Improving Performance and Lifetime of NAND Storage Systems Using Relaxed Program Sequence

Jisung Park[†], Jaeyong Jeong[†], Sungjin Lee[‡], Youngsun Song[†], and Jihong Kim[†]
[†]Department of Computer Science and Engineering, Seoul National University
[‡]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology
[†]{jspark, jyjeong, ysunsong, jihong}@davinci.snu.ac.kr, [‡]chamdoo@csail.mit.edu

ABSTRACT

We propose a new system-level solution that improves both the performance and lifetime of NAND storage systems by exploiting the performance asymmetry of NAND devices. At the device level, we propose a new program sequence, called relaxed program sequence (RPS), which allows more flexible page allocations in a block without compromising NAND reliability. By combining RPS with per-block parity pages, we can improve the write bandwidth and eliminate expensive paired page backup operations. Experimental results show that the proposed technique can increase IOPS by up to 56% and reduce the number of block erasures by up to 30% over an existing RPS-oblivious FTL.

1. INTRODUCTION

A multi-leveling technique, which allows a single NAND memory cell to store multiple bits, has been widely used for recent NAND flash memory as a key enabling technique for rapidly decreasing the cost-per-bit of single-level cell (SLC) NAND flash memory. Although the multi-leveling technique is effective in increasing the capacity of a NAND device, it also negatively affects the performance and lifetime of NAND storage systems because of a fine-grained charge placement and sensing mechanism used in the multi-leveling technique.

Figure 1 illustrates a typical program scheme for 2-bit multi-level cell (MLC) NAND devices under the fine-grained charge placement and sensing mechanism. (We use 2-bit MLC NAND devices as examples when specific multi-level NAND devices are needed, but our proposed technique can be applicable for other NAND devices such as triple-level cell (TLC) NAND devices [1] with a similar program scheme.) As shown in Figure 1, when the first bit (i.e., the least significant bit (LSB)) is programmed, the NAND flash controller quickly forms a threshold voltage (V_{th}) distribution because it is required to form only two V_{th} distributions which are separated by a large voltage margin. On the other hand, when the second bit (i.e., the most significant bit (MSB)) is programmed, the NAND flash controller takes more times because it needs to represent one of four V_{th} distributions within the same V_{th} window in a finer-grained fashion. Therefore, when an MSB page is programmed, a NAND storage system experiences a significant latency in-

DAC '16, June 05-09, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4236-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2897937.2898032

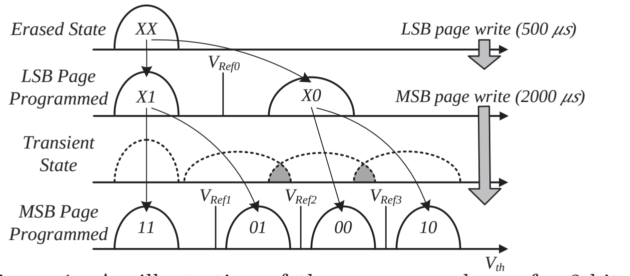


Figure 1: An illustration of the program scheme for 2-bit MLC devices.

crease over when an LSB page is programmed. For example, in recent 2X-nm MLC NAND devices, the program latency of an MSB page is about four times longer than that of an LSB page (i.e., 2,000 μs vs. 500 μs) [2].

The performance asymmetry between the LSB page and MSB page at the NAND device level can be further amplified at the storage controller level. When an MSB page is written, its paired LSB page which shares memory cells with the MSB page must be saved to a different NAND page in order to ensure the data durability. This is because the MSB-page program is intrinsically a *destructive* process. As shown in Figure 1, during the MSB-page program, the LSB-programmed V_{th} states are gradually rearranged so that the stored LSB data are temporarily destructed. Without a paired LSB-page backup, if a sudden power-off occurs during an MSB page program, the valid data of the paired LSB page previously stored may get lost because there is no way of retrieving the LSB-page data [3]. Since a paired LSB-page backup operation requires a page copy operation, the program latency of an MSB page can be further increased. In 2X-nm MLC NAND devices, for example, the effective program latency of an MSB page can be five times longer than that of an LSB page. Furthermore, since extra page writes are required for paired LSB-page backup operations, the lifetime of MLC NAND devices can be deteriorated as well; three page writes are necessary for storing two pages.

Although some existing studies [4, 5, 6] attempted to alleviate the performance and lifetime degradation of MLC NAND storage systems by exploiting the performance asymmetry of MLC NAND pages, an amount of improvement by these techniques is rather limited because of the underlying page program order constraint within an MLC NAND block. For example, a common program sequence (called as fixed program sequence (FPS)) specifies a linear page program order for the pages in the same block. Because of this strict program order, an upper management layer such as a flash translation layer (FTL) has little room to exploit the underlying device heterogeneity between different page types, thus limiting the effectiveness of the existing techniques. For example, when burst writes are requested in a short time interval, more LSB-page writes would be preferred for satisfying a high peak performance requirement.

On the other hand, when write requests come in a sporadic fashion, slow MSB-page writes may be sufficient. However, such flexible page selections within the same block are not possible under the FPS scheme.

In this paper, we present a new system-level solution that improves both the performance and lifetime of MLC NAND storage systems by *fully* exploiting the performance asymmetry of MLC NAND devices. Our key insight is that the FPS scheme is an *over-specification* which unnecessarily restricts orderings between LSB pages and MSB pages. We propose a new device-level program sequence, called relaxed program sequence (RPS), which removes an unnecessary program constraint of the existing program sequence, thus allowing LSB-page writes and MSB-page writes to be mixed in a more flexible fashion. From experimental evaluations using 2X-nm MLC NAND devices, we validated that our proposed RPS scheme does not compromise the NAND reliability requirement over the existing FPS scheme.

Under the proposed RPS scheme, since an upper management layer can choose the page type in a flexible fashion, new optimizations are possible at the FTL level. We propose two such optimizations in this paper, a write bandwidth optimization technique for write-intensive workloads and an overhead minimization technique for paired page backups.

In order to evaluate the effectiveness of the proposed optimization techniques, we have developed an RPS-aware FTL, called `flexFTL`, which we have implemented as a host-level FTL for a custom NAND flash board [7]. In `flexFTL`, NAND pages are programmed under the RPS scheme, not the FPS scheme. Our experimental results show that significant gains are possible with `flexFTL`, in terms of both the performance and lifetime over an existing FTL. By exploiting the performance asymmetry, `flexFTL` can increase IOPS (Input/Out-put Operations Per Second) by up to 56% over the baseline FTL (which is performance asymmetry-oblivious). Since `flexFTL` avoids most of paired LSB-page backup overhead, it also significantly improved the lifetime of an MLC NAND storage system. Our evaluation results show that the number of block erasures is reduced by up to 30% over the baseline FTL.

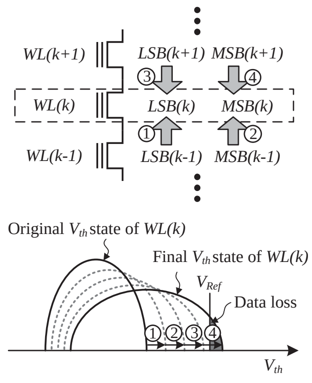
The rest of the paper is organized as follows. The proposed RPS scheme is described in Section 2. In Section 3, we present the proposed RPS-aware FTL, `flexFTL`. Experimental results follow in Section 4, and related work is summarized in Section 5. Section 6 concludes with a summary and future work.

2. RPS: RELAXED PROGRAM SEQUENCE

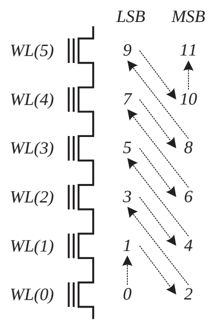
2.1 Fixed Program Sequence Schemes

Most MLC NAND devices require that pages in the same block are written following a fixed program order specified by NAND device manufacturers. The main goal of a program sequence scheme is to minimize the cell-to-cell interference which is the side effect of NAND program operations so that the operation margin in the V_{th} window can be secured [8]. Since the NAND operation margin gets reduced with shrinking semiconductor processes, minimizing the negative impact of the cell-to-cell interference has been one of the most critical technical issues at the device level.

The cell-to-cell interference is a phenomenon that a programmed page is additionally programmed by program operations to its immediately neighboring pages. When this interference is strong, the V_{th} states of the cells in the programmed page can be shifted to the right, thus causing unexpected changes in the stored data. Figure 2(a) illustrates the worst-case example of the cell-to-cell interference prob-



(a) An illustration of the cell-to-cell interference problem.



(b) A representative FPS scheme [8].

Figure 2: The FPS scheme for minimizing the cell-to-cell interference problem in MLC NAND devices.

lem when MLC pages in a block can be written without any *restriction* on the page program order. (We denote the k-th word line in a block by *WL(k)*, and the MSB page and LSB page of *WL(k)* by *MSB(k)* and *LSB(k)*, respectively.) For *WL(k)* whose LSB page and MSB page were all written, the cell-to-cell interference to *WL(k)* can be maximized when its four neighboring pages, *LSB(k-1)*, *MSB(k-1)*, *LSB(k+1)*, and *MSB(k+1)* are successively written. In such a case, the original V_{th} state may be shifted to the far right (e.g., ④ in Figure 2(a)) so that its V_{th} state can be misinterpreted, thus losing the original data stored in the cell.

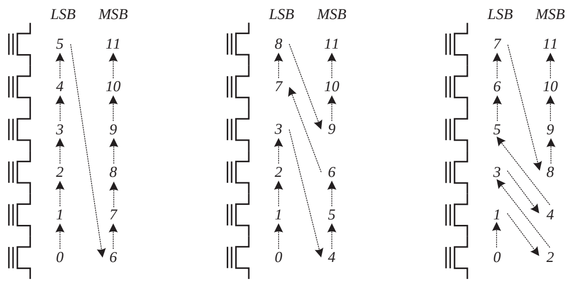
In order to minimize the cell-to-cell interference problem, the FPS scheme was proposed. Since the total sum of the cell-to-cell interference on *WL(k)* is directly proportional to the number of aggressor program operations (i.e., program operations performed for *WL(k-1)* and *WL(k+1)* after *MSB(k)* is written), the existing FPS scheme limits the number of aggressor program operations by fixing a program sequence for pages in a block. Figure 2(b) shows a representative FPS scheme which is commonly employed in recent MLC NAND devices [8]. As shown in Figure 2(b), only one aggressor program operation, writing to *MSB(k+1)*, can affect the V_{th} state of *WL(k)* after *MSB(k)* is written.

2.2 Relaxed Program Sequence Schemes

In order to explore the possibility of more flexible page program orders, we formalized the FPS scheme of Figure 2(b) using its four constraints on the page program order as summarized below:

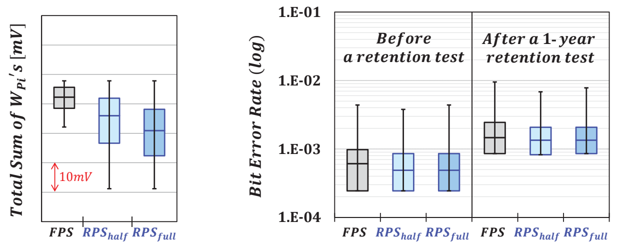
- *Constraints 1* & *2*: Before *LSB(k)* (or *MSB(k)*) is written, *LSB(k-1)* (or *MSB(k-1)*) should be written (where $k \geq 1$).
- *Constraint 3*: Before *MSB(k)* is written, *LSB(k+1)* should be written (where $k \geq 0$).
- *Constraint 4*: Before *LSB(k)* is written, *MSB(k-2)* should be written (where $k \geq 2$).

Constraints 1 and *2* specify the program orders between the same type of pages. When pages in a block are written following these constraints, since *LSB(k-1)* and *MSB(k-1)* do not affect *MSB(k)*, the total sum of the cell-to-cell interference for *WL(k)* can be reduced by 50%. On the other hand, *Constraints 3* and *4* specify the program orders between the different type of pages. *Constraint 3* contributes to reducing the cell-to-cell interference for *MSB(k)* by removing the interference from *LSB(k+1)*. However, *Constraint 4* is an over-specified constraint because writing to *WL(k-2)* does not interfere with *WL(k)*. In other words, *Constraint 4* can be removed without affecting the cell-to-cell interference among MLC NAND pages. When a program sequence



(a) RPS$_{full}$ order. (b) RPS$_{half}$ order. (c) A random order.

Figure 3: Examples of three different program orders under the RPS scheme.



(a) Measured distributions of W_{Pi}'s. (b) Measured distributions of bit error rates.

Figure 4: Reliability comparison results for the FPS scheme and RPS schemes under the worst-case condition.

scheme satisfies only the first three constrains, *Constraints 1, 2,* and *3*, we call it a relaxed program sequence scheme.

By removing *Constraint 4*, the RPS scheme allows very flexible program orders between LSB pages and MSB pages. For example, as shown in Figure 3(a), all the LSB pages in a block can be sequentially written before their paired MSB pages. Figures 3(b) and 3(c) show other examples of possible program orders where LSB-page writes and MSB-page writes are flexibly intermixed.

In order to validate that the proposed RPS scheme can guarantee the same level of the NAND reliability over the FPS scheme, we compared the overall impact of the cell-to-cell interference under different schemes. As a main evaluation metric, we measured the width W_{Pi} of the V_{th} distribution for each V_{th} state because W_{Pi} reflects the overall impact of the cell-to-cell interference quantitatively. Our verifications were performed with more than 90 blocks out of three 2X-nm MLC NAND chips. Since there is a high flexibility in selecting the program order with the RPS scheme, we tested two typical program orders, RPS$_{full}$ and RPS$_{half}$, as shown in Figures 3(a) and 3(b), respectively. Figure 4(a) shows the measured distributions of the total sum of W_{Pi}'s for more than 5,000 pages out of 90 blocks using box plots. As we expected, W_{Pi}'s under RPS$_{full}$ and RPS$_{half}$ were not increased over the FPS scheme, thus showing that the overall cell-to-cell interference with the RPS scheme was not higher than that with the FPS scheme. In order to compare the overall NAND reliability under the RPS scheme over the FPS scheme, we further measured the bit error rate of tested pages under the worst-case operating conditions (i.e., 3K P/E cycles and 1-year retention time) of MLC NAND devices. As shown in Figure 4(b), the bit error rate for the RPS scheme was not higher than that for the FPS scheme under the worst-case operating conditions. Based on our verification results, we concluded that the proposed RPS scheme can be employed instead of the existing FPS scheme without affecting the overall NAND reliability.

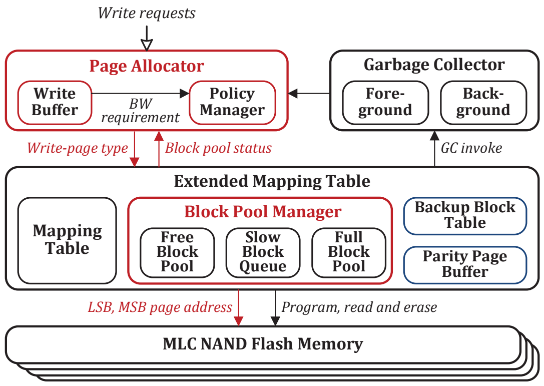


Figure 5: An organizational overview of `flexFTL`.

3. FLEXFTL: RPS-AWARE FTL

Since the RPS scheme allows more flexible orderings between LSB-page writes and MSB-page writes, several new optimizations are feasible at the FTL level. In `flexFTL`, we employ two RPS-enabled optimizations. First, under the RPS scheme, the write bandwidth can be more easily increased when such accelerations are necessary. Since fast LSB-page writes can be consecutively performed without an intervening slow MSB-page write, the peak write bandwidth can be improved up to the write bandwidth of SLC NAND devices. Second, under the RPS scheme, a large number of LSB pages on the same block can be successively written in a row, making the parity-based backup scheme very efficient (as described in Section 3.3). For a typical MLC NAND block with 256 pages, only a single parity page write is required if all the LSB pages of the block are written before an MSB page of the block, thus significantly reducing the overhead of paired LSB-page backup operations.

Figure 5 shows an overall organization of `flexFTL`. `FlexFTL` is based on an existing page-level mapping FTL with additional modules, the page allocator and the block pool manager, for supporting RPS-enabled two optimizations. By considering the amount of free MSB pages and free LSB pages under the current I/O workload characteristics, the page allocator chooses an appropriate page type for a given request. The block pool manager is in charge of managing NAND blocks in a performance asymmetry-aware fashion under the RPS scheme. When the block pool manager detects that the number of available LSB pages is not sufficient for future write requests, it invokes a background garbage collector for reclaiming free LSB pages while consuming slow MSB pages during background garbage collections. Furthermore, in order to balance the amount of free LSB pages and MSB pages for future requests, the block pool manager provides the block pool state to the page allocator so as to control the consumption of each page type.

3.1 Two-Phase Block Management

Since `flexFTL` is based on the RPS scheme, it has a large freedom in choosing a page program order; if needed, for example, it may even change page program orders during run time. However, such high flexibility in page program orders may be too expensive to implement in practice. Therefore, in `flexFTL`, we choose a particular page program order (which is an instance of the RPS scheme), called as two-phase ordering (2PO). Under the 2PO scheme, all the LSB pages of a block are first written followed by all the MSB pages of the block (i.e., RPS$_{full}$ in Figure 3(a)). When the 2PO scheme is used, a block can be viewed as cycling through four distinct states as shown in Figure 6. Starting

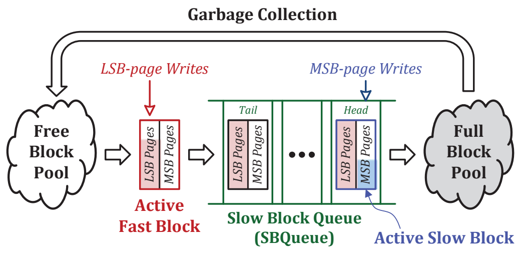


Figure 6: A life cycle of a NAND block in `flexFTL`.

from a *free* block, it remains as a *fast* block as long as it has a free LSB page. Once all the LSB pages of a *fast* block are written, it becomes a *slow* block. When all the MSB pages of a *slow* block are written, it becomes a *full* block. A garbage collector returns a *full* block to a *free* block again.

One of the main functions of the block pool manager is to keep track of block state changes. The block pool manager maintains a single active fast block per chip in order to support fast LSB-page writes. Once that active fast block runs out of LSB pages, it is added to the slow block queue (SBQueue) while a new active fast block is selected from the free block pool. The block pool manager also maintains a single active slow block per chip (from the SBQueue) for serving MSB-page writes. Since we manage the SBQueue in a FIFO fashion, the head block of the SBQueue automatically becomes the active slow block. By employing two active blocks, one from fast blocks and the other from slow blocks, `flexFTL` can adapt very flexibly to varying write workloads without introducing any extra backup operation as long as the per-block parity page is properly maintained.

3.2 Adaptive Page Allocation

In order to better meet varying write-bandwidth requirements, the policy manager (within the page allocator) selects the most appropriate page type for each page write. As a general guideline, the policy manager prefers a fast LSB-page write when two conditions are satisfied: [C1] a high write bandwidth is required and [C2] an LSB-page write does not significantly degrade a future write bandwidth. In order to implement this guideline, the policy manager monitors two parameters, the write buffer utilization u (for C1) and the quota q for successive LSB-page writes (for C2).

When u is high, the policy manager estimates that a high write bandwidth is required. On the other hand, when u is low, it predicts that no high write bandwidth is necessary.

The quota for successive LSB-page writes is used to estimate how a future write bandwidth is affected by the current LSB-page write. When q is large, we interpret that more LSB-page writes would not hurt a future write bandwidth. However, when q is small or zero, we understand that some additional LSB-page writes may hurt a future write bandwidth. The quota q is initially set to the maximum size of successive LSB-page writes that a storage system may need to support. Since this size may not be known in advance, we conservatively choose a large value. (In the current `flexFTL`, the initial value of q is set to 5% of the total number of LSB pages.) During run time, each LSB-page write decrements q by one while each MSB-page write increments q by one.

Based on the current u and q values, the policy manager chooses the right page type as follows. For the given two utilization threshold values u_{high} and u_{low} (where $u_{high} > u_{low}$), when u is higher than u_{high}, the policy manager checks if $q > 0$. If $q > 0$, the policy manager chooses an LSB-page write. Otherwise, the policy manager chooses LSB pages and MSB pages in an alternate fashion. On the other hand, when $u < u_{low}$, the policy manager chooses an MSB-page write[1]. When $u_{low} \leq u \leq u_{high}$, the policy manager alternately selects LSB pages and MSB pages.

The main role of q is to avoid large performance fluctuations in `flexFTL` which may occur when successive LSB-page writes are allowed without any restriction. For example, if there were no limit on the size of consecutive LSB-page writes, all the free LSB pages may be consumed by a large write-intensive workload. Once no free LSB page is available, a future write request must be serviced with slow MSB pages only, thus significantly lowering the write bandwidth achieved. Using q avoids such a dramatic drop in the write bandwidth by restricting the size of consecutive LSB-page writes, thus forcing to use MSB pages when $q \leq 0$. Once the quota q is expended, `flexFTL` works in a similar fashion as an FPS-based FTL by alternating LSB-page writes and MSB-page writes. If we can maintain q large enough to service most write requests with high peak write bandwidth requirements, it is possible to support occasional high peak write bandwidth with small performance fluctuations.

Since the higher q, the higher write bandwidth, `flexFTL` tries to keep q at a high value range by intelligently invoking a background garbage collector during idle times. In idle times, if the number of free blocks is less than a threshold (e.g., 10% of the total capacity), the block pool manager invokes the background garbage collector in order to reclaim free LSB pages for future write requests. Once the background garbage collector is invoked, it chooses a *victim* block with the largest number of invalid pages. Since the background garbage collector is invoked during idle times, the valid pages of the victim block are copied using MSB pages, thus increasing q while free LSB pages are reclaimed. A higher q value after a background garbage collection enables a future write request to be served with fast LSB pages.

3.3 Per-Block Parity Page-Based Backup

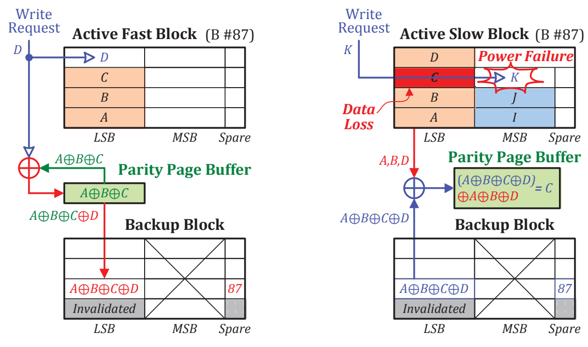
`FlexFTL` efficiently reduces the paired LSB-page backup overhead by leveraging the 2PO scheme and the parity backup scheme [6]. Figure 7(a) illustrates how the backup procedure works in `flexFTL`. The backup procedure is closely connected to the proposed 2PO scheme. While the LSB pages of the active fast block are written, using the parity page buffer, `flexFTL` computes the accumulated parity values of all the LSB pages written in the active fast block. For example, in Figure 7(a), the parity page buffer contains the accumulated parity page of three LSB pages written, A, B, and C. When the last LSB page of the active fast block is written (i.e., D is written), our backup procedure stores the accumulated parity page to the reserved backup block[2]. When the accumulated parity page is written to the backup block, we also store the block number (e.g., 87) of the corresponding active fast block to the spare area of the parity page. (This inverse mapping of a backup page to a block is necessary to safely recover from a sudden power-off.) Once the pages of a slow block are all written, the saved backup page is invalidated because we do not need it any more.[3]

The error recovery procedure takes reverse steps of the backup procedure as shown in Figure 7(b). When a sudden power-off occurs during an MSB-page write (e.g., during writing the page K to the paired MSB page of the page C in block #87), `flexFTL` checks, at the time of a reboot, if there was any data loss in the active slow block. In order

[1]As a corner case, if there is no slow block, an LSB page is selected.

[2]In order to reduce the backup overhead, parity pages are written to the LSB pages of the backup block.

[3]The overhead of computing parity values is insignificant over the NAND program time.



(a) An example of the parity backup procedure.

(b) An example of the recovery procedure.

Figure 7: Examples of backup and recovery procedures.

to check a data loss in the active slow block, we read all the LSB pages of the active slow block while recomputing the accumulated parity values on the parity page buffer. If all the LSB pages were successfully read, we have recomputed the parity page for the active slow block and the error recovery procedure ends without further actions for the active slow block. If some page lost its data during a sudden power-off (e.g., C), we cannot read the page because of ECC-uncorrectable errors in the page. In this case, we skip the lost page in computing the accumulated parity value. However, we continue to read the rest of LSB pages so that the accumulated parity values can be computed. By comparing the saved parity page (in the backup block) and the recomputed parity values, we can recover the lost page.

The per-block parity page backup scheme may increase the reboot time after a sudden power-off because the error recovery procedure must recompute parity values for all the active slow blocks. Furthermore, partially accumulated parity values for all the active fast blocks should be recomputed during the reboot time. Although a large number of LSB pages should be read during the reboot time after a sudden power-off, the overhead of the error recovery procedure is relatively insignificant because these extra reads occur during the *reboot* time. For example, when a storage system has 16 NAND chips and each block has 64 LSB pages, the extra read overhead is less than 100 ms (i.e., $16\ [chips] \times 2\ [\frac{blocks}{chip}] \times 64\ [\frac{pages}{block}] \times 40\ [\frac{\mu s}{page}] = 81.92$ ms). This read overhead may be acceptable for most cases because a total reboot time may take a few seconds to several tens of seconds.

4. EXPERIMENTAL RESULTS

4.1 Experimental Settings

To evaluate the effectiveness of the proposed `flexFTL`, we implemented `flexFTL` as a host-level FTL for a custom-built MLC NAND board from BlueDBM [7]. In our experiments, we limited the storage capacity of the BlueDBM (which can support up to 512 GB of a flash storage system) to 16 GB for fast evaluations. The 16-GB custom storage system consists 8 channels and each channel has 4 NAND chips. Each chip has 512 blocks and each block has 256 4-KB pages.

For our evaluations, we used five distinct I/O workloads, which were generated from Sysbench [10] and Filebench [11]. As summarized in Table 1, five benchmarks represent different I/O characteristics of various enterprise applications with different I/O intensiveness and read/write combinations. OLTP and NTRX, which were generated from Sysbench, represent intensive DB workloads with little idle times between successive I/O requests. Webserver, Varmail and Fileserver, on the other hand, were generated from Filebench. Webserver, a read dominant workload with large idle times, represents the I/O activities of a simple web server. Varmail and Fileserver emulate a mail server and a file server, respectively. Both represent write-intensive workloads with a fair amount of idle times.

Table 1: I/O characteristics of five benchmark workloads.

	OLTP	NTRX	Webserver	Varmail	Fileserver
Read:Write	7:3	3:7	4:1	1:1	1:2
I/O intensiveness	Very high	Very high	Moderate	High	High

We compared our `flexFTL` with three different FPS-based FTLs: `pageFTL`, `parityFTL`, and `rtfFTL`. `PageFTL` is a baseline page-level mapping FTL based on the FPS scheme under no sudden power-off assumption. Since `pageFTL` does not need paired page backups, it is used to indicate the maximum performance level of a page-level FTL under the FPS scheme. `ParityFTL`, which employs an advanced paired page backup technique of [6], maximally exploits the parity page backup scheme under the FPS scheme while taking into account of the inter-channel parallelism of NAND storage systems. In order to minimize the backup overhead, `parityFTL` pre-backups a single parity page for two LSB pages[4]. `RtfFTL`, which is based on the *return-to-fast* scheme proposed in [5], performs successive LSB-page writes for incoming write requests under the FPS scheme, by maintaining multiple active blocks per chip. A background garbage collector aggressively consumes paired MSB pages in idle times so as to sustain successive LSB-page writes. In our `rtfFTL` implementation, eight active blocks are used for each chip, thus supporting the maximum 256 successive LSB-page writes with 32 chips. In our evaluations, we set u_{high} and u_{low} to 80% and 10%, respectively. The initial value of q is set to 5% of total LSB pages. When the number of free blocks is less than 10% of the total capacity, a background garbage collector is invoked during storage idle times in all four FTLs we evaluated.

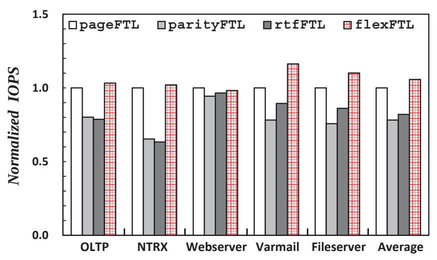
4.2 Evaluation Results

In order to compare the performance and lifetime gains of `flexFTL` over the other FTLs, we measured IOPS values and block erasure counts for each FTL. Figure 8(a) shows normalized IOPS values of four different FTLs under each workload. As shown in Figure 8(a), `flexFTL` outperforms `pageFTL`, `parityFTL`, and `rtfFTL` by up to 16% (5% on average), 56% (35% on average), and 61% (29% on average), respectively. In particular, `flexFTL` achieves higher IOPS's even over `pageFTL` except for Webserver (which is read dominant). For Varmail and Fileserver, `flexFTL` was the most effective in serving long successive LSB-page writes. On the other hand, since the background garbage collector cannot increase q due to little idle times in OLTP and NTRX, `flexFTL` achieved a similar IOPS level as `pageFTL`. However, `flexFTL` shows large performance gains over `parityFTL` and `rtfFTL` for OLTP and NTRX, because the paired page-backup overhead affects the effective performance more significantly under more intensive workloads.

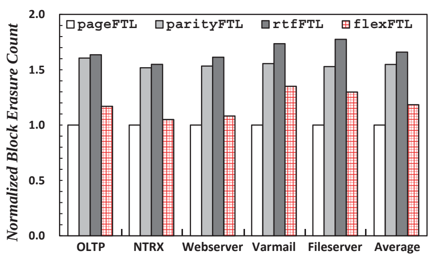
Figure 8(b) shows that `flexFTL` also reduces block erasures by up to 30% (23% on average) and 32% (28% on average) over `parityFTL` and `rtfFTL`, respectively. This is mainly because of the per-block parity scheme used in `flexFTL` which becomes feasible under the 2PO scheme. On the other hand, `parityFTL` and `rtfFTL` consume more free pages under the same workload for backup operations.

In order to understand how `flexFTL` can better meet

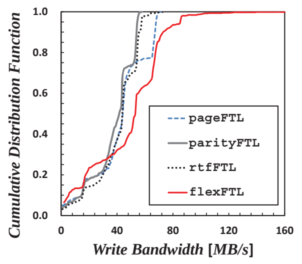
[4]As shown in Figure 2(b), at most two LSB pages can share a parity backup page before programming their paired MSB pages.



(a) Comparisons of normalized IOPS's for five I/O workloads.



(b) Comparisons of normalized block erasure counts for five I/O workloads.



(c) CDF curves of write bandwidth for Varmail.

Figure 8: Comparisons of the performance and lifetime under different techniques.

high write-bandwidth requirements over the other FTLs, we ploted the cumulative distribution function (CDF) curves of write bandwidth for Varmail. As shown in Figure 8(c), the peak write bandwidth of `flexFTL` is about 2.13 times higher than that of `rtfFTL` (which has the highest peak write bandwidth among the FTLs compared to `flexFTL`). Overall, `flexFTL` achieves 24% and 17% higher write bandwidth, on average, over `parityFTL` and `rtfFTL`, respectively.

5. RELATED WORK

There have been several previous studies which exploited the performance asymmetry of MLC NAND devices for improving the performance of MLC NAND storage systems. Lee *et al.* [4] proposed a new flash file system which can, when required, service write requests using only fast LSB-page writes. Although this technique may achieve the peak I/O performance close to SLC flash memory, all the MSB pages of a block are skipped when fast LSB-page writes are used, thus wasting half the capacity of the block. `FlexFTL`, however, does not sacrifice the NAND capacity when a high write bandwidth is necessary because the RPS scheme allows successive LSB-page writes without skipping MSB pages. Grupp *et al.* [5] proposed another technique for serving successive LSB-page writes by maintaining a pool of free LSB pages from multiple active blocks. However, the maximum size of the LSB-page pool is limited by a small number of active blocks per chip (e.g., 8 successive LSB-page writes per chip). Furthermore, because of the FPS scheme, once all the LSB pages in the LSB-page pool are used, the paired MSB pages (from the LSB-page pool) should be consumed by background garbage collections so that subsequent LSB-page writes can be serviced. On the other hand, `flexFTL` can support much longer successive LSB-page writes as long as the quota q is maintained in a positive value range.

In order to reduce the paired page backup overhead, Lee *et al.* [6] proposed an adaptive pre-backup scheme which maximally exploits the parallelism of NAND devices. Although this technique can reduce backup operations by up to 50% with a parity backup page, its potential benefit is severely limited because the maximum two LSB pages can share a parity page under the FPS scheme, while `flexFTL` allows all the LSB pages (e.g., 128 pages) of a block to share a single parity page.

6. CONCLUSIONS

We have presented a new system-level solution that efficiently exploits the performance asymmetry on MLC NAND devices in a holistic fashion. Based on a simple but effective new RPS scheme proposed at the NAND device level, we described several novel NAND flash management techniques at the FTL level for fully exploiting the device-level performance asymmetry. The 2PO scheme combined with the per-block parity scheme enables an FTL to achieve higher write bandwidth while removing most overhead of paired page backup operations, thus improving both the performance and lifetime of MLC NAND storage systems. We also proposed an adaptive page allocation policy which chooses appropriate page types under varying write bandwidth requirements. Our experimental results show that `flexFTL` can increase the performance by up to 56% while improving the lifetime by up to 30% over an existing FPS-based FTL.

The current version of `flexFTL` can be further improved in several directions. For example, the page allocator is using a rather simple heuristic for choosing a page type for given requests. If `flexFTL` can more accurately estimate the amount of future writes, for example, by using a page cache-based future write predictor [9], a background garbage collector can reclaim free blocks more efficiently so that more LSB-page writes can be used for future write requests.

ACKNOWLEDGMENTS

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2013R1A2A2-A01068260), and the Next-Generation Information Computing Development Program through the NRF funded by the MSIP (NRF-2015M3C4A7065645). The ICT at Seoul National University and IDEC provided research facilities for this study.

REFERENCES

[1] G. Naso *et al.* A 128Gb 3b/Cell NAND Flash Design Using 20nm Planar-Cell Technology. In *Proc. IEEE Int. Solid-State Circuits Conf.*, 2013.

[2] C. Kim *et al.* A 21 nm High Performance 64 Gb MLC NAND Flash Memory with 400 MB/s Asynchronous Toggle DDR Interface. *IEEE J. Solid-State Circuits*, 47(4):981–989, 2012.

[3] H.-W. Tseng *et al.* Understanding the Impact of Power Loss on Flash Memory. In *Proc. Design Automation Conf.*, 2011.

[4] S. Lee *et al.* Improving Performance and Capacity of Flash Storage Devices by Exploiting Heterogeneity of MLC Flash Memory. *IEEE Trans. Comput.*, 63(10):2445–2458, 2014.

[5] L. M. Grupp *et al.* The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs. In *Proc. USENIX Annu. Tech. Conf.*, 2013.

[6] J. Lee *et al.* Adaptive Paired Page Prebackup Scheme for MLC NAND Flash Memory. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 33(7):1110–1114, 2014.

[7] S.-W. Jun *et al.* BlueDBM: An Appliance for Big Data Analytics. In *Proc. Int. Symp. Comput. Archit.*, 2015.

[8] K.-T. Park *et al.* A Zeroing Cell-to-Cell Interference Page Architecture with Temporary LSB Storing and Parallel MSB Program Scheme for MLC NAND Flash Memories. *IEEE J. Solid-State Circuits*, 43(4):919–928, 2008.

[9] S. S. Hahn *et al.* To Collect or Not to Collect: Just-in-Time Garbage Collection for High-Performance SSDs with Long Lifetimes. In *Proc. Design Automation Conf.*, 2015.

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

[11] Filebench. http://filebench.sourceforge.net.