Improving Performance and Lifetime of Large-Page NAND Storages Using Erase-Free Subpage Programming

†Myungsuk Kim, †Jaehoon Lee, ‡Sungjin Lee, †Jisung Park, †Jihong Kim
†Seoul National University, ‡Inha University
{morssola75, jhoonl89, jspark, jihong}@davinci.snu.ac.kr, sungjin.lee@inha.ac.kr

ABSTRACT

Recent NAND flash devices have large page sizes. Although large pages are useful in increasing the flash capacity, they can degrade both the performance and lifetime of flash storage systems when small writes are dominant. We propose a new NAND programming scheme, called erase-free subpage programming (ESP), which allows the same page to be programmed multiple times for small writes. By avoiding internal fragmentation, the ESP scheme reduces the overhead of garbage collection for large-page NAND storages. Experimental results show that an ESP-aware FTL can improve the IOPS and lifetime by up to 74% and 177%, respectively.

1. INTRODUCTION

With continuing semiconductor process scaling (e.g., 10 nm-node process technology [1]) and various technical innovations (such as 3D VNAND [2]), the NAND device capacity has been significantly increased in a cost-effective fashion. For example, the NAND device capacity has increased by about 8 times from 16 GB (in 4x nm-node in 2008) to 128 GB (in 1x nm-node in 2016) while improving the cost-per-bit of NAND devices by about 7 times under the same form factor [1]. Although this rapid increase in the NAND device capacity has enabled many new market opportunities for flash-based storage systems, these high-capacity NAND devices also present several new technical challenges for maximizing their potential benefits at the storage system level. In this paper, we focus on large-page problem of high-capacity NAND devices.

As shown in Fig. 1, the NAND page size, which was initially 256 B, has steadily increased up to 16 KB in 2016 [1, 2] as with the increasing NAND device capacity. When the NAND device capacity gets higher, a larger page size is preferred because of two main reasons. First, a larger NAND page can achieve a higher I/O bandwidth. This is because more cells can be simultaneously read/written on larger NAND pages while the fixed I/O management cost (such as a handshaking cost between a SW driver and a NAND device) can be better amortized over a large amount of data. Second, although there is no technical barrier to supporting a smaller page size for high-capacity NAND devices, such NAND devices would be less cost-efficient. This is because they need additional peripheral circuits to access a large number of smaller pages, thus significantly increasing the overall NAND chip size.

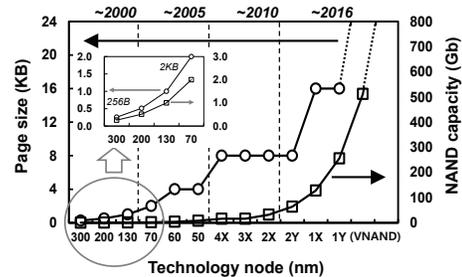Although, from the I/O bandwidth and cost perspective, a large page size is desirable in high-capacity NAND de-

Figure 1: Trend of the NAND page size and capacity.

vices, it can cause a serious performance degradation at the storage system level, in particular, when small writes are dominant. (In this paper, when the size of a write request is smaller than the size of a physical NAND page, we call such a write request *small*). In a conventional page-level mapping scheme, which we call a coarse-grained mapping (CGM) scheme, a small write would be mapped to a full page, wasting a large portion of the full page. Because of internal fragmentation, when small writes are dominant, the overhead of garbage collection (GC) significantly increases. When a large percentage of writes are small *updates*, the performance of the CGM scheme drops sharply by expensive read-modify-write (RMW) operations. Even if only a part of the full page is changed by a small write, the modified full page should be written after reading and modifying the full page. When small updates are dominant, for example, the IOPS of the CGM scheme may drop to about 15% of the peak IOPS because of frequent RMW operations.

A common solution to the poor performance of the CGM scheme with small writes is to employ a fine-grained mapping (FGM) scheme where the size of a logical page is smaller than that of a NAND page. In the FGM scheme, a write buffer is used so that small writes can be merged into full-page writes before they are flushed to a flash storage system. In the FGM scheme, if small writes can be merged into full-page writes, no internal fragmentation occurs, thus not affecting the GC efficiency. If a small update is made by the size of a logical page in the FGM scheme, a small update becomes a simple out-place write, which does not require RMW operations. Although the FGM scheme works generally better than the CGM scheme, when small writes cannot be merged into full-page writes, the GC efficiency in the FGM scheme is quickly deteriorated to the level in the CGM scheme. For example, when small writes are mostly synchronous requests, they must be stored right away and miss an opportunity to be merged in the write buffer. Since many real workloads exhibit this characteristics, the FGM scheme shows large fluctuations in performance and lifetime depending on workload characteristics. Another main drawback of the FGM scheme is that it requires a very large

logical-to-physical (L2P) mapping table because L2P mapping is based on a smaller logical page.

In this paper, we propose a new system-level technique that solves the large-page problem of high-capacity NAND devices. Our proposed technique is based on a new NAND programming scheme, called *erase-free subpage programming (ESP)*, which allows the same NAND page to be programmed *multiple times* in a sequential subpage order with the reduced retention capability. Since a small write can be serviced using a subpage write, there is no internal fragmentation for small writes in the ESP scheme, thus improving the performance and lifetime of a flash-based storage system even when synchronous small writes are dominant.

Since the ESP scheme reduces the data retention time of subpages (over full pages) as a side effect of subpage programming, it is important for an FTL to monitor the data retention requirement of small writes. When data require longer retention times than the retention capability of the subpage, the FTL needs to reclaim them to different pages. Based on the proposed ESP scheme with an adaptive retention management technique, we have implemented an ESP-aware FTL, called `subFTL`, which stores small writes into subpages in an erase-free fashion, thus significantly reducing the GC overhead of for large-page NAND storage systems. Our experimental results using various benchmark programs show that `subFTL` can improve the IOPS and lifetime by up to 74% and 177% respectively, over the FGM-based FTL. In `subFTL`, we also significantly reduced the L2P mapping memory requirement over the FGM scheme by managing the subpage region and full-page region with different mapping methods in a hybrid fashion.

The rest of this paper is organized as follows. In Section 2, we evaluate how small writes affect the performance and lifetime of flash storages. Section 3 describes the proposed ESP scheme. In Section 4, we present the design and implementation of `subFTL`. Experimental results follow in Section 5, and related work is summarized in Section 6. Section 7 concludes with a summary and future work.

2. IMPACT OF SMALL WRITES ON PERFORMANCE AND LIFETIME

Before we describe the proposed ESP scheme, we present how the performance and lifetime of a flash storage system are affected by small writes under the CGM scheme and FGM scheme. Since small writes affect mostly the GC efficiency, in this section, we focus on understanding the impact of each scheme on the GC overhead. We assume that the size of a write request is given by a multiple of the subpage size S_{sub} which is $1/N_{sub}$ of the full-page size S_{full} (i.e., $S_{full} = N_{sub} \times S_{sub}$). Furthermore, we define that the request WAF $w(r)$ of a small request r with the size s to be s_{flash}/s where s_{flash} represents the size of data written to the flash memory for the request r. For example, in the CGM scheme, when a 4-KB write request r_1 was written to a 16-KB full page, $w(r_1)$ is 4.

In order to understand the impact of small writes on the overall storage performance, we measured IOPS values of two schemes on our 16-GB emulated SSD with $S_{full} = 16$ KB and $S_{sub} = 4$ KB using several I/O workloads generated from Sysbench [3]. (For a detailed description of our emulated SSD, see Section 5.) As shown in Fig. 2, these workloads are different in two ratios, the ratio r_{small} of small writes to total writes and ratio r_{synch} of synchronous small writes to total small writes. In each measurement, our emulated SSD was preconditioned to the same SSD steady state by filling 10-GB data to the SSD before running Sysbench.

Fig. 2(a) shows normalized IOPS values for the CGM and FGM schemes under different combinations of r_{small} and r_{synch}. IOPS values were normalized over that of the FGM scheme when both r_{small} and r_{synch} were set to 0 (which has
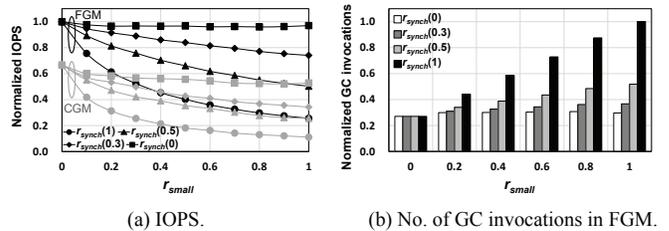

(a) IOPS. (b) No. of GC invocations in FGM.

Figure 2: Effects of small writes.

the highest IOPS). In both the CGM and FGM schemes, the IOPS is directly affected by r_{small} and r_{synch}: the smaller r_{small} (and r_{synch}), the higher IOPS. As shown in Fig. 2(a), although IOPS values of the CGM scheme and FGM scheme are proportional to $(1 - r_{small})$, the CGM scheme shows much worse performance than the FGM scheme because the IOPS of the CGM scheme is dominated by the number of expensive RMW operations.[1]

To understand why IOPS values are affected by r_{small} and r_{synch} in the manner shown in Fig. 2(a), we measured the number of GC invocations in each scheme. Fig. 2(b) shows how the number of GC invocations changes under different combinations of r_{small} and r_{synch} in the FGM scheme.[2] The values were normalized over that of the FGM scheme when both r_{small} and r_{synch} were set to 1 (which has the highest number of GC invocations). The number of GC invocations changes in a similar fashion as the IOPS values of Fig. 2(a). When small writes are dominant, the GC efficiency largely depends on the average request WAF value of a small request. Since the average request WAF value of a small request is directly proportional to r_{small} and r_{synch}, we observe that the number of GC invocations increases as with increasing r_{small} and r_{synch} values. The higher the number of GC invocations, the higher the number of erase operations, so the lifetime of a flash storage system is affected in a similar manner shown in Fig. 2(b).

The key insight from our evaluation study in this section is that when small writes are dominant, in order to achieve high performance and long lifetime of a flash storage, it is important to keep the average request WAF value of a small write *small*. The obvious solution to lower the average request WAF value of a small write would be to support *subpage-granularity writes* so that internal fragmentation can be avoided. Our proposed ESP scheme, which achieves the average request WAF value very close to 1 for a small write, is one of such schemes.

3. ERASE-FREE SUBPAGE PROGRAMMING

3.1 Subpage Programming Support

In order to support subpage programming, a NAND device should support writes in a subpage unit. Since NAND flash memory supports bit-level selective operations with the self boosting program inhibit (SBPI) scheme [4], subpage programming can be easily implemented without any new hardware support. Fig. 3 illustrates how the SBPI scheme supports bit-by-bit selective program operations. When a page WL_k is programmed, its i-th cell within the page is selectively programmed depending on the value of BL_i. If

---

[1]Note that in Fig. 2(a), when $r_{small} = 0$, there is a large IOPS gap between the FGM scheme and CGM scheme. Intuitively, there should be no IOPS difference because all writes are full-page writes when $r_{small} = 0$. This IOPS difference is due to misaligned logical addresses. In the CGM scheme, when a 16-KB write is not aligned to the 16-KB page boundary, it is split into two small requests, which require RMW operations.

[2]The result for the CGM scheme is very similar to Fig. 2(b), so we omit it in the paper because of a page limit.
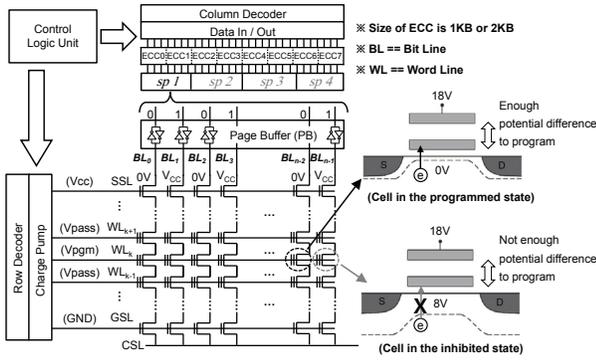
Figure 3: Bit-level selective programming support in NAND flash memory using the SBPI scheme.

(a) Cell V_{th} distributions after sp1 subpage programming.

(b) Cell V_{th} distributions after sp2 subpage programming.

Figure 4: Effect of subpage programming on NAND reliability.

BL_i is set to 0 (i.e., 0V), the i-th cell is programmed. If BL_i is set to 1 (i.e., Vcc), the i-th cell is not programmed (i.e., inhibited). Therefore, when we need to write a subpage sp only within a full page fp, we set BL_i's for sp to 0 while the rest of BL_i's to 1. (For example, in Fig. 4(a), the first subpage is programmed while the second one is inhibited.)

Although subpage programming can be easily supported in NAND devices, it is rarely used in modern high-capacity NAND devices because it significantly impairs the reliability of NAND devices. When several subpages in the same page are successively programmed without intervening erase operations, the reliability of stored data is seriously lowered because of cell-to-cell coupling effect from neighboring cells and program disturbance.[3] The pages damaged by subpage programming cannot store data properly. For example, such pages cannot satisfy the data retention requirement of commercial-grade flash products (e.g., 1 year at 30°C in JEDEC Standard).

3.2 Basic Idea of ESP

Our proposed ESP scheme overcomes the reliability degradation problem of a generic subpage programming scheme by allowing a subpage write to a full page only when there is no valid subpage within the full page. This condition for a subpage write is motivated by our NAND characterization study. In our study, we observed that when the same page is programmed multiple times in a subpage granularity without erase operations between successive subpage programs, the bit error rate (BER) of each subpage is significantly different depending on whether the subpage was previously programmed or not. If the subpage was previously programmed, a subsequent program will destroy its data. On the other hand, if the subpage was not yet programmed, it can hold the written data with a reduced retention capability. By permitting a subpage program only when there is no valid subpage in a page, the ESP scheme can support multiple subpage writes in an erase-free fashion.

Fig. 4 illustrates the key idea of the ESP scheme assuming that a full page consists of two subpages, sp1 and sp2. As shown in Fig. 4(a), when sp1 is programmed, there is no degradation in the NAND reliability because this is a normal page program operation. However, when sp2 is subsequently programmed without an intervening erase, the data in sp1 is so severely corrupted that the number of bit errors in sp1 exceeds the maximum ECC correction capability. (That is, sp1 loses its data because of the program disturbance and coupling effect from the subsequent sp2 program.) On the
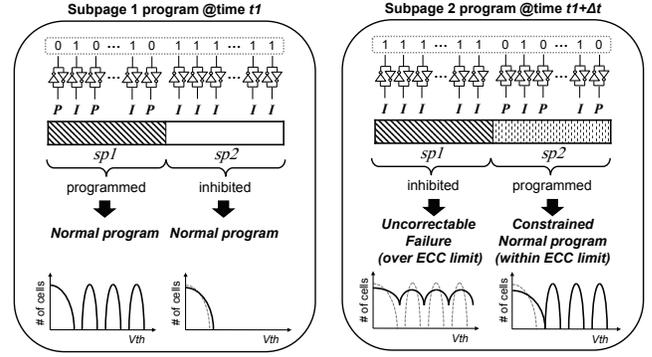
other hand, as shown in Fig. 4(b), although sp2 was programmed twice as with sp1, sp2 can store data properly after a subpage write to sp2. Since sp2 was inhibited during the sp1's program, the reliability impact of sp1's program was rather limited to a degree that reduces the data retention time of sp2. Although the data retention time of sp2 was reduced, the data can be stored normally during its reduced data retention time.

In order for the ESP scheme to work properly, therefore, there should be no valid subpage within a page before a successive subpage write is performed. Fortunately, many small writes have short retention requirements [5], thus making the ESP scheme an effective solution to solve the large-page problem when small writes are dominant.

3.3 Subpage-Aware NAND Retention Model

In order to take advantage of the ESP scheme at the FTL level, it is important for an FTL to understand the reduced retention capability of subpage writes. Using 2x nm-node TLC NAND chips, we constructed a simple NAND retention model for supporting subpage programming. To make the NAND retention model, we conducted experiments using a total of 81,920 pages out of 20 TLC NAND chips. In our test TLC chips, a page consists of 4 subpages. Since the retention capability of the subpage to be programmed depends on how many program operations were performed before programming the current subpage, we classify subpages depending on the number of previous program operations to subpages. When a subpage sp was programmed k times before programming the current subpage, sp is called an N_{pp}^k-type subpage. For example, in Fig. 4(b), sp2 is an N_{pp}^1-type subpage while sp1 is an N_{pp}^0-type subpage.

Fig. 5 shows how the data retention capability of a subpage changes under varying number of previous program operations before the subpage is programmed. As a measurement metric of the retention capability of a subpage, we used the NAND retention BER[4]. Following the endurance requirement of TLC NAND devices, we performed 1K P/E cycles for our TLC NAND devices before our measurements. Retention BER values were normalized over the retention BER of an N_{pp}^0-type subpage right after 1K P/E cycles. (This retention BER is commonly called as the endurance BER.) As shown in Fig. 5, the more the subpage experiences previous program operations, the higher the retention BER of the subpage. For example, the retention BER of an

---

[3]Even if cells in a page are inhibited during a program operation, their V_{th} distributions are affected from the high V_{pgm} stress, which effectively causes the inhibited cells to be softly programmed. This undesired phenomenon is called a program disturbance.

[4]The NAND retention BER is based on the number $N_{ret}(x,t)$ of retention errors after t-year retention time for x pre-cycled NAND cells [6].
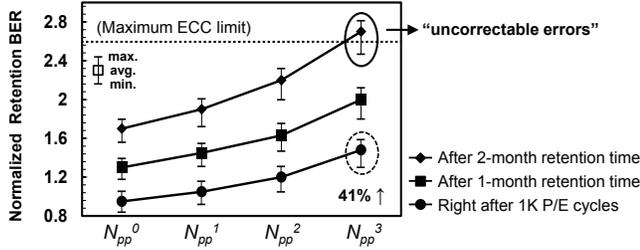
Figure 5: Impact of previous program operations on the retention capability of subpages.

N_{pp}^3-type subpage is 41% higher than that of an N_{pp}^0-type subpage right after 1K P/E cycles. Furthermore, it is clear that the retention BER gets worse as the retention time requirement gets longer. For example, an N_{pp}^3-type subpage satisfies the 1-month retention time requirement while it fails to meet the 2-month retention time requirement, resulting in uncorrectable errors. In order to make a NAND retention model that can be applied to all subpages, we conservatively interpreted the measurement results shown in Fig. 5. Our subpage-aware NAND retention model assumes that each subpage can hold its data properly for one month only. We used this model in the following section in implementing an ESP-aware FTL.

4. SUBFTL: ESP-AWARE FTL

Based on our subpage-aware NAND retention model described in Section 3, we have implemented subFTL, a new ESP-aware FTL. SubFTL is based on a hybrid mapping FTL with additional modules for supporting erase-free subpage programming. In this section, we first explain an overall architecture of subFTL and then give detailed explanations of how subFTL avoids the internal fragmentation problem by leveraging the erase-free subpage programming feature. Note that, in the rest of this paper, we focus on explaining the handling of writes because there are no significant differences from conventional FTLs in handling reads.

Fig. 6 illustrates an overall organization of subFTL. For an illustration purpose, we assume that S_{full} and S_{sub} are 16 KB and 4 KB, respectively. SubFTL divides NAND flash into two regions, the subpage region and full-page region, which are managed in a different manner. In the subpage region, writes are handled in the unit of 4 KB. Using the ESP scheme allows us to write 4-KB subpages multiple times to the same physical page and helps better utilize NAND space without internal fragmentation. Since the subpage region must be managed by the fine-grained mapping unit (i.e., 4-KB), a relatively large amount of memory is required to maintain mapping information. The full-page region is managed by using a CGM scheme based on full-page programming. While the full-page region can be supported with a smaller mapping table, costly RMW operations are unavoidable when small writes are stored in the full-page region.

The main design goal of subFTL is to realize fragmentation-free NAND space management using subpage programming, but, at the same time, we want to maintain a small mapping table. For this reason, only 20% of the total flash space is assigned to the subpage region, and the rest is allocated to the full-page region. While the size of the subpage region is limited, subFTL can fully exploit the benefits of the subpage programming by intelligently placing data in two different regions in accordance with their update characteristics.

4.1 ESP-Aware Data Placement

Upon receiving write requests from the host, subFTL puts them into a write buffer to merge several small writes with consecutive logical block addresses into one sequential write.
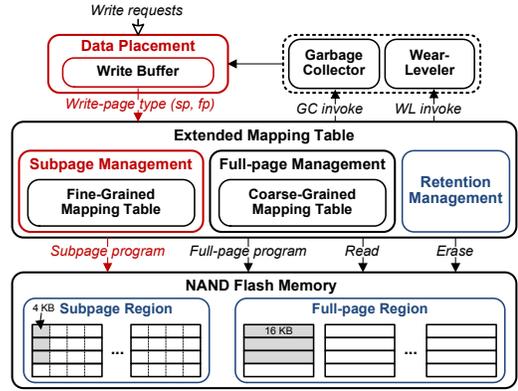

Figure 6: An organizational overview of subFTL.

Before flushing a buffered write to NAND flash, subFTL has to decide a target flash region to which the flushed data should be sent. In the current version of subFTL, the length of a flushed write request is used as a criterion that decides a target region – if its length is shorter than S_{full}, it is sent to the subpage region; otherwise, it is allocated to the full-page region. When the size of a flushed write request is larger than S_{full} but it is not multiples of S_{full}, the request is split into two regions. For example, if the size of a flushed write request was 20 KB, the first 16-KB data is sent to the full-page region while the rest 4-KB data is sent to the subpage region. In this way, subFTL always sends small writes to the subpage region, completely avoiding internal fragmentation.

Although it seems too simple, this data placement strategy is effective in several aspects. First, even though a relatively small portion of space is assigned to the subpage region (i.e., 20% of the total capacity), it is large enough to accommodate incoming data because only small writes are stored on it. Second, according to our experiments with various traces, small writes are likely to have higher update frequencies than large writes. Similar observations were also reported in previous studies [5]. In subFTL, therefore, hot and cold pages tend to be isolated within two different regions. Thanks to a simple but efficient data separation heuristic between hot data and cold data, subFTL effectively lowers the GC overhead. Third, frequent updates in the subpage region create many invalid data, which makes it easier for us to manage the subpage region that has shorter retention time (see Section 4.2).

The full-page region is managed in exactly the same way as the CGM-based FTLs – subFTL employs the same L2P mapping, garbage collection, and wear-leveling algorithms as the CGM-based FTL for the full-page region. The subpage region, however, must be treated in a different manner because of its unique subpage programming feature.

4.2 Subpage Region Management

The writing policy of subFTL in the subpage region is different from that in the full-page region. Fig. 7 depicts how subFTL writes data to the subpage region. Suppose that there are two blocks, B_X and B_Y, in the subpage region, each of which has four 16-KB full pages, i.e., $B_X = \{p_0^X, ..., p_3^X\}$ and $B_Y = \{p_0^Y, ..., p_3^Y\}$. Each full page is divided into four 4-KB subpages, i.e., $p_0^X = \{sp_{(0,0)}^X, ..., sp_{(0,3)}^X\}$, which can be individually programmed. B_X is thus a set of 16 subpages, i.e., $B_X = \{sp_{(0,0)}^X, sp_{(0,1)}^X, ..., sp_{(3,2)}^X, sp_{(3,3)}^X\}$. (For a simple illustration, we do not depict channels and ways in Fig. 7, but subFTL is developed to maximize I/O parallelism of a multi-channel architecture.)

Suppose that a sequence R of write requests, which are sent to the subpage region, is expressed as follow: $R = <r_0, r_1, r_2, ...>$, where r_i is a logical address of a 4-KB sub-

(a) The initial status of B_X and B_Y. (b) The status after 8 subpage writes. (c) The status after 3 more subpage writes.
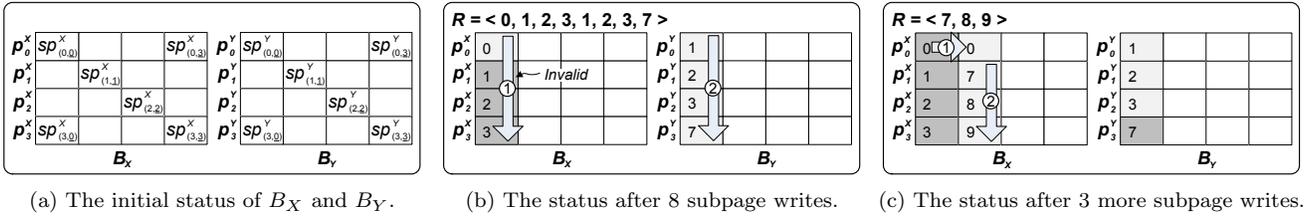
Figure 7: An illustrative example of the writing policy of subFTL in the subpage region.

page write and r_i arrives before r_j if $i < j$. In Fig. 7(b), $R =$ $<0, 1, 2, 3, 1, 2, 3, 7>$. SubFTL redirects r_i to the subpage region so that the 0th subpages of blocks are filled up before other subpages. In Fig. 7(b), four requests, $0, 1, 2,$ and 3 at the beginning of the sequence R, are written to the four 0th pages, $sp^X_{(0,0)}, sp^X_{(1,0)}, sp^X_{(2,0)}$ and $sp^X_{(3,0)}$, of B_X, respectively (see ① in Fig. 7(b)), while four requests, $1, 2, 3,$ and 7, at the rest of the sequence R are written to $sp^Y_{(0,0)}, sp^Y_{(1,0)}, sp^Y_{(2,0)}$ and $sp^Y_{(3,0)}$ of B_Y, respectively (see ② in Fig. 7(b)).

Once all of the 0th subpages are used up, subFTL attempts to send the following requests to the 1st subpages in the blocks. Unfortunately, writing data to 1st subpages results in the corruption of data in the corresponding 0th subpages in the same full page. For example, the content of $sp^X_{(0,0)}$ is destroyed after new data is written to $sp^X_{(0,1)}$. To avoid this, subFTL tries to select a block with only obsolete subpages containing invalid data, and then uses it as a target block where new data can be written. This is a reasonable choice because if 0th subpages are all obsolete, the data loss of 0th subpages does not affect the data reliability. If subFTL cannot find a block with no valid subpage, subFTL looks for a block with the smallest number of valid subpages. Fig. 7(c) illustrates such a case where subFTL selects B_X as a block with the smallest number of valid subpages. As shown in Fig. 7(b), since the new versions of $1, 2,$ and 3 were written to B_Y, the old versions of $1, 2,$ and 3 in B_X are no more valid. However, since 0 in B_X is still valid, subFTL moves 0 to the next subpage $sp^X_{(0,1)}$ (see ① in Fig. 7(c)). Subsequent subpage writes, $7, 8,$ and 9, can be written to $sp^X_{(1,1)}, sp^X_{(2,1)},$ and $sp^X_{(3,1)}$, respectively (see ② in Fig. 7(c)). In this way, subFTL keeps writing data to the subpage region without any data corruption.

It is worth noting that extra I/Os for the movements of valid subpages to their neighboring ones account for an insignificant portion of the total I/Os. Thus, their impact on performance is negligible. As mentioned before, subpage writes have a high temporal locality, so the majority of them become obsolete. Moreover, subFTL allocates 20% of the total space to the subpage region, which facilitates more subpages becoming obsolete before blocks (where they reside) are being selected as a target.

Once all the free subpages in blocks are exhausted, subFTL performs garbage collection to reclaim free blocks. SubFTL selects a victim block with the smallest number of valid subpages, and then decides whether or not to move valid subpages to a free block (which is reserved for garbage collection before). For subpages that have been updated at least once (since it was written to the subpage region), subFTL moves them to the free block because they are likely to be updated again. On the other hand, subpages that haven't been updated (since it was written to the subpage region) are evicted to the full-page region. This causes RMW operations, but allows us to regularly evict cold subpages to the full-page region, helping us keep only hot data in the subpage region.

Since hot data is frequently overwritten, blocks in the subpage region are more rapidly worn out than those in the full-page region. This unbalanced wearing problem is solved by using existing wear-leveling algorithms. The type of blocks (i.e., subpage or full-page blocks) are decided at the program time, not at the design time. Thus, converting subpage blocks to full-page ones (or vice versa) can be done by swapping data between them as commonly done in conventional wear-leveling.

In order to mitigate memory overhead for fine-grained L2P mapping, subFTL employs a hash table to manage the subpage region. The memory requirement for the hash table is not huge because each full page can hold only one valid subpage – that is, the number of hash entries pointing to valid subpages is one fourth of the total subpages. Therefore, even with a relatively small hash table, subFTL can quickly find a physical location of a given logical subpage, without being severely affected by hash collisions.

4.3 Retention Management

As discussed in Section 3, the subpage region has a shorter retention time than the full-page region – if data stay longer than 1 month, it cannot be retrieved due to uncorrectable errors. To prevent this, subFTL checks if there are subpages holding data longer than 15 days. If such subpages are found, subFTL evicts long-lived subpages to the full-page region. According to our experiments with various traces, such long-lived subpages account for a trivial portion of the total subpages, and its impact on performance is negligible.

5. EXPERIMENTAL RESULT

In order to evaluate the effectiveness of the proposed technique, we implemented subFTL as a host-level FTL using the open flash development platform [7]. SubFTL supported 512-GB flash capacity in maximum, but, for fast evaluations, we limited its storage capacity to 16 GB. This reduction of the storage capacity did not distort experimental results because the performance of the FTL was decided by the characteristics of input workloads, not by the storage capacity. Our flash device was composed of 8 channels, each of which had 4 TLC NAND chips.

Based on our measurement study with real TLC NAND chips, a full-page write latency (i.e., 16 KB) was set to 1600 μs, while a subpage write latency was 1300 μs. This is because, when programming a 4-KB subpage, both the number of bit lines (BLs) to be precharged in verify-read operations and the length of word lines (WLs) to be driven to high V_{pgm} are reduced. This shortens a RC delay and reduces a setup time in program operations [8].

The five benchmarks were used for our evaluations: Sysbench (system performance testing benchmarks), Varmail (mail-server workload benchmarks), Postmark (mail-server workload benchmarks), YCSB (Yahoo! Cloud Serving Benchmark running on Cassandra), and TPC-C (one of on-line transaction processing benchmarks). Each benchmark has different characteristics, including r_{small} and r_{synch}. We have compared the performance of subFTL with two conventional FTLs, cgmFTL and fgmFTL, where cgmFTL is based on the CGM scheme and fgmFTL is based on the FGM scheme. Fig. 8(a) shows the normalized IOPS of the three FTLs under the five benchmarks. SubFTL improves IOPS by up to 249.2% and 74.3% (120.8% and 35.1%, on average) over
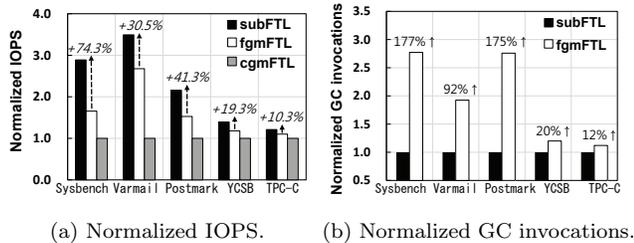
(a) Normalized IOPS.　　　(b) Normalized GC invocations.

Figure 8: Performance comparisons of three FTLs.

Table 1: Detailed analysis of `subFTL`.

	Sysbench	Varmail	Postmark	YCSB	TPC-C
% of small write	99.7%	95.3%	99.9%	19.3%	11.8%
average request WAF	1.005	1.007	1.003	1.005	1.008

`cgmFTL` and `fgmFTL`, respectively. As expected, `cgmFTL` performs the worst because of a large number of RMW operations. For example, 89.3% of the total writes in Varmail were serviced using RMW operations.

To better understand why `subFTL` outperforms `fgmFTL` (as well as `cgmFTL`), we measured the number of GC invocations, in `fgmFTL` and `subFTL`, respectively, as shown in Fig. 8(b). The number of GC invocations in `subFTL` is reduced by up to 177% (95.2%, on average) over `fgmFTL`. As expected, `cgmFTL` and `fgmFTL` often waste 16-KB full pages to accommodate 4-KB subpage writes. This underutilization of full-page space never occurs in `subFTL` because of subpage writes in `subFTL`. By efficiently utilizing NAND space, `subFTL` greatly reduces extra I/Os caused by garbage collection as well as the number of GC invocations, thereby improving overall IOPS of all the benchmarks.

Fig. 8(a) shows that there exists a significant variation in the degree of performance improvement in `subFTL` over `fgmFTL` depending on different benchmarks. This variation largely depends on r_{small} and r_{synch} values of different benchmarks. In the cases of Sysbench, Varmail, and Postmark, where synchronous small writes account for a considerable proportion (more than 95%) of the total writes, the performance improvement by `subFTL` is significant. For the other benchmarks, YCSB and TPC-C, relatively smaller performance gains are observed because of a small proportion (less than 20%) of 4-KB writes. As depicted in Fig. 8(a), however, `subFTL` still exhibits 19.3% and 10.3% higher IOPS than `fgmFTL` for YCSB and TPC-C, respectively.

We finally evaluate the average request WAF of small writes in `subFTL`. As summarized in Table 1, the values of average request WAF is very close to 1.0, regardless of the benchmarks. This indicates that `subFTL` can avoid almost all internal fragmentation as well as RMW operations. The average request WAF, however, is not exactly 1.0 `subFTL` incurs two types of small extra I/Os, one for migrations of long-lived subpages within the subpage region and the other for evictions of cold subpages to the full-page region.

6.   RELATED WORK

There have been several studies that attempt to mitigate various side effects of small writes in flash storage [9, 10, 11]. However, most existing techniques support *quasi-*subpage programming in that they do not allow *multiple* subpage programs on the same page in an erase-free fashion as in `subFTL`. Jin *et al.* proposed the *sector-log* technique for large-page NAND devices [9]. The sector-log technique is similar to `subFTL` in that it is based on a hybrid mapping method. It appends small writes to a reserved log area managed by fine-grained mapping, while storing full-page writes to an ordinary flash area that is managed by coarse-grained mapping. However, unlike `subFTL`, since the sector-log technique supports subpage programming at the logical level as with other FGM-based FTLs, its performance suffers when synchronous small writes occur fairly frequently. Kim *et al.* proposed another subpage-based programming technique for small writes [10]. This technique, unlike `subFTL`, uses subpage programming as a means to extend the life-

time of NAND flash by avoiding unnecessary write stress to NAND pages for small writes. Furthermore, this technique is not a real subpage programming technique because the partially programmed page should be erased before another subpage on the same page can be programmed.

The subpage programming technique [11] proposed by Zhang *et al.* supports multiple program operations on the same page without intervening erase operations as in `subFTL`. However, this technique targets SLC-mode pages only, thus limiting its applicability to high-capacity NAND devices which are mostly based on MLC/TLC flash memory.

7.   CONCLUSIONS

We have presented a new system-level technique that solves the large-page problem of high-capacity NAND devices. Our proposed technique is based on the ESP scheme, which enables the same page to be programmed multiple times by a subpage granularity in an erase-free manner. Since the ESP scheme can avoid internal fragmentation when small writes are written to large pages, it significantly reduces the overhead of garbage collection for flash storage systems with large-page flash memory. In order to overcome the reduced data retention capability of subpages written by the ESP scheme, we exploit the lifetime characteristics of small writes which tend to have short retention times. Based on our novel subpage-aware NAND retention model, we have implemented an ESP-aware FTL, `subFTL`, which takes advantages of ESP-enabled NAND devices. Our experimental results show that `subFTL` can improve the IOPS by up to 74% while the number of GC invocations are reduced by 177% over the FGM-based FTL.

The current version of `subFTL` can be extended in several directions. For example, although we did not consider subpage read operations in the current version of `subFTL`, we plan to support subpage read operations in the next version of `subFTL`. If subpage read operations can be made faster than full-page reads, we believe that they can be useful for read latency-sensitive applications.

REFERENCES

[1] S. Lee *et al.* A 128Gb 2b/Cell NAND Flash Memory in 14nm Technology with tPROG=640µs and 800MB/s I/O Rate. In *Proc. IEEE Int. Solid-State Circuits Conf.*, 2016.

[2] D. Kang *et al.* 256Gb 3b/Cell V-NAND Flash Memory with 48 Stacked WL Layers. In *Proc. IEEE Int. Solid-State Circuits Conf.*, 2016.

[3] Sysbench. http://github.com/akopytov/sysbench.

[4] K. Suh *et al.* A 3.3V 32 Mb NAND Flash Memory with Incremental Step Pulse Programming Scheme. *IEEE Tran. on Consumer Electronics*, 30(11):1149–1156, 1995.

[5] L.-P. Chang *et al.* Hybrid Solid-State Disks: Combining Heterogeneous NAND Flash in Large SSDs. In *Proc. Design Automation Conf.*, 2008.

[6] J. Jeong *et al.* Dynamic Erase Voltage and Time Scaling for Extending Lifetime of NAND Flash-based SSDs. *IEEE Tran. on Computers*, PP(99), 2016.

[7] S. Lee *et al.* Application-Managed Flash. In *Proc. USENIX Conf. on File and Storage Technologies*, 2016.

[8] http://www.anandtech.com/show/7147/micron-announces-16nm-128gb-mlc-nand-ssds-in-2014.

[9] S.-W. Jin *et al.* Sector Log: Fine-Grained Storage Management for Solid State Drives. In *Proc. ACM Symp. on Applied Computing*, 2011.

[10] J.-H. Kim *et al.* Subpage Programming for Extending the Lifetime of NAND Flash Memory. In *Proc. Design, Automation and Test in Europe Conf. and Exhib.*, 2015.

[11] X.-B. Zhang *et al.* Reducing Solid-State Storage Device Write Stress through Opportunistic In-place Delta Compression. In *Proc. USENIX Conf. on File and Storage Technologies*, 2016.