GC and activates metadata writes during the process of the given I/O traces.

When running pattern A on DFTL$_{por}$, we can see that the significant amount of FTL metadata traffic comes from PMT writes (89.8%), while the writes for the BIT and the GXT are responsible for 7.3% and 2.9%, respectively. This is due to the small and random write pattern, which touches PMT/BIT pages for each write request. We observe that the total amount of FTL metadata writes accounts for 5.50 GiB in pattern A, which is 7.93 times of the data written by the user. Similar results are observed for patterns B and C. We can observe that the total amount of FTL metadata writes in pattern C is notably higher than that in patterns A and B. This suggests that the amount of FTL metadata writes is proportional to the amount of data written by the user.

We can observe that two SPFTLs significantly decrease the amount of metadata writes for all the given traces. SPFTL reduces the amount by 42.9%, 42.2%, and 43.0% for patterns A, B, and C, respectively, compared to DFTL$_{por}$. Specifically, SPFTL$_{log}$ reduces the amount by 44.8%, 45.0%, and 45.4%. This shows that the logging operations of SPFTL$_{log}$ can further decrease the metadata amount occurred due to the spatial locality of real workloads. In Fig. 5, we also break down PMT/BIT to PMT-GC/BIT-GC which represents the amount of valid FTL metadata pages migrated during FTL metadata GC. PMT-GC and BIT-GC of SPFTL and SPFTL$_{log}$ are slightly increased because the increased number of FTL metadata entries due to the use of subpages, but their amounts are negligible compared to the total FTL metadata writes. Consequently, the amount of FTL metadata writes in all three real I/O traces is reduced by up to 45.4% compared to that of DFTL$_{por}$.

C. Lifetime Analysis

The use of SP generally increases the erase count of the SP block. However, our previous research shows that the SP block can endure a greater number of erase counts compared to other conventional blocks with full-page programming [9]. To calculate the accurate lifetime of SPFTL designs, we should reflect the increased endurance of the SP block in the total erase count of SPFTL.

Because SPFTL and SPFTL$_{log}$ use both full and SP, we define the normalized wear index (NWI) to take into account the erase count of SP blocks, which can be calculated as

$$\text{NWI} = \sum_{b=1}^{m} \text{Erase}_{\text{fp}}(b) + \sum_{b=1}^{n} \text{Erase}_{\text{sp}}(b) \times \text{Weight}_{\text{sp}} \quad (1)$$

where m and n indicate the number of full-page blocks and SP blocks, and Erase$_{\text{fp}}$ and Erase$_{\text{sp}}$ mean the erase counts of them, respectively. Weight$_{\text{sp}}$ is a normalization factor for the erase counts of SP blocks. In our previous result, the endurance cycle of SP blocks is improved by 1.71 times compared to the blocks with full-page programming, which was measured on the SP2O$_{\text{SLC}}$ configuration that was composed of 4-KiB subpages [9]. Therefore, we set Weight$_{\text{sp}}$ to 1/1.71 in order to calculate NWI of SPFTL and SPFTL$_{log}$. Note that DFTL$_{por}$ does not use any SP blocks, while SPFTL use SP blocks for all metadata blocks. In SPFTL$_{log}$, only the logging blocks are

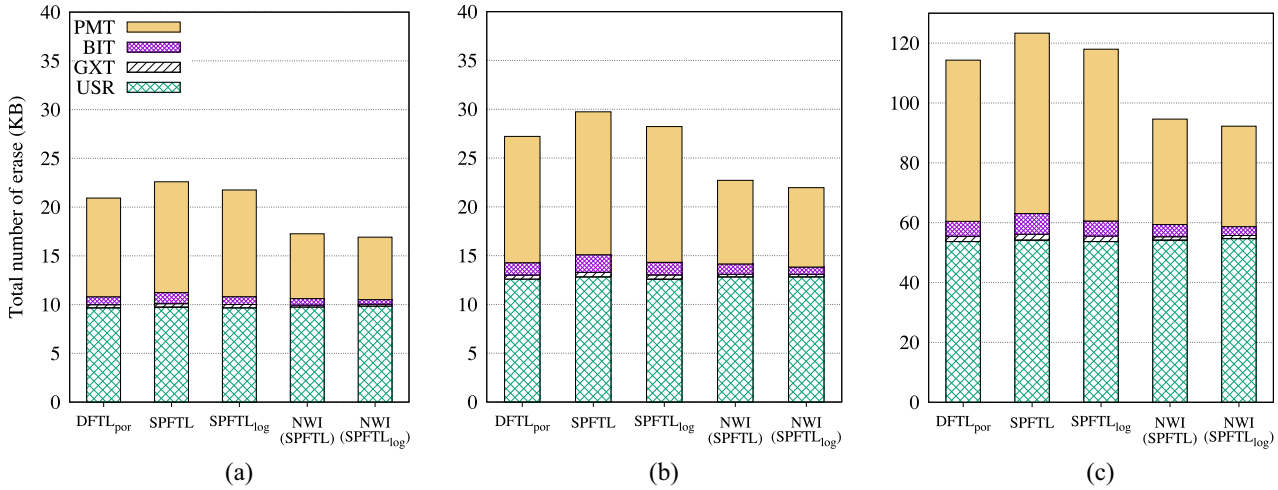

Fig. 6. Comparing the total erase count of DFTL$_{por}$, SPFTL, and SPFTL$_{log}$. (a) Pattern-A. (b) Pattern-B. (c) Pattern-C.

operated in SP blocks and the rest of the metadata blocks are operated same as in DFTL$_{por}$.

Fig. 6 depicts the total erase counts of DFTL$_{por}$, SPFTL, and SPFTL$_{log}$ in three real I/O traces. NWI(SPFTL) and NWI(SPFTL$_{log}$) represent the normalized total erase count according to (1). Due to the use of SP, SPFTL, and SPFTL$_{log}$ show the slightly increased erase count. However, the normalized total erase counts are estimated to be less than others and the device lifetime can be significantly increased for all the given traces. As a result, SPFTL increases the lifetime by up to 17.5% in pattern A, 16.5% in pattern B, and 17.2% in pattern C. SPFTL$_{log}$ also gains more lifetime than DFTL$_{por}$ by 19.2% in pattern A, 19.3% in pattern B, and 19.3% in pattern C.

D. Performance Analysis

We utilize the *FTL service time* to estimate the performance of SPFTL and SPFTL$_{log}$. Because the only difference is whether FTL metadata is managed with SP or not, we evaluate the performance by using the nand timing parameters shown in Table I. Compared to DFTL$_{por}$, SPFTL$_{log}$ increases the service time by 4.7% in pattern A, 4.4% in pattern B, and 4.1% in pattern C. This is because the overhead of looking up the LID for the updated log is added in the critical path. However, SPFTL decreases the service time by 4.3%, 3.2%, and 4.2% in patterns A, B, and C, respectively. This improvement depends on the amount of data written by the SP scheme. We also investigate the response time taken by nand operations per request. SPFTL$_{log}$ increases the average write latency by up to 3.1% in pattern A, 3.1% in pattern B, and 2.9% in pattern C. However, SPFTL reduces the average write latency by up to 5.3%, 0.9%, and 5.4% for patterns A, B, and C, respectively. The write latency of SPFTL is also decreased due to the transfer time of subpages which is faster than that of full pages.

VI. Related Work

FTL can gather small host write requests to the write buffer [26] which reduces read, modify, and write operations for the full-page programming. However, SPFTL immediately writes the subpage and therefore reduces the redundancy incurred by the small portion of metadata update. To increase the endurance cycle of nand flash memory, equalizer [4] considers both the erase count of the block and the variation of the page programming time. In case of wear-unleveling [15], every page endurance is inspected before a block wears out. Dynamic program and erase scaling [23] can change the erase voltage and the erase time of nand flash memory so that it improves the lifetime of nand flash-based storage.

Similar to other approaches, content-aware FTL increases the lifetime by removing the unnecessary and duplicated data. Object-based FTL [24] enables lazy persistency of index metadata in a host file-system and eliminates journals while maintaining consistency to extend the lifetime of nand flash-based storage.

Our approach proposed in this paper is orthogonal to the aforementioned approaches in that SPFTL can be combined with other schemes to enhance the lifetime of nand flash-based storage devices.

VII. Conclusion

Traditional FTLs always perform full-page programming, which is extremely redundant, to store the small changes of FTL metadata writes. In order to decrease the redundancy of FTL metadata writes, we proposed a novel SPFTL that utilizes the SP technique. We believe the proposed approaches can be adopted for other FTLs.

Our evaluation results with real I/O traces showed that SPFTL reduces the amount of metadata written by up to 45.4%. Finally, SPFTL improves the lifetime of embedded nand flash-based storage by up to 19.3% in the environment composed of 20-nm-class nand chips.

References

[1] R. Chen *et al.*, "On-demand block-level address mapping in large-scale NAND flash storage systems," *IEEE Trans. Comput.*, vol. 64, no. 6, pp. 1729–1741, Jun. 2015.

[2] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. ASPLOS*, Washington, DC, USA, Mar. 2009, pp. 229–240.
[3] F. Chen, T. Luo, and X. Zhang, "CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proc. 9th USENIX Conf. File Stor. Technol.*, San Jose, CA, USA, Feb. 2011, pp. 77–90.
[4] Y.-J. Woo and J.-S. Kim, "Diversifying wear index for MLC NAND flash memory to extend the lifetime of SSDs," in *Proc. EMSOFT*, Montreal, QC, Canada, 2013, pp. 1–10.
[5] H. Kim, S. Jung, and Y. H. Song, "Map cache management using dual granularity for mobile storage systems," *IEEE Trans. Consum. Electron.*, vol. 60, no. 4, pp. 644–652, Nov. 2014.
[6] K. Ha, T. Kim, B. Y. Ahn, and J. Kim, "Resource-aware sector translation layer for resource-sensitive NAND flash-based storage systems," *IEEE Trans. Consum. Electron.*, vol. 58, no. 2, pp. 462–469, May 2012.
[7] *JEDEC e.MMC Standard v5.1*, JEDEC Standard JESD84-B51, Feb. 2015.
[8] *JEDEC Universal Flash Storage (UFS)*, JEDEC Standard JESD223-1A, May 2016.
[9] J.-H. Kim, S.-H. Kim, and J.-S. Kim, "Subpage programming for extending the lifetime of NAND flash memory," in *Proc. DATE*, Grenoble, France, 2015, pp. 555–560.
[10] S. Ji and D. Shin, "An efficient garbage collection for flash memory-based virtual memory systems," *IEEE Trans. Consum. Electron.*, vol. 56, no. 4, pp. 2355–2363, Nov. 2010.
[11] J. Lee and J.-S. Kim, "An empirical study of hot/cold data separation policies in solid state drives (SSDs)," in *Proc. SYSTOR*, Haifa, Israel, 2013, pp. 1–6.
[12] S.-W. Lee *et al.*, "A log buffer-based flash translation layer using fully-associative sector," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, Jul. 2007, Art. no. 18.
[13] M. Murugan and D. H. C. Du, "Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead," in *Proc. IEEE 27th Symp. Mass Stor. Syst. Technol.*, Denver, CO, USA, May 2011, pp. 1–12.
[14] J.-Y. Ha, Y.-S. Lee, and J.-S. Kim, "Deduplication with block-level content-aware chunking for solid state drives (SSDs)," in *Proc. IEEE Int. Conf. Embedded Ubiquitous Comput. High Perform. Comput. Commun. (HPCC_EUC)*, Nov. 2013, pp. 1982–1989.
[15] X. Jimenez, D. Novo, and P. Ienne, "Wear unleveling: improving NAND flash lifetime by balancing page endurance," in *Proc. 12th USENIX Conf. File Stor. Technol.*, Santa Clara, CA, USA, Feb. 2014, pp. 47–59.
[16] Y.-H. Chang, P.-L. Wu, T.-W. Kuo, and S.-H. Hung, "An adaptive file-system-oriented FTL mechanism for flash-memory storage systems," *ACM Trans. Embedded Comput. Syst.*, vol. 11, no. 1, pp. 1–19, Apr. 2012.
[17] M. Huang, Z. Liu, L. Qiao, Y. Wang, and Z. Shao, "An endurance-aware metadata allocation strategy for MLC NAND flash memory storage systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 4, pp. 691–694, Apr. 2016.
[18] S.-H. Lim, "Implementation of metadata logging and power loss recovery for page-mapping FTL," *IEICE Electron. Exp.*, vol. 10, no. 11, pp. 1–6, 2013.
[19] Y. Li, Y. K. Fong, and T. Miwa, "Non-volatile memory and control with improved partial page program capability," U.S. Patent 7 057 939, Jun. 6, 2006.
[20] M. Lasser and K. Yair, "Flash memory management method that is resistant to data corruption by power loss," U.S. Patent 6 988 175, Jan. 17, 2006.
[21] H.-W. Tseng, L. M. Grupp, R. E. Spada, and S. Swanson, "Understanding the impact of power loss on flash memory," in *Proc. 48th Design Autom. Conf.*, Jun. 2011, pp. 35–40.
[22] Y. Kim, B. Tauras, A. Gupta, D. Nistor, and B. Urgaonkar, "Flashsim: A simulator for NAND flash-based solid-state drives," Dept. Comput. Sci. Eng., Pennsylvania State Univ., State College, PA, USA, Rep. CSE-09-008, 2009.
[23] J. Jeong, S. S. Hahn, S. Lee, and J. Kim, "Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling," in *Proc. 12th USENIX Conf. File Stor. Technol.*, Santa Clara, CA, USA, Feb. 2014, pp. 61–74.
[24] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. 11th USENIX Conf. File Stor. Technol.*, San Jose, CA, USA, Feb. 2013, pp. 257–270.
[25] P. Amato, D. Caraccio, E. Confalonieri, and M. Sforzin, "An analytical model of eMMC key performance indicators," in *Proc. IEEE Int. Memory Workshop (IMW)*, Monterey, CA, USA, May 2015, pp. 1–4.
[26] M. Kang, W. Lee, and S. Kim, "Subpage-based flash translation layer for solid state drivers," in *Proc. KAIST Open Access Self Archiving Syst. (KOASAS)*, 2016, pp. 1–30.

Jung-Hoon Kim received the B.S. and M.S. degrees in computer science from Hanyang University, Seoul, South Korea, in 1998 and 2000, respectively. He is currently pursuing the Ph.D. degree with the Department of Semiconductor and Display Engineering, Sungkyunkwan University, Seoul.

Since 2006, he has been a Firmware Engineer with the Storage Department, Samsung Electronics Corporation, Suwon, South Korea. His current research interests include memory systems, embedded systems, and operating systems.

Sang-Hoon Kim received the B.S. and Ph.D. degrees in computer science from the Korea Advanced Institute of Science and Technology, Daejeon, South Korea, in 2002 and 2016, respectively.

He is currently a Post-Doctoral Associate with Virginia Tech, Blacksburg, VA, USA. His current research interests include operating systems, storage systems, and mobile systems.

Jin-Soo Kim (M'89) received the B.S., M.S., and Ph.D. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1991, 1993, and 1999, respectively.

He is currently a Professor with Sungkyunkwan University, Seoul. He was an Associate Professor with the Korea Advanced Institute of Science and Technology, Daejeon, South Korea, from 2002 to 2008. He was a Senior Member of Research Staff with Electronics and Telecommunications Research Institute, Daejeon, from 1999 to 2002, and an Academic Visitor with IBM T. J. Watson Research Center, Cambridge, MA, USA, from 1998 to 1999. His current research interests include embedded systems, storage systems, and operating systems.

Prof. Kim is a member of the IEEE Computer Society.