Improving I/O Performance of Large-Page Flash Storage Systems Using Subpage-Parallel Reads

Jisung Park[1], Myungsuk Kim[1], Sungjin Lee[2], and Jihong Kim[1]
[1]Department of Computer Science and Engineering, Seoul National University
[2]Department of Information and Communication Engineering, DGIST
Email: [1]{jspark, morssola75, jihong}@davinci.snu.ac.kr, [2]sungjin.lee@dgist.ac.kr

***Abstract*—Handling small read requests is important on large-page flash storage systems because small reads tend to waste the read bandwidth. We present a system level solution that avoids wasting the read bandwidth based on a new page read operation, called the subpage-parallel read (SPREAD). SPREAD achieves an optimal latency for a small read request by reading requested subpages only. By improving the read performance of applications as well as garbage collection, SPREAD improves the overall I/O performance. Experimental results show that an SPREAD-aware FTL can improve the IOPS and read latency by up to 122% and 56%, respectively.**

I. INTRODUCTION

In recent years, the capacity of a NAND flash chip has steadily increased by about 30% per year [1] [2]. Although this dramatic improvement in the capacity per flash chip helped flash-based storage systems to be widely adopted from mobile systems to enterprise servers, these high-capacity NAND chips have also introduced several technical difficulties at the storage system level because of their capacity-oriented design decisions such as a large block size and a large page size. For example, while the capacity of a NAND chip was increased from 16 Gb to 128 Gb, the NAND page size was increased by 8 times as well from 2 KB to 16 KB [1] [2]. In this paper, we investigate the large-page problem of high-density NAND devices from a read operation's perspective.

In order to maximally increase the capacity of a NAND chip, a large NAND page is an (somewhat) inevitable design choice. If a NAND page were small, more peripheral circuits would be needed to access a larger number of small pages, thus sacrificing a valuable die area for the peripheral circuits. Furthermore, a large NAND page is helpful in improving the I/O bandwidth because it allows more cells to be read/written at the same time.

Although it is a reasonable design decision to use a large page at the flash chip level, a large page size can degrade the performance of a flash storage system. When a read needs to access small data, if the NAND page is much larger than the requested data, a large portion of the NAND page, which was not requested by the read, is unnecessarily read, thus increasing the read amplification factor (RAF) of the read. (In this paper, we call such a read *amplified*). Since a page is the minimum unit of read operations in NAND flash memory, many amplified reads occur on large-page flash storage systems, when small read requests are dominant. For example, even when a file system needs to read only 4-KB data from a NAND device with 16-KB pages, the full 16-KB page should be read, wasting the read bandwidth in reading the unwanted 12-KB data. When a workload is dominated by small random reads (e.g., workloads in key-value stores [3] and graph processing applications [4]), the effective read bandwidth of the flash storage system can be degraded to just 25% of the maximum read bandwidth.

In addition to amplified reads caused from small read requests of applications or operating systems, a large-page flash storage system can generate many amplified reads internally as well because of its common mapping scheme. When the storage system employs a fine-grained mapping (FGM)[1] scheme, multiple (small) logical pages can be mapped to a single (large) physical page. When these logical pages are updated, many physical pages contain both valid and invalid logical pages within the same physical page, thus resulting in many amplified reads for accessing these fragmented physical pages. For example, in our experiments, the read bandwidth during garbage collection barely reached about half of the maximum value under a workload with many small writes.

In this paper, we present a novel system-level technique that solves the amplified read problem of large-page SSDs. Our approach is based on an observation that the root cause of the amplified read problem is that reading a part of a NAND page is not *size-proportional* [2]. In order to improve the size proportionality of read operations, we aggressively exploited another observation at the device level: the latency of a small read can be significantly reduced by 1) sensing only necessary NAND cells and 2) transferring only demanded data. Based on our observations, at the device level, we devise a new page read operation called a subpage-parallel read (SPREAD). SPREAD can read multiple subpages at the same time while skipping unneeded subpages so as to enable us to reduce the read latency as well as read amplification factor. For a number of possible combinations of valid subpages within a given NAND page, SPREAD can decide whether to read a subpage or not at the subpage granularity, thus making it *size-proportional* even when reading multiple (not contiguous) subpages.

Based on the proposed SPREAD operation, we have developed an SPREAD-aware flash translation layer (FTL), called

[1]The FGM scheme is commonly used to avoid expensive read-modify-write operations for small writes on large-page flash storages [5]. For more details, see Section II.

[2]We define that a read is size-proportional if the read latency is proportional to the size of the requested data.

2575-257X/18/$31.00 ©2018 IEEE
DOI 10.1109/NVMSA.2018.00017

Authorized licensed use limited to: POSTECH Library. Downloaded on March 24,2023 at 06:23:58 UTC from IEEE Xplore. Restrictions apply.

spFTL. When a read is requested, spFTL decides which subpages to be read in parallel, and issues the optimal number of SPREAD operations to underlying NAND devices. SpFTL aggressively exploits SPREAD when internal page migrations are needed by SSD management tasks (e.g., garbage collection) so that no read bandwidth is wasted by amplified reads during page migrations. Our experimental results using various benchmark tools and real-world traces show that spFTL improves the read bandwidth and latency by up to 122% and 56%, respectively. By effectively exploiting the size proportionality of SPREAD, spFTL can also reduce garbage collection (GC) overheads by up to 13%, thus improving the overall I/O performance of flash storage systems even under write dominant workloads.

The rest of the paper is organized as follows. In Section II, we present a key motivation behind SPREAD by explaining how amplified reads affect the I/O performance. Section III describes the proposed SPREAD with its implementation details. In Section IV, we present the proposed SPREAD-aware FTL, spFTL. Experimental results follow in Section V, and Section VI concludes with a summary and future work.

II. Motivation

It is well known that the latency of page read tends to increase as the NAND page size becomes larger [6]. However, it does not mean that NAND devices with larger pages provide inferior performance all the time than ones with smaller pages. In fact, with large-page NAND, it is expected to get higher performance with an improved read bandwidth. For example, reading four separate 4-KB pages from the NAND device requires about 180 μs, but the same amount of data can be read from a 16-KB page in about 120 μs [1] [2]. However, our observation shows that many real-world applications fail to enjoy such high throughput of large-page NAND; instead, they seriously suffer from the degraded bandwidth due to the *read amplification*. We have found that this problem stems from frequent amplified reads which read the whole page (e.g., 16 KB), but actually use only part of it (e.g., 4 KB).

In order to figure out what mainly causes amplified reads, we have analyzed two popular applications in high performance computing (HPC) systems, a key-value store [3] and a graph processing application [4]. We first realized that the HPC applications themselves generate lots of small random reads to storage devices. Figure 1 shows distributions of read data sizes of the two applications. As shown in Figure 1, most values (over 90%) in the key-value store are smaller than 1 KB, and more than 99% of adjacency lists in the graph processing application are smaller than 4 KB. Moreover, since both the applications exhibit very low spatial locality [3] [4], performance improvements from page-cache or storage-buffer hits are marginal.

Data fragmentation is another root cause that creates amplified reads. Modern SSDs typically employ a fine-grained mapping (FGM) scheme that maps 4-KB logical pages to a

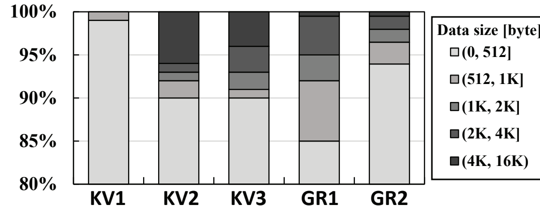


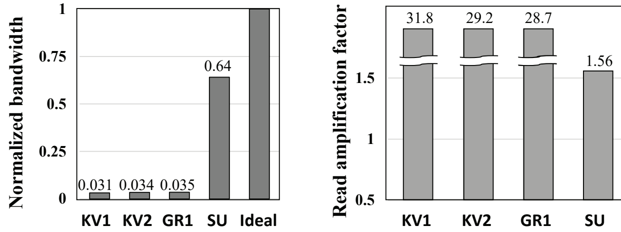
Fig. 1: Distributions of read data sizes.

larger physical page, say 16-KB[3]. The FGM scheme buffers four 4-KB logical pages and writes them to a 16-KB NAND page together. By doing this, FGM allows us to avoid expensive read-modify-write (RMW) operations in case where a small random update comes [5]. For example, suppose that one 4-KB logical page out of the four in the same physical page is updated. If the mapping unit were 16 KB (which is equivalent to a NAND page size), the FTL has to load the entire 16-KB NAND page to an internal buffer (read), update the buffered page with new 4-KB data (modify), and write it back to another NAND page (write). On the other hand, under the FGM scheme, up-to-date 4-KB data can be sent to a new physical page, and its old version is just marked invalid. While it is effective in avoiding RMW operations, the FGM scheme results in serious data fragmentation inside physical pages. Suppose that application wants to read those four 4-KB pages again. In this case, the FGM scheme has to read two 16-KB NAND pages: one to get three logical pages and the other one to get the recently updated one. As a result, 32-KB data have to be read from the NAND device to deliver requested 16-KB data to the host.

In order to understand the impact of amplified reads on I/O performance, we carried out preliminary experiments on a 16-GB DRAM-emulated SSD with 16-KB NAND pages. We used FIO benchmark tool [7] to generate similar distributions of read data sizes as in Figure 1. Figure 2(a) compares the read bandwidths of four different workloads, KV1, KV2, GR1, and SU. KV1 and KV2 mimic I/O distributions of the key-value stores, while GR1 generates similar I/O patterns as the graph processing application. SU (sequential update) is designed to assess the impact of fragmentation: it first sequentially writes 16 GB of data, and randomly writes 4-GB data with 4-KB I/Os, and finally issues 16-KB reads to the SSD. Ideal shows the ideal read bandwidth that NAND devices would achieve when there is no amplified read. As shown in Figure 2(a), the read bandwidths under all the workloads are far lower than the ideal bandwidth. It is worth noting that the read bandwidth of SU decreases by 36% over Ideal, even though there are only 16-KB reads. It indirectly shows that data fragmentation greatly lowers the overall read throughput. Figure 2(b) shows the RAF of the workloads which indicate the ratio of the data actually read from NAND devices to the data requested from the application. It clearly shows that amplified reads significantly increase the RAF, wasting the raw bandwidth of

[3]The logical page size is typically set to 4 KB, to be equal to the default block size of many modern file systems such as Linux ext4 and FAT32.

Authorized licensed use limited to: POSTECH Library. Downloaded on March 24,2023 at 06:23:58 UTC from IEEE Xplore. Restrictions apply.



(a) Normalized read bandwidth. (b) Read amplification factor.

Fig. 2: Impact of amplified reads on I/O performance.

large-page NAND devices to read unwanted data.

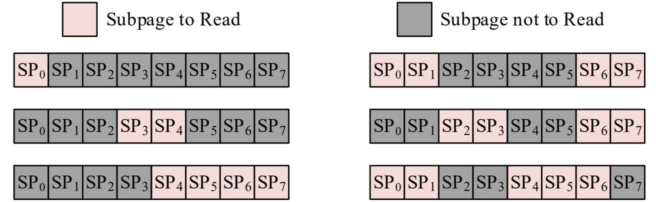
The key insight from our preliminary experiments is that, in order to achieve the high performance of large-page NAND flash, a new NAND device-level read scheme is required which enables us to selectively read data actually needed from NAND devices. One might think that higher-level approaches would be more feasible because these do not require us to modify underlying devices. For example, increasing a data block size of a file system might be able to remove the read amplification problem. However, this could not be an ultimate solution. As mentioned above, small random reads are dominant in many applications. Thus, regardless of a file-system block size (which is a data allocation unit), there will be many small reads since the smallest I/O unit size is still 512 B or 4 KB. Some might suggest to increase the minimum I/O unit size to 16 KB or more. Since applications always issue read requests larger than 16 KB to SSDs, it would eliminate amplified reads at the device level. However, because of the internal fragmentation at the file system level, most of data read from the SSD are unnecessary and are not used. That is, it just moves the amplified read problem to the file-system level, instead of getting rid of its root problem.

III. Subpage-Parallel Read

A. Design Requirements

Based on insights from Section II, we propose a new type of a NAND device that supports selective subpage reads from a NAND page, called SPRead. An SPRead-enabled NAND device exposes a reconfigurable subpage read interface that allows us to read *one* or *more* subpages *in parallel* from a large-size NAND page. By sensing a limited number of cells and transferring fewer bit values to the flash controller, SPRead shortens both the read (sensing) time t_R at the device and the time t_{DMA} for transferring data to the flash controller, thus significantly increasing the size proportionality of small reads.

In order to realize the idea of SPRead, we should answer the following two questions: 1) how to decide the size of a subpage which is the minimum unit of reading data, and 2) how to support parallel reads for multiple subpages scattered within a single large page. The answer to the first question is straightforward. The minimum unit of a subpage read must be aligned with the length of an error correction code (ECC) code word. In NAND flash memory, since all written data are encoded by an ECC function, we need to read an entire ECC code word for obtaining original data previously written. While it is different depending on NAND designs, recent NAND devices employ a 2-KB ECC code word [8] , which is small enough to minimize the read amplification by amplified reads[4].



(a) Contiguous subpages. (b) Fragmented subpages.

Fig. 3: Examples of parallel subpage read patterns.

The second question is a little more complicated, because we have to take into account various patterns of demanded subpages falling into a single large-size page. Figure 3 depicts example cases of when multiple subpage reads happen on a single page simultaneously. In Figure 3, we assume that the size of a page is 16 KB and an ECC code word is 2 KB (Unless otherwise stated, we keep using this NAND configuration). To meet the ECC requirement, we logically divide a 16-KB NAND page to multiple 2-KB subpages (SPs). Figure 3(a) illustrates cases where small read requests come from the host, but they result in contiguous subpages in the NAND page. These cases can be easily handled: for subpage reads, NAND devices just need the information about the offset of the start subpage, along with the read size. However, the problem gets more complicated when subpages are severely fragmented over the NAND page as shown in Figure 3(b). In these cases, providing the offset and the length of desired data is not sufficient to support SPRead.

The naive solutions for supporting those various subpage combinations may be 1) adding dedicated read commands for individual cases or 2) supporting only simple and limited combinations (e.g., only continuous SPs). The former option is not feasible because too many new NAND commands should be added to NAND chips to cope with all the possible combinations of subpage reads. In theory, when a NAND page has n ECC code words, $2^n - 1$ combinations are possible. The latter one would reduce the design complexity of SPRead, but it may lose a lot of optimization opportunities.

B. Implementation of SPRead *Command*

We address all the problems mentioned above with a simple yet effective co-design of NAND devices and flash controllers. Our key idea is to allow underlying flash controllers and NAND devices to be aware of each subpage's necessity, by

[4]The size of an ECC code word directly affects the error correction capability. In general, the longer code words, the stronger capability. The 2-KB LDPC is widely adopted in recent NAND flash memory for compensating the degraded NAND reliability.

Authorized licensed use limited to: POSTECH Library. Downloaded on March 24,2023 at 06:23:58 UTC from IEEE Xplore. Restrictions apply.

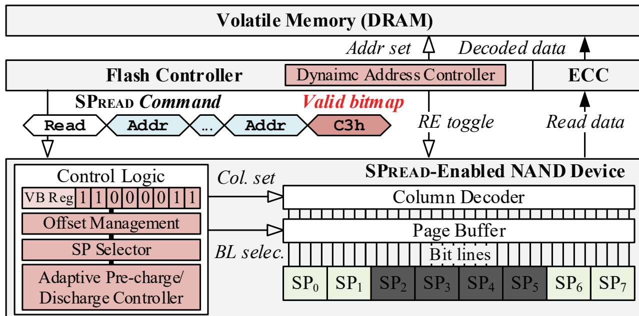


Fig. 4: An operational overview of SPREAD.

providing them a *valid bitmap*. The valid bitmap indicates the information of all the demanded subpages within a NAND page. Figure 4 illustrates how the proposed SPREAD command works with 16-KB NAND pages and 2-KB ECC words. As shown in Figure 4, the 8-bit valid bitmap for a page read is delivered to the target flash controller and NAND device via an extended NAND read command with an additional command slot for the valid bitmap. All the possible combinations of subpage reads, therefore, can be specified in a unified single NAND read command. Note that a full page read can be performed by using the same command with a valid bitmap of 0xFF.

With the new NAND read command, SPREAD effectively reduces t_R and t_{DMA}. SPREAD shortens t_R by selectively reading only necessary subpages within the target page. When an SPread command arrives, the given valid bitmap is temporarily kept in a dedicated register, called a *VB register*. Referring to the VB register, the SPREAD-enabled NAND device selectively pre-charges the desired bit lines (BLs), while the others are inhibited. The additional selective pre-charging logic, an *SP selector* in Figure 4, does this task by simply pulling down the BLs of inhibited subpages. For SPREAD, the elapsed times for pre-charging (t_{PRE}) and discharging (t_{DISCH}) BLs are not fixed, but vary depending on the number of BLs we want to sense. To deal with such variable elapsed times, it is required to add extra logics, denoted by *Adaptive Pre-charge/Discharge Controller* in Figure 4. Depending on the requested subpage configuration, it chooses appropriate t_{PRE} and t_{DISCH} which are sufficient for sensing all the desired BLs among the pre-defined values. This selective sensing makes t_R vary depending on the number of BLs since t_R is mostly decided by t_{PRE} and t_{DISCH}. That is, as the less BLs are being sensed, the shorter t_R is. This adaptive control of t_{PRE} and t_{DISCH} is the key to make SPREAD *size-proportional*.

SPREAD transfers only the sensed bit values to the storage firmware, which results in the reduction of t_{DMA}. The SPREAD-enabled NAND device maintains the valid bitmap inside, so it is able to specify the column offsets for fetching the required bytes from the page buffer, skipping unwanted subpages. This is accomplished by adding a simple FSM logic to the NAND device which dynamically generates column addresses. To selectively send the required bytes to the flash controller via DMA, it is inevitable to modify the DMA master engine (*Dynamic Address Controller* in Figure 4) in the flash controller. This modification, however, is actually simple; the DMA engine just needs to toggle RE signals for only bytes it wants, and the NAND device sends sensed bytes to the flash controller in sync with RE signals.

TABLE I: Estimated SPREAD latencies over difference sizes.

size [KB]	2	4	6	8	10	12	14	16
t_{DMA} [μs]	2.5	5	7.5	10	12.5	15	17.5	20
t_R [μs]	35	45	55	65	75	85	95	99
t_{SPR} [μs]	37.5	50	62.5	75	87.5	100	112.5	119

Although there exists a similar approach to perform such a *dynamic* DMA without the proposed SPREAD, it can rather increase t_{DMA} due to additional overheads. For example, an existing random data out (RDO) NAND command [9] allows us to manually modify the column offset of NAND devices. However, without a modification of underlying hardware, the storage firmware (i.e., FTL) is responsible for performing the dynamic DMA, and it should issue one or multiple RDO commands by itself to the target flash controller and NAND device. Such an approach can introduce additional overheads for handshaking and context switching.

To evaluate the benefit from SPREAD, we have estimated the SPREAD latency $t_{SPR}(n)$ for n-KB reads, which is the sum of $t_R(n)$ and $t_{DMA}(n)$. It is obvious that $t_{DMA}(n)$ is linearly proportional to the size n, since the sensed data are transferred by one byte per RE toggle. We can also expect that $t_R(n)$, which largely depends on $t_{PRE}(n)$ and $t_{DISCH}(n)$, must be shorter with a smaller n, since fewer BLs are pre-charged and discharged.[5] Although it is ideal to develop a new SPREAD-enabled device to get the exact value of $t_R(n)$, because of practical limitations in academia, we have estimated the $t_R(n)$ based on NAND device physics [11] [10] using known NAND device parameter values [1]. Table I summarizes the estimated $t_{SPR}(n)$ for reading n-KB data.

IV. spFTL: SPREAD-AWARE FTL

We have developed an SPREAD-aware FTL, `spFTL`, which leverages the proposed SPREAD command. Figure 5 depicts an overall organization of `spFTL`, which is based on an page-level FTL with the FGM scheme. The main addition in `spFTL` over other FGM-based FTLs is the SPREAD Mode Selector (SMS) module. The SMS module is responsible for deciding a proper SPREAD mode for the SPREAD command before the SPREAD command is sent to a NAND device.

Figure 5 illustrates how the SMS module constructs appropriate SPREADS for a host read request. In this example, we assume that a logical mapping size is 4 KB while the NAND page size is 16 KB. As shown in Figure 5, logically contiguous pages may map to multiple physical pages under the FGM scheme; four subpages whose logical addresses are `F0h`, `F1h`,

[5] $t_{PRES}(n)$ is hardly depends on the size n since BLs are pre-charged *in parallel*. However, $t_{DISCH}(n)$ is linearly proportional to n, since all the BLs are discharged though a single common source line (CSL) [10].

Authorized licensed use limited to: POSTECH Library. Downloaded on March 24,2023 at 06:23:58 UTC from IEEE Xplore. Restrictions apply.

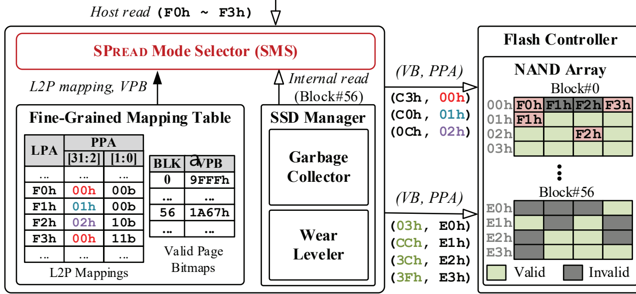


Fig. 5: An organizational overview of spFTL.

F2h and F3h were written together, but two of them F1h and F2h were overwritten with new data at different times later. Based on the logical address range of the host read request, the SMS module looks up the L2P mappings and figures out which subpages can be read in parallel by a single SPREAD. Once the subpages are decided, the SMS module sends an SPREAD command with a proper valid bitmap to a flash controller. In the above example, spFTL uses three SPREADS to read the 16-KB logical address range requested by host: one for F0h and F3h, one for F1h, and one for F2h. (Note that when SPREAD is not used, three full page reads, that is, 3×16-KB reads, are required.)

SpFTL exploits the SPREAD to reduce internal data copy overheads for garbage collection and wear-leveling. Once the garbage collector and wear-leveler are invoked, it is required to read physical NAND pages for copying valid data to other locations. In many cases, physical pages to read are severely fragmented with valid and invalid ones as shown in Figure 5. When the garbage collector or wear-leveler issues internal reads for a target block (e.g., Block#56 in Figure 5), the SMS module decides an optimal SPread mode for each page with the block's valid page bitmap (VPB) which indicates the status of all the physical pages in the block. By selectively reading only valid subpages with SPREAD, spFTL can avoid expensive amplified reads, thereby improving the garbage collection and wear-leveling performance.

V. EXPERIMENTAL RESULTS

A. Experimental Settings

To evaluate the effectiveness of the proposed SPREAD, we have implemented spFTL as a host-level FTL using an open flash development platform [12]. Our evaluation platform supports 512-GB capacity in maximum, but we limited its capacity to 16 GB for fast evaluations. The target SSD was composed of 4 channels, each of which had 2 NAND chips. Each chip was comprised of 512 NAND blocks with 256 16-KB NAND pages. Based on the same NAND specification used in our estimation [1], full-page write latency (t_{PROG}) and block erasure latency (t_{BER}) were set to 660 μs and 3.5 ms, respectively. The estimated values in Table I were used for modeling a 16-KB page read latency (99 μs) and SPREAD latency with the other sizes.

TABLE II: I/O characteristics of five benchmark workloads.

workload	read:write	dominant request size
KV	read only	2-KB reads over 90%
GRP	read only	4-KB:8-KB = 3:2
PRJ	9:1	4-KB and 16-KB reads similarly mixed
USR	7:3	16-KB reads and 4-KB writes
STG	3:7	16-KB reads and 4-KB writes

The five distinct I/O workloads were used for our evaluation. Table II summaries the characteristics of the workloads in terms of a read request size and a read/write ratio. For evaluating how spFTL better handles small reads of HPC applications, we collected traces from two read-only workloads, KV (key-value stores) and GRP (graph processing applications), from db_bench [13] and LinkBench [14], respectively. To evaluate the performance impact of SPREAD in more general applications, we used three traces from MSR-Cambridge traces [15]. In USR and STG, reads and writes were issued in a mixed manner but, in PRJ, reads were dominant.

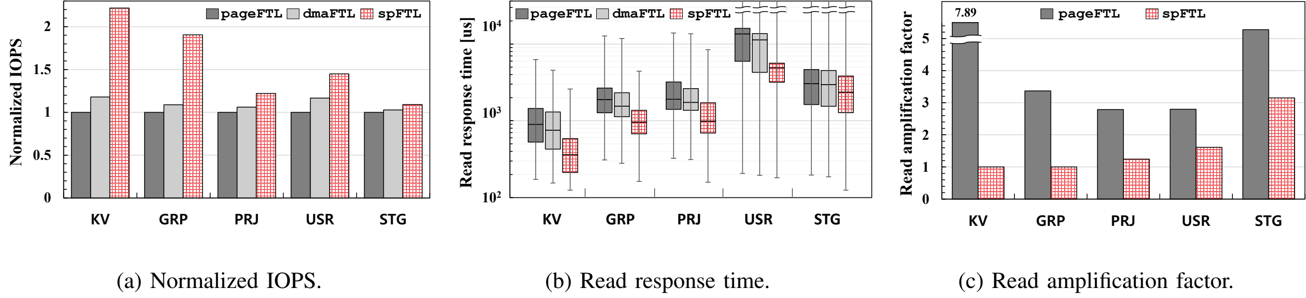
We compared our spFTL with two different FTLs, pageFTL and dmaFTL. PageFTL is a baseline page-level mapping FTL with the FGM scheme. DmaFTL only used the dynamic DMA to reduce t_{DMA} for small reads, but t_R remained the same as in pageFTL since it had to read the entire 16-KB page from NAND chips. SpFTL can shorten t_{DMA} and t_R, as explained in Section III-B.

B. Evaluation Results

In order to compare the performance gains of spFTL over the other FTLs, we measured IOPS values and read latencies for each FTL. As shown in Figure 6(a), spFTL improved IOPS by up to 2.2x and up to 1.9x over pageFTL and dmaFTL, respectively. As expected, this benefit mostly came from the reduction of unnecessary data reads and data transfers for small reads. As shown in Figure 6(b), spFTL reduced the average read latency up to 56% and 50% over pageFTL and dmaFTL, respectively. DmaFTL also exhibited higher I/O performance than that of pageFTL, but the improvements gains were far limited (at most 18%) over spFTL. This was because, with the high speed 0.8 Gb/s I/O bus, t_{DMA} accounted for an insignificant proportion of the total read latency.

One notable observation in our experiments was that spFTL could improve the overall I/O performance even under a workload with many writes. It was an unexpected benefit since spFTL was not optimized for improving write performance at all. Assuming that a read/write ratio of a workload is 7:3, the maximum performance gain could not be higher than 27% because the write latency was about 6.6x longer than the read latency. However, as shown in Figure 6(a), spFTL achieved performance gains over pageFTL by 44% and 10% under USR and STG, respectively, despite their write ratios. As will be explained below, this was due to the negative effect of internal fragmentation.

Authorized licensed use limited to: POSTECH Library. Downloaded on March 24,2023 at 06:23:58 UTC from IEEE Xplore. Restrictions apply.



(a) Normalized IOPS. (b) Read response time. (c) Read amplification factor.

Fig. 6: Performance comparisons of different FTLs under five workloads.

To better understand how spFTL outperformed the other FTLs in detail, we measured RAF values in spFTL and pageFTL as shown in Figure 6(c). The RAF values of spFTL in small read dominant workloads (KV, GRP, and PRJ) were very closed to 1. This told us that the minimum unit size of the SPREAD operation (i.e., 2 KB) was small enough to eliminate read amplification for small reads. Our results showed that, even under 16-KB reads dominant workloads (USR and STG), the RAF values of pageFTL were significantly higher (more than that in PRJ) than those in spFTL. This was because small writes under the FGM scheme created serious internal fragmentation within NAND pages, which greatly increased the number of amplified reads.

Next, we investigated the impact of SPREAD on reducing GC overheads. For USR and STG, we measured the average elapsed time per GC. Some might expect that, since page writes take much longer than page reads, the GC execution time would be dominated by writing valid data, and thus, the benefit of using SPREAD would be trivial. However, our experimental results revealed that, in USR and STG, spFTL reduced the GC execution times over pageFTL by 13% and by 7% on average, respectively. This is because NAND pages were highly fragmented due to many small writes, so most of them held valid and invalid logical pages within the same physcial page. Therefore, in pageFTL, the number of logical pages moved (or written) to free locations can be decreased, while the number of (amplified) page reads remains the same. Consequently, overheads caused by amplified reads accounted for a nontrivial proportion of the total GC I/Os.

VI. CONCLUSIONS

We have presented a new system-level solution that significantly mitigates the amplified read problem on large-page NAND devices. In order to improve the size proportionality of small reads, we proposed a new NAND read operation, SPREAD, which allows selected multiple subpages to be read in parallel. By slightly extending the existing data path and control of the NAND read operation, SPREAD supports most reads without read amplification. In order to take advantages of SPREAD-enabled NAND devices at the storage level, we have developed an SPREAD-aware FTL, spFTL. Our experiment results show that spFTL can increase the IOPS and read latency by up to 122% and 56%, respectively.

Our work in this paper can be extended in several directions. For example, in this paper, we have not considered the large-page problem from a write operation's perspective. It will be an interesting extension to combine the existing subpage write scheme (e.g., [5] [16]) with our SPREAD-based technique.

ACKNOWLEDGEMENTS

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-11.

REFERENCES

[1] D. Kang *et al.*, "256gb 3b/cell v-nand flash memory with 48 stacked wl layers," in *Proceedings of IEEE International Solid-State Circuits Conference (ISSCC)*, 2016.
[2] A. L. Shimpi. (2014) Micron announces 16nm 128gb mlc nand, ssds in 2014. [Online]. Available: https://www.anandtech.com/show/7147/micron-announces-16nm-128gb-mlc-nand-ssds-in-2014
[3] B. Atikoglu *et al.*, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, 2012, pp. 55–64.
[4] H. Liu and H. H. Huang, "Graphene: Fine-grained io management for graph computing," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2017.
[5] M. Kim *et al.*, "Improving performance and lifetime of large-page nand storages using erase-free subpage programming," in *Proceedings of Design Automation Conference (DAC)*, 2017.
[6] L. M. Grupp *et al.*, "The bleak future of nand flash memory," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012.
[7] J. Axboe, "Fio-flexible i/o tester synthetic benchmark," 2005. [Online]. Available: https://github.com/axboe/fio
[8] K. Zhao *et al.*, "Ldpc-in-ssd: Making advanced error correction codes work effectively in solid state drives," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2013.
[9] JEDS230, "Nand flash interface interoperability," JEDEC Solid State Technology Association, Tech. Rep., 2014.
[10] R. Micheloni *et al.*, *Inside NAND Flash Memories*. Springer, 2010.
[11] J. E. Brewer *et al.*, *Nonvolatile Memory Technologies with Emphasis on Flash*. IEEE Press, 2008.
[12] S. Lee *et al.*, "Application-managed flash," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2016.
[13] Facebook. (2013) Rocksdb git repository. [Online]. Available: https://github.com/facebook/rocksdb
[14] T. G. Armstrong *et al.*, "Linkbench: a database benchmark based on the facebook social graph," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.
[15] D. Narayanan *et al.*, "Write off-loading: Practical power management for enterprise storage," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2008.
[16] X. Zhang *et al.*, "Reducing solid-state storage device write stress through opportunistic in-place delta compression," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2017.

Authorized licensed use limited to: POSTECH Library. Downloaded on March 24,2023 at 06:23:58 UTC from IEEE Xplore. Restrictions apply.