공학박사학위논문

대용량 낸드 플래시 저장장치를 위한 성능 및 수명 최적화 기법

Performance and Lifetime Optimizations for Large-Capacity NAND Storage Systems

2019년 8월

서울대학교 대학원
전기·컴퓨터공학부
박지성

대용량 낸드 플래시 저장장치를 위한 성능 및 수명 최적화 기법

Performance and Lifetime Optimizations for Large-Capacity NAND Storage Systems

지도교수 김 지 홍

이 논문을 공학박사 학위논문으로 제출함
2019년 5월

서울대학교 대학원
전기·컴퓨터공학부
박지성

박지성의 공학박사 학위논문을 인준함
2019년 6월

위 원 장 유 승 주 (인)

부위원장 김 지 홍 (인)

위 원 이 재 욱 (인)

위 원 정 명 수 (인)

위 원 이 성 진 (인)

Abstract

In recent years, NAND flash-based storage systems have grown in the popularity owing to the continuous reduction of cost-per-bit value of NAND devices, which is enabled by capacity-oriented design decisions such as semiconductor process scaling, multi-leveling techniques, and large operation units. Although those capacity-oriented design decisions are effective in increasing the NAND density, they have also introduced significant performance and lifetime degradation of NAND devices. On the other hand, in the era of data explosion, modern computing systems have been demanding storage systems to satisfy the high-performance and long-lifetime requirements as well as the large-capacity requirement. Considering that capacity-oriented design decisions are expected to be continuously and more aggressively adopted to meet the large-capacity requirement, it is crucial to develop new optimization techniques that can overcome the limitations of existing techniques.

In this dissertation, we propose various optimization techniques that address the side effects of capacity-oriented design decisions, thus improving performance and lifetime of large-capacity NAND flash-based storage systems. In order to overcome the limitations of existing techniques, our proposed techniques tackle to root causes of performance and lifetime problems, by revisiting common perceptions widely accepted without proper reasoning. By doing so, they allow

NAND flash-based storage systems to satisfy high-performance, long-lifetime, and large-capacity requirements of modern data-centric applications at the same time.

We first introduce a new page-program sequence for multi-level cell (MLC) NAND flash memory, called *Relaxed Program Sequence* (RPS). The RPS scheme removes an unnecessary constraint in the existing page-program sequence, which significantly limits the effectiveness of existing techniques for improving performance and lifetime of MLC NAND flash-based storage systems. It enables the upper management layers to fully exploit heterogeneity of MLC NAND pages without compromising data reliability over the existing scheme. We propose an RPS-aware FTL, `flexFTL`, which leverages two RPS-enabled techniques. It can almost eliminate overheads of page-backup operations required in MLC NAND flash memory for ensuring the data durability, and provide high write-performance that typical MLC NAND flash-based storage systems cannot achieve.

Secondly, we propose a new read operation for large-page NAND flash memory, called *Subpage-Parallel Read* (SPread). By supporting fine-granular reads with short latencies, SPread directly addresses the read amplification problem originated from the I/O unit mismatch between NAND devices and upper system layers. Furthermore, SPread allows the storage firmware to effectively deal with *fragmented* reads generated in storage management tasks, thereby improving overall I/O performance under workloads with many writes, as well as under read-intensive workloads.

Finally, we present an effective integration of existing data-reduction techniques, called *Dedup-Assisted Compression* (DAC). DAC integrates deduplication and lossless compression in a synergistic fashion so that potential benefits of individual ones can be maximized while overcoming their inherent limitations. By effectively reusing the hardware resources and data structures of individual techniques, our proposed technique further reduces write-traffic to underlying NAND devices with negligible overheads over a naive combination of deduplication and lossless compression.

In order to evaluate the effectiveness of the proposed techniques, we conducted a set of experiments with various benchmark tools and I/O traces collected from real-world applications. The experimental results showed that our proposed techniques significantly improve both the performance and lifetime, thus enabling large-capacity NAND storage systems to better meet the high-performance and long-lifetime requirements of modern computing systems.

Keywords: NAND Flash Memory, Flash Translation Layer, NAND Flash-Based Storage Systems, Embedded Systems, Performance Optimization, Lifetime Optimization

Student Number: 2011-23362

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the past decade, NAND flash-based storage systems have been more widely adopted, owing to their various advantages over traditional hard disk drives (HDDs) such as high performance, small form factor, high shock resistance, and low power consumption. This wider adoption of NAND flash-based storage systems are enabled mainly by the continuous improvement of NAND chip density, which has significantly contributed to reduce its cost-per-bit value. The capacity of a NAND flash chip has steadily increased by about 30% per year [1, 2], and NAND flash-based storage systems are being inevitably used as main storages not only in mobile computing systems, but also in enterprise server systems.

Capacity-oriented design decisions, represented by semiconductor process scaling, multi-leveling techniques, and large operation units, have played a key role in leading to the dramatic improvement of the NAND density, but they come to at the cost of the performance and lifetime degradation in NAND flash-based storage systems. For example, the read and write latencies of single level cell (SLC) NAND flash memory in 70-nm process are about 20 μs and 200 μs, respectively,

while those of 3-bit multi-level cell (MLC) NAND flash memory in 1x nm process increase to longer than 100 μs and 1000 μs, respectively. In addition, the maximum program and erase (P/E) cycles of the SLC NAND flash memory is about 100K, but that of the 3-bit MLC (i.e., triple-level cell, TLC) NAND flash memory is decreased to few hundreds. On the other hand, since modern high performance computing (HPC) systems require NAND flash-based storage systems to satisfy both the high-performance and long-lifetime requirements as well as the high-capacity requirement, the compromised performance and lifetime must be properly resolved.

1.2 Dissertation Goals

Since each capacity-oriented design decision affects the performance or lifetime of NAND devices differently, it is hard to develop a single ultimate solution that can properly address all the performance and lifetime problems. Therefore, in order for large-capacity NAND storage systems to satisfy various storage requirements at the same time, multiple techniques that properly address each problem need to be developed and effectively integrated. Moreover, considering that semi-conductor manufacturers are trying to continuously increase the NAND density, newly developed techniques should be able to overcome the limitations of existing techniques to deal with the continuously degraded performance and lifetime of large-capacity NAND devices.

In this dissertation, we provide three optimization techniques that aim at improving the performance and/or lifetime of large-capacity NAND storage systems. In particular, our primary goal is to find new optimization hints which have not been exploited by the existing techniques, from the low-level physical properties of NAND devices to I/O characteristic of input workloads. Then we develop performance and lifetime improvement techniques which vertically integrate the optimization hints from different system layers, so as to maximize improvement gains. In developing each optimization technique, we also focus on minimizing negative effects to other storage requirements for enabling an effective integration of individual techniques.

First, we present a system-level approach to improve the performance and lifetime of MLC NAND storage systems. To store multiple bits in a single cell, the write performance of MLC NAND devices is significantly degraded over SLC NAND devices due to a fine-grained voltage control. Moreover, for ensuring the data durability, the number of page programs on NAND devices can be significantly amplified in MLC NAND storage systems, which does not only further degrades the write performance but also negatively affects the storage lifetime. Although the performance and lifetime degradation can be alleviated at the firmware level by leveraging the heterogeneity of MLC NAND pages, the existing page-program sequence significantly limits room of improvements by compelling the firmware to follow a fixed page-program order in a block. By relaxing an unnecessary constrain in the existing page-program sequence, we removed the critical barrier

in flexibly choosing fast pages and slow pages in a block while not compromising the data reliability.

Second, we present a read performance improvement approach for large-page NAND storage systems. As the page size in modern NAND devices becomes larger than the typical request size from host systems, the raw bandwidth of large-page NAND storage systems can be severely wasted because a page is the minimum operational unit of NAND devices. Moreover, since small random reads are dominant in modern HPC applications, the performance of large-page NAND storage systems can be significantly degraded. To resolve this large-page problem, we propose a new page read operation that selectively reads only demanded data with a much shorter latency over full-page reads. By providing an optimal read latency at the NAND device level, the small random reads can be efficiently handled in large-page NAND devices with the minimized waste of raw bandwidth.

Third, we propose an integrated data reduction approach for improving the lifetime of high-density NAND devices. As the semiconductor processes are continuously scaled down and the multi-leveling technique is aggressively exploited, the lifetime of high-density NAND devices are significantly degraded. While several individual data reduction techniques are proposed in previous studies, how to efficiently combining them is not thoroughly investigated yet. Since the target data and the effectiveness of each technique is different, further lifetime gains can be obtained with an effective integration. In particular, we aim to minimize overheads from the integration while maximizing

the data reduction efficiency.

1.3 Contributions

In this dissertation, we introduce three optimization techniques to improve the performance and/or lifetime for large-capacity NAND flash-based storage systems. The contributions of our work can be summarized as follows:

- We present a new page-program sequence, called relaxed program sequence (RPS), which allows the upper management layer to use heterogeneous MLC NAND pages in a much more flexible manner. Using 2x-nm 2D MLC NAND devices and state-of-the-art 3D devices, we validated that the RPS scheme does not compromise the NAND reliability over the existing page-program sequence. We propose an RPS-aware flash translation layer (FTL), called `flexFTL`, which employs two RPS-enabled optimization techniques to improve the performance and lifetime of MLC NAND flash-based storage systems. By taking full advantages of the parity-page backup approach, `flexFTL` almost eliminates the performance and lifetime overheads of paired LSB-page backup operations. Furthermore, by using precious fast pages in a workload-aware fashion, `flexFTL` provides a better user-perceived I/O performance with negligible lifetime overheads.

- We propose a new page read operation, called subpage-parallel read (SPread), which allows large-page NAND storage systems

can efficiently handle small host reads without accessing unwanted page data. Based on the proposed SPread operation, we have developed an SPread-aware FTL, called `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 in storage management tasks (e.g., garbage collection), thus improving the overall I/O performance even under a workload with many writes.

- We introduce an integrated data-reduction technique, called dedup-assisted compression (DAC), for high-density NAND devices. DAC efficiently integrates the existing deduplication and lossless compression techniques, bridging their gap with negligible overheads. By taking into account data similarity across entire storage space, DAC can achieve the maximal data-reduction efficiency under a wider spectrum of input workloads where neither of the individual techniques can effectively deal with.

1.4 Dissertation Structure

This dissertation is composed of six chapters including the introduction and conclusions which are at the first and the last, respectively. The four intermediate chapters are organized as follows:

Chapter 2 briefly explains the background for our work, including basics of NAND flash memory and overall architecture of NAND flash-

based storage systems. We also summaries the existing techniques to optimize performance or lifetime of NAND flash-based storage systems highly related to our proposed techniques.

Chapter 3 introduces a new program sequence in multi-level cell (MLC) NAND flash memory, called Relaxed Program Sequence (RPS), which enables a much flexible page ordering within a NAND block. We explain that the existing program sequence is over-constrained, and show the feasibility of the RPS with validation using the state-of-the-art NAND devices. We also propose an RPS-aware optimization technique, which can almost remove the backup overheads in multi-level cell (MLC) NAND flash memory. In addition, we propose a new system level technique which chooses appropriate pages for incoming host writes to better meet varying performance requirement. By taking into account the performance asymmetry of MLC NAND flash memory, an RPS-aware FTL can use precious fast pages in a workload-ware fashion, thus providing a better user-perceived I/O performance.

In Chapter 4, we introduce a simple yet effective co-design of NAND devices and flash controllers to improve the read performance of NAND flash-based storage systems with large operation units. We first explain the impact of large operation units of NAND devices on the performance at the storage system level, and figure out the feasibility of supporting a fast subpage read in the current design of NAND flash memory. Then we propose design and implementation of a new NAND page-read command that can efficiently handle various patterns of small reads by reading only demanded subpages with an

optimal latency.

Lastly, Chapter 5 presents a new integrated data reduction technique, called Dedup-Assisted Compression (DAC), for enhancing the lifetime of high-density NAND devices. By integrating deduplication and lossless compression in a synergistic fashion, DAC further reduce the amount of stored data over their naive combination. After explaining the key mechanism of the proposed compression algorithm, we present several optimizations to further mitigate its performance and resource overheads.

Chapter 2

Background

In this section, we first explain basics of NAND flash-based storage systems, starting from the physical characteristics of underlying flash cells, to the overall architecture of modern NAND flash-based storage systems. We then review three representative capacity-oriented decisions in designing high-density NAND devices in the perspectives of key idea and impact on performance and lifetime of storage systems. Finally, previous studies highly related to our work are summarized.

2.1 Flash Memory Cell

A flash memory cell stores bit data by changing its threshold voltage V_{th}. Figure 1 describes the structure of a flash memory cell based on the *floating-gate cell* structure[1] and how it stores a bit information. As shown in Figure 1, a flash memory cell is a special transistor with the floating gate which can hold electrons for a certain amount of time (e.g., 1-2 years). The V_{th} level of a flash cell depends on the amount of electrons in its floating gate; the more electrons in the floating gate,

[1] In 2D NAND flash memory, floating-gate cell structures were typically used, while many 3D NAND devices adopt charge trap (CT)-type cell structures. Although their physical characteristics are quite different each other, their functionality is the same in storing and reading bit data.

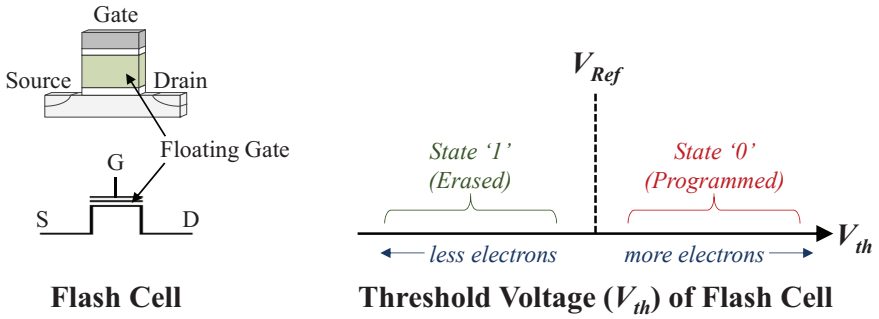


Figure 1: An overview of flash memory cell.

the higher V_{th} level of the cell.

We can inject or eject electrons to/from the floating gate by applying an sufficiently-high voltage (e.g., 10 V), and the V_{th} level of the cell represents bit information stored in the cell. Typically, the operation for injecting electrons in the floating gate is called *program*, and we call the operation for ejection electrons from the floating gate as *erase*. We can *read* bit data stored in a cell by applying a reference voltage V_{Ref} which is between the V_{th} of the programmed state and that of erased state; if the cell is turned on (i.e., $V_{th} < V_{Ref}$), the cell is in the erased state (i.e., '1'); otherwise, it is in the programmed state (i.e., '0').

Flash memory cells have limited endurance. The program voltage V_{PGM} and erase voltage V_{ERASE} induced to memory cell are quite high and repeated program and erasure operations decrease flash cells' (or floating gates') capability to keep their data (or electrons) for the requirement (e.g., 1-2 years). As a results, after a certain number of program and erase (P/E) cycles, flash cell .

2.2 NAND Flash Memory

In NAND flash memory, multiple flash cells operate in a grouped manner in order to achieve high bandwidth. Figure 2 illustrates the overall architecture of NAND flash memory. Multiple cells compose a *page* which is the unit of read and program operations. Since multiple cells are programmed together, it is impossible to overwrite page (i.e., re-program cells with different data). For a simple example, suppose that a page is consist of two flash cells, and its cells were programmed with data '0' and '1'. In that case, we cannot change the page's data, i.e., '01', to '10', because the most significant bit '0' cannot be changed to '1' by a program operation (it can be done only by an erase operation).[2] For this reason, a page has to be used in an *erase-before-write* fashion.

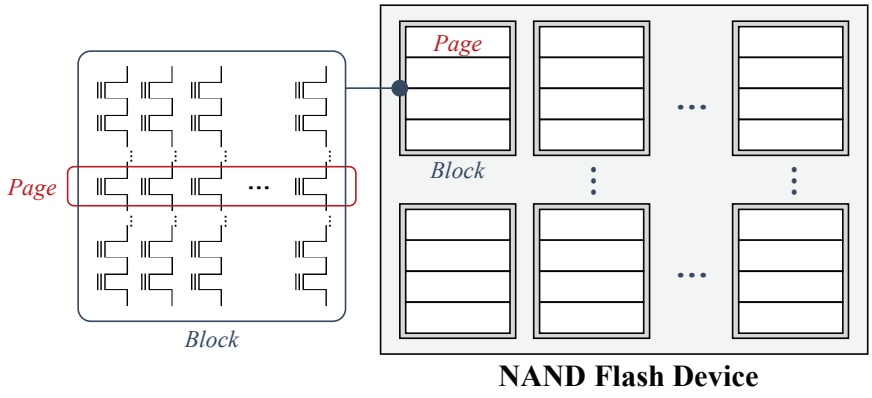


Figure 2: An overall architecture of NAND flash memory.

[2]In the example, overwriting is possible if data to newly program were '00'. However, considering that a page in modern NAND flash memory consists of 32-128K flash cells, it has little change in practice.

For various reasons, NAND flash memory performs erase operations on a group of multiple pages, called a *block*. First, it allows more cells to be share the substrate, thus enabling a higher device density. Second, it can also enhance the reliability, since flash cells are less affected by erasures of neighboring cells (they erased at the same time). Finally, the erase bandwidth can be improved since more cells are erased in parallel, even though block erasures typically take more than twice as long as the page-program latency.

2.3 NAND Flash-Based Storage Systems

Figure 3 depicts an overall architecture of modern NAND flash-based storage systems. To achieve high performance, NAND flash-based storage systems employ several NAND devices. The read and write bandwidth of a single NAND device is far limited even over the traditional hard disk drives (HDDs). For example, a state-of-the-art 3D NAND device takes about 50 μs to read 4-KB page, thus providing only 80 MB/s of read bandwidth. NAND flash-based storage systems overcome this limited performance of a single NAND device by allowing multiple NAND devices to be operate in parallel. In order to minimize performance interference between NAND devices, NAND flash-based storage systems are design to have as many buses as possible (i.e., as less NAND devices per bus as possible) with dedicated hardware controllers which can handle multiple NAND commands on their NAND device simultaneously.

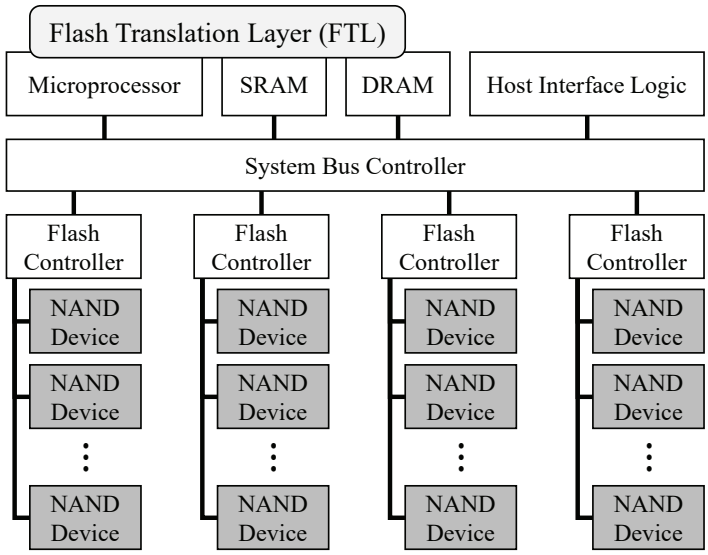


Figure 3: An overall architecture of typical NAND flash-based storage systems.

Most modern NAND flash-based systems run a special storage firmware, called a *flash translation layer* (FTL). Due to the unique features of NAND flash memory, such as the erase-before-write nature and operation-unit asymmetry, a NAND flash-based storage system should deal with underlying NAND devices in different ways over other storage medium such as DRAM and magnetic disks. An FTL is responsible for providing backwards compatibility to the block I/O interface, hiding the unique features of NAND flash memory.

FTLs support two main functionalities: the address translation and garbage collection (GC). NAND flash memory does not support *in-place update*, so an FTL should handle host writes in an append-only manner, writing incoming data to free (i.e., erased) pages. As

a result, when a host system updates data at the same location in the host system view, the physical location of data are changed. In order to handle future reads on the data, an FTL have to keep track of the physical location (i.e., physical address) in the storage system corresponding the location in the host system view (i.e., logical address). Since this logical-to-physical (L2P) mapping is accessed in every read/write requests, the effectiveness of managing L2P mappings is critical to the performance of NAND flash-based storage systems.

In order to maintain free pages for future writes, an FTL has to perform GC. GC procedure reclaims *invalid* pages whose data are not necessary any longer as host system deleted the data or update them with new data at their corresponding logical address. It requires a block erasure, so all the valid pages in a victim block to be erased have to be moved to other free pages. These copy overheads can significantly degrade the performance and lifetime of NAND flash-based storage systems, so typical garbage collectors choose the target block which has a largest number of invalid pages.

2.4 Capacity-Oriented Design of NAND Flash Memory

Despite various advantages of NAND flash memory over HDDs, NAND flash-based storage systems were not so popular in the early stage because of the high cost-per-bit value over HDDs. Hence, NAND manufacturers' primary goal has always been to improve the density

of NAND devices, even though it may negatively affects on the performance and/or lifetime of NAND devices. Thanks to various capacity-oriented design decisions such as semiconductor process scaling, multi-leveling technique, and large operation unit, the cost-per-bit value of NAND flash-based storage systems have continuously decreased, thereby significantly increasing the popularity of NAND flash-based storage systems in modern computing systems. On the other hand, it also leads to serious degradation of the performance and lifetime of NAND devices. In this section, we briefly explain capacity-oriented design decisions in NAND flash memory, focusing on their key ideas and impacts.

2.4.1 Semiconductor Process Scaling

Semiconductor process scaling enables a small cell size, thus enabling more flash cells to reside in the same die area. It is obvious that the improvement gains of process scaling is proportional to the square of the reduction amount of the cell size. Because of such a huge improvement gain, NAND manufacturers have aggressively developed advanced process technologies, reducing the technology nodes from hundreds nm to 1x-nm [2].

In spite of its effectiveness in increasing the NAND density, the performance and lifetime of NAND flash memory have been degraded by on continuous process scaling. Since the smaller cell, the fewer electrons in its floating gate (i.e., the narrower V_{th} margin of a cell), more precise voltage control is required for program operations, thus

increasing the program latency. Furthermore, as the V_{th} margin between programmed and erased states is also reduced, impact of charge loss in the floating gate severely increases, making it difficult for flash cells to guarantee the data reliability at the same P/E cycles.

2.4.2 Multi-Leveling Technique

Multi-leveling technique allows multiple bit values to be stored in a single flash cell, by increasing the number of distinct V_{th} states in a cell. Figure 4 depicts different V_{th} distributions of flash cells in a single-level-cell (SLC) NAND page and 2-bit multi-level-cell (MLC) NAND page. In 2-bit MLC NAND devices, four distinct V_{th} states exist, each of which represent 2-bit values, '11', '01', '00' and '10'. Since the multi-leveling technique can increase the device density without

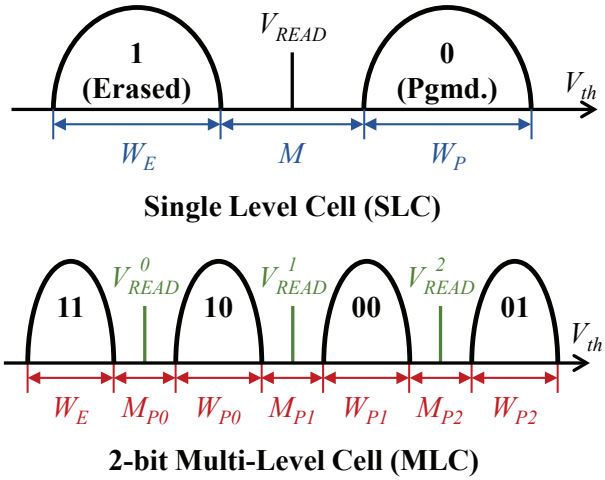


Figure 4: Threshold voltage distributions of SLC and MLC NAND Flash Memory.

semiconductor process scaling, most NAND manufactures have aggressively adopted it to overcome the scaling limitation.

However, as shown in Figure 4, in order to increase the number of V_{th} states in the same V_{th} window, it is inevitable to reduce the margin M_{Pi} between each state and the width W_{Pi} of each state. As a consequence, the program performance and lifetime are degraded in a similar manner to in the process scaling. Furthermore, more verifications (in the above example, two verifications) are necessary to read the most-significant-bit (MSB) value, thus increasing the page-read latency as well.

2.4.3 Large Operation Unit

The page size have been steadily increased over decades, from 256 bytes to 16 KB [3, 4]. A large operation unit is preferable to improve the density of NAND devices, since it reduces the amount of peripheral circuits for addressing pages and blocks. Moreover, increase of the program latency is trivial, thus providing a higher write-bandwidth at the device level.

On the other hands, read and erase operations takes more time on a large page and block, as the amount of charged/dis-charged electrons for each operation increases over a small page. Furthermore, at the storage system layer, a large page incurs a lot of inefficiency in dealing with small requests whose size is smaller than the page size, and it may result in the performance and lifetime degradation.

2.5 Related Work

2.5.1 Optimizations for MLC NAND devices

There have been several previous studies which attempted to alleviate the performance and lifetime degradation of MLC NAND flash-based storage systems. Lee *et al.* [5] proposed a parity page-based backup scheme that reduces overheads of least-significant-bit (LSB)-page backup operations which is necessary for guaranteeing the data durability in MLC NAND devices. Instead of backing up individual LSB pages, it backs up only a single parity page of multiple LSB pages. Although the proposed scheme can greatly reduce LSB-page backup operations over a naive approaches, the backup overheads still remain non-trivial since the number of LSB pages that can be backed up together is severely limited under the current program-order constraints.

Lee *et al.* [6] proposed a new flash file system, called FlexFS, which exploits the SLC-mode programming [7] in MLC NAND flash memory. When a high write bandwidth is required, FlexFS quickly handles incoming writes by performing only LSB-page writes (i.e., using MLC NAND blocks as SLC NAND blocks). Although FlexFS may achieve the peak I/O performance close to SLC NAND devices, all the MSB pages in a block used with the SLC-mode programming must be discarded under the existing page-program sequence, thus wasting half the capacity of the block.

Grupp *et al.* [8] proposed a *Return to Fast* (RTF) scheme, which supports successive LSB-page writes in MLC NAND storage systems.

Instead of exploiting the SLC-mode programming, it maintains multiple writing points within multiple blocks so that multiple LSB pages can be successively used. The RTF schemes tries to handle host writes with as many LSB pages as possible, while consuming MSB pages for internal writes so as to return write points to be LSB pages. The proposed RTF scheme can increase the host-experienced performance without wasting the capacity, but the improvement gains can be limited because a large LSB-page pool makes the block management more complicated. Furthermore, returning write points to be LSB pages may lead to a premature GC which can significantly degrade the performance and lifetime of NAND storage systems.

Ho *et al.* [9] proposed a new programming scheme, called SLC-Like Programming (SLP) scheme, which can rapidly program MSB pages when their corresponding LSB pages are invalidated. If an LSB page is invalidated, the two V_{th} states previously formed for the LSB data can be treated as the same state '1' (*not programmed*), since the LSB data became meaningless. Therefore, for programming new data to the corresponding MSB page, it is sufficient to form a single V_{th} state '0' distinguished from the previous two states, instead of two different V_{th} states. In SLP scheme, since V_{th} margin per state is significantly increased, an MSB page can be quickly programmed with a coarse-grained voltage control. However, there is little chance for a written LSB-page to be invalidated before using its paired MSB page, since an MSB page is to used shortly after its LSB page in the existing page program sequence.

2.5.2 Optimizations for Large-Page NAND Devices

In order to solve the large page problem, several optimization techniques at different system levels are presented in previous studies. In particular, subpage-based programming techniques [10, 11] have been proposed for alleviating the performance degradation due to small writes. By allowing multiple subpage writes at a single NAND page without block erasures, those techniques can significantly reduce the write amplification factor (WAF) of small writes, thus improving both the performance of lifetime of large-page NAND storage systems. Although they are effective to alleviate the small write problem in large-page flash, little performance gains are obtained when small reads are dominant.

In order to improve the I/O performance for small random reads, various techniques have been proposed at upper system layers. Liu *et al.* [12] proposed a new I/O management scheme for graph processing applications which keeps application's (small and random-accessed) data in a fine-grained unit to minimize the amount of data unnecessarily read. Hahn *et al.* [13] proposed a defragmenter for flash storage systems that reduces small random reads caused from file fragmentations, by making them logically contiguous so that they can be issued with fewer number of block I/Os. Both approaches can efficiently reduce I/O stack overheads introduced from small random reads, but their performance gains can be limited if handling small reads takes

the same amount time as the full-page read latency in large-page NAND devices.

2.5.3 Write-Traffic Reduction Techniques

The content-aware SSD (CASSD) [14] and context-aware FTL (CAFTL) [15] have been proposed in previous studies, to reduce the write traffic to NAND devices by exploiting the data deduplication technique that skips writes whose data are identical to the previously stored. CaSSD supports in-line deduplication with a hardware acceleration module to minimized computational overhead of a cryptographic hash function, while CaFTL performs software-based deduplication for achieving the high cost-efficiency of storage systems. To compensate performance overheads from software-based hash calculation, CaFTL reduces runtime overheads by adopting a set of acceleration techniques.

There have been several previous studies [16, 17, 18] which exploit lossless compression technique to reduce the amount of data written to NAND storages. The proposed techniques reduce the write traffic sent to the NAND devices by using lossless compression algorithm to remove repeated bit patterns. In order retain the high I/O performance, they alleviate computational overheads from encoding/decoding incoming data by leveraging hardware compression modules.

Wu *et al.* [19] proposed a novel design of NAND storage systems, called ΔFTL, which exploits delta-compression techniques. Based on

an observation that many file systems overwrite data slightly updated from the original data, ΔFTL performs delta-compression with incoming data, using data previously stored at the same logical address as a reference. finds reference (i.e., similar) data previously stored in the storage system. It first calculates XORed data from two similar data, and then it compresses the resulting data with a lossless compression algorithm. Since the XORed data of similar data sets have low entropy, the compression efficiency can be significantly improved.

Individual techniques mentioned above exploit different properties of incoming data, so their effects on write traffic reduction are different depending on characteristics of workloads. Data deduplication is effective only when data has high value locality – that is, exactly the same values are frequently written to flash. It fails even if there is a single bit difference between the incoming data and reference data (which is previously written in flash). Lossless compression is not dependent upon value locality, but it works well only when incoming data has low entropy. Although delta compression is not affected by value locality and data entropy, it can achieve a high compression ratio only when slightly modified data is frequently overwritten.

Chapter 3

Relaxed Program Sequence for MLC NAND Devices

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 SLC NAND flash memory. Despite its high effectiveness in increasing the density of NAND devices, it also negatively affects the performance of NAND flash memory, introducing several technical challenges at the storage system layer. Considering that the multi-leveling techniques will be inevitably adopted in NAND flash-based storage system to satisfy the high-capacity requirement of modern computing systems, these challenges should be properly addressed. In this chapter, we address the side-effects of the multi-leveling techniques for a program operation's perspective.

3.1 Motivation

3.1.1 Performance Asymmetry between MLC pages

Figure 5 illustrates a typical program scheme for 2-bit MLC NAND devices under the fine-grained charge placement and sensing

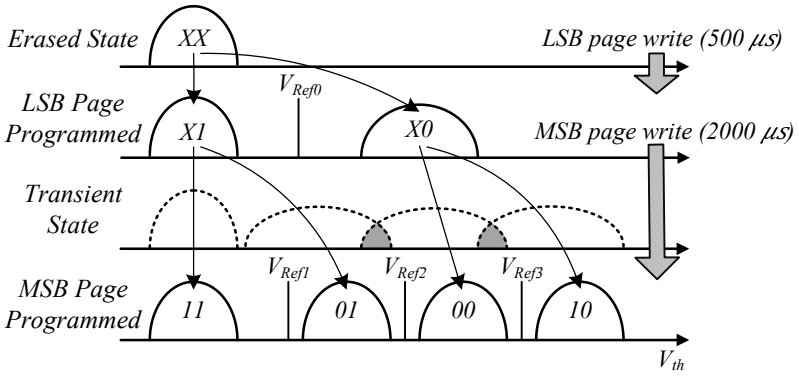


Figure 5: An illustration of the program mechanism for 2-bit MLC NAND devices.

mechanism[1]. Many MLC NAND devices allows multiple bits in a cell to be individually programmed so that an FTL can efficiently deal with small host writes. If individual programs of page is not supported, i.e., whole pages that share a wordline (a group of flash cells) have to be programmed at once, an FTL need to buffering a page-sized write until next writes' arrival or to program the page with a dummy page. To avoid such inefficiencies, as shown in Figure 5, the least significant bit (LSB) page is first programmed by forming two V_{th} distributions, and the most significant bit (MSB) page is programmed later with four V_{th} distributions based on the corresponding LSB page's V_{th} distributions.

The average latency increase in MLC NAND devices is inevitable as explained in Section 2.2, but such an individual page program additionally introduces a unique property of MLC NAND flash mem-

[1]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 TLC NAND devices [20] with a similar program scheme

ory: the performance asymmetry between MLC NAND pages. When an 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, an MSB-page program 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 increase 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) [21].

Although some existing studies [6, 8] attempted to alleviate the performance 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.

3.1.2 Paired LSB Page Backup Problem

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 the same wordline with the MSB page must be saved to a different wordline, 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, and thus 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 [22].

Buffering the paired LSB page until finishing an MSB-page program can guarantee the data durability without paired LSB-page backup operations, but it leads to non-trivial cost of storage systems. Specifically, it requires super-capacitors to guarantee that ongoing MSB-page programs can be normally finished under a sudden power-off. Furthermore, the capacity of super-capacitors should be huge enough to supply power until buffered LSB pages also repro-

grammed (on other pages) upon unexpected failures of the on-going MSB-page programs. Since there are a lot of metadata to be immediately flushed to NAND devices such as logical-to-physical (L2P) mappings under a sudden power-off, increasing the amount of data to flush is not only impractical, but also infeasible in some environments.

A paired LSB-page backup operation requires a page copy operation, and thus 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, even when we use only LSB pages for paired LSB-page backup operations. 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.

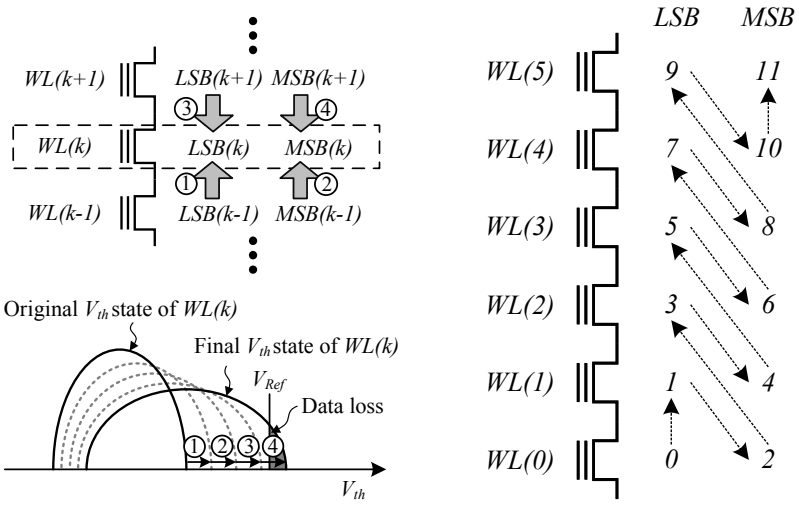
In order to tackle the paired LSB-page backup problem, a parity page backup scheme [5] has been proposed, but its improvement gain is rather limited under the FPS scheme. It reduces the number of paired LSB-page backup operations, by backing up only one parity page of multiple LSB-pages whose associated MSB pages not yet programmed. However, under the FPS scheme, an MSB page is programmed shortly after programming the corresponding LSB page. As a result, the maximum number of LSB pages that can be backed up together is significantly limited.

3.2 Relaxing Program Constraints in MLC NAND Devices

3.2.1 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*, the side effect of NAND program operations, so that the operation margin in the V_{th} window can be secured [23]. Since the NAND operation margin gets reduced with shrinking semiconductor processes and adoption of multi-leveling techniques, 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 flash cell is additionally programmed by program operations to its immediately neighboring cells. When this interference is strong, the V_{th} states of programmed cells in the wordline can be shifted to the right, thus causing unexpected changes in the stored data. Figure 6(a) illustrates the worst-case example of the cell-to-cell interference problem when MLC pages in a block can be written without any *restriction* on the page program order. We denote the k-th wordline 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 max-



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

(b) A representative FPS scheme [23].

Figure 6: The FPS scheme for minimizing the cell-to-cell interference problem in MLC NAND devices.

imized when its four neighboring pages, $LSB(k-1)$, $MSB(k-1)$, $LSB(k+1)$, and $MSB(k+1)$ are programmed after *WL(k)*. In such a case, the original V_{th} state may be shifted to the far right (e.g., ④ in Figure 6(a)) so that its V_{th} state can be misinterpreted, thus losing the original data stored in the cell.

The FPS scheme addresses the cell-to-cell interference problem by limiting the maximum amount of interference that a programmed wordline can be affected. 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 pro-

gram sequence for pages in a block. Figure 6(b) shows a representative FPS scheme which is commonly employed in recent MLC NAND devices [23]. As shown in Figure 6(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.

3.2.2 Relaxed Program Sequence Schemes

Our key insight is that the FPS scheme, which significantly limits the effectiveness of the existing optimization techniques, is an *over-specification* that 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. In order to explore the possibility of more flexible page program orders, we formalized the FPS scheme of Figure 6(b) using its four constraints on the page program order as summarized below:

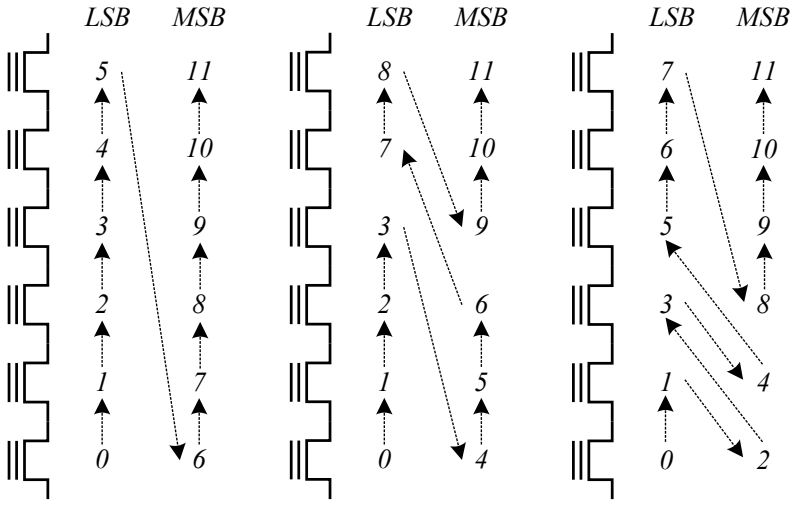
- *Constraints 1 & 2*: Before $LSB(k)$ (or $MSB(k)$) is written, $LSB(k-1)$ (or $MSB(k-1)$) should be written (where $k \geq 1$).
- *Constraint 3*: Before $MSB(k)$ is written, $LSB(k+1)$ should be written (where $k \geq 0$).
- *Constraint 4*: Before $LSB(k)$ is written, $MSB(k-2)$ should be written (where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

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 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 7(a), all the LSB pages in a block can be sequentially written before their paired MSB pages. Figures 7(b) and 7(c) show other examples of possible program orders where LSB-page writes and MSB-page writes are flexibly intermixed.

3.2.3 Validation of RPS Schemes

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 page ordering schemes. As a main evaluation metric, we measured the

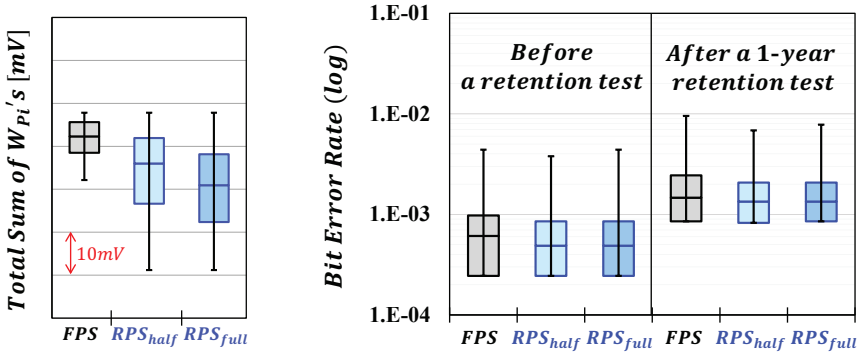


(a) RPS$_{full}$ order. (b) RPS$_{half}$ order. (c) A random order.

Figure 7: Examples of three different program orders under the RPS scheme.

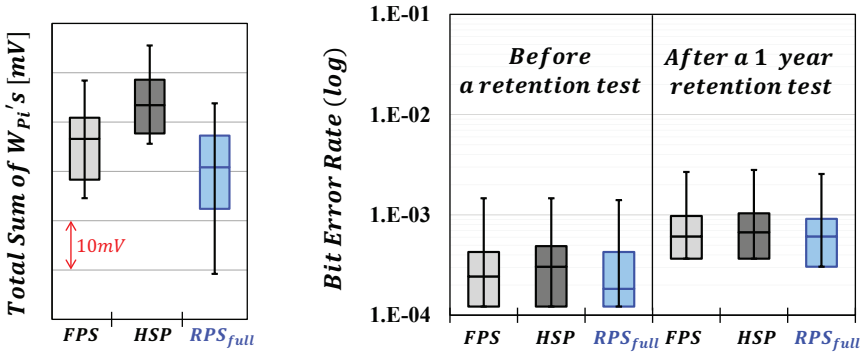
width W_{Pi} of the V_{th} distribution for each V_{th} state because W_{Pi} reflects the overall impact of the cell-to-cell interference quantitatively (the lower, the better). For our validation, we used state-of-the art 3D MLC NAND devices, as well as 2x-nm 2D MLC NAND devices. For each technology of devices, our verifications were performed with more than 90 blocks out of three NAND devices. 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$_{half}$, in 2D NAND devices, as shown in Figures 7(a) and 7(b), respectively. In addition, for 3D NAND devices, we compared RPS$_{full}$ with the existing High Speed Program (HSP) scheme [24] as well as the FPS scheme, which programs two paired pages at the same time (after buffering them).

Figures 8(a) and 9(a) show the measured distributions of the



(a) W_{Pi} distributions. (b) Bit error rate distributions.

Figure 8: Reliability comparisons for the FPS scheme and RPS schemes in 2D NAND flash memory at the worst-case condition.



(a) W_{Pi} distributions. (b) Bit error rate distributions.

Figure 9: Reliability comparisons for three different page orderings in 3D NAND flash memory at the worst-case condition.

total sum of W_{Pi}'s for more than 5,000 pages out of 90 blocks in each type of device. As we expected, W_{Pi}'s under RPS schemes were not increased over the FPS scheme as well as the HPS scheme, thus showing that the overall cell-to-cell interference with the RPS scheme

was not higher than that with other page-ordering schemes. In order to compare the overall NAND reliability under the RPS scheme over the other schemes, 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 Figures 8(b) and 9(b), the bit error rate for the RPS scheme was not higher than that for the other schemes 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 page ordering schemes without affecting the overall NAND reliability.

3.3 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. First, under the RPS scheme, a large number of LSB pages on the same block can be successively written without an MSB-page write, making the parity-based backup scheme very efficient. For example, if all the LSB pages of a block are written before an MSB page of the block, only a single parity-page backup is required during using all the pages in the block. Considering modern NAND devices typically have a number of pages in a block (e.g., 384 pages per block), the overheads of paired LSB-page backup can be negligible. Second, 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.

Based on these RPS-enabled optimizations, we have designed a novel FTL for MLC NAND flash-based storage systems, called `flexFTL`. Figure 10 shows an overall organization of `flexFTL`. `FlexFTL` is based on an existing page-level mapping FTL with two additional modules, the page allocator and the block pool manager, for supporting the RPS-enabled 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

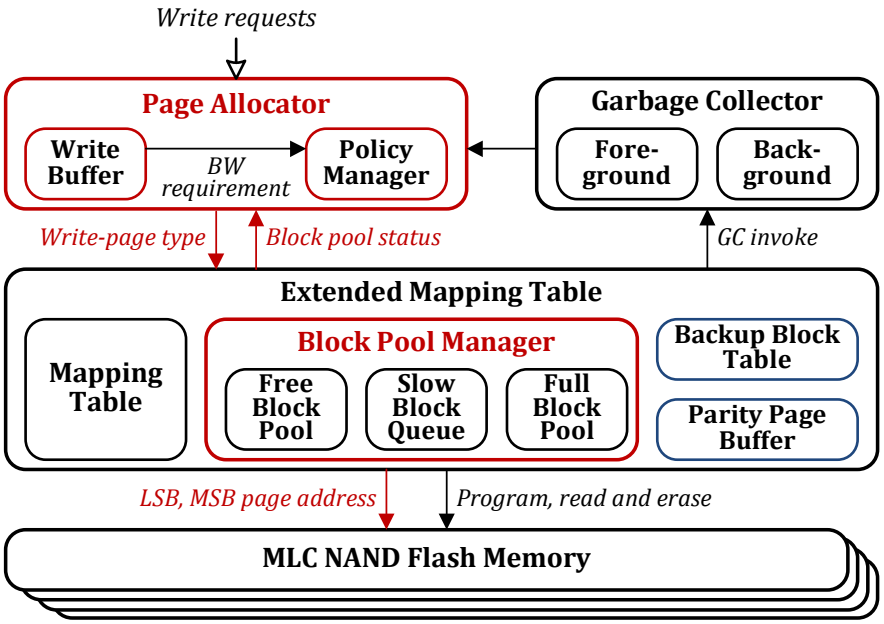


Figure 10: An organizational overview of `flexFTL`.

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 the storage idle time. 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 type of pages.

3.3.1 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. For this reason, in `flexFTL`, we chose a particular page-program order (which is an instance of the RPS scheme), and designed a new block management technique, called 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 7(a)). When the 2PO scheme is used, a block can be viewed as cycling through four distinct states as shown in Figure 11. Starting 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

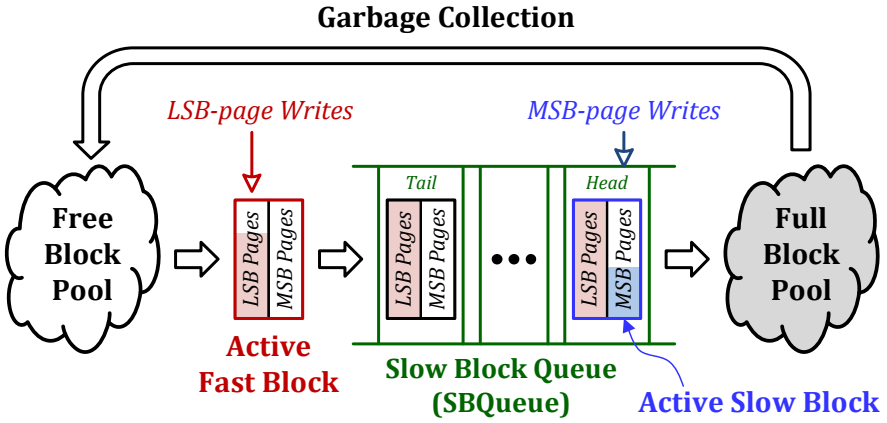


Figure 11: A life cycle of a NAND block in `flexFTL`.

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, which is used to perform 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 (in 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.

3.3.2 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 and the quota q for successive LSB-page writes.

The write buffer utilization u is used to estimates the current write-bandwidth requirement of the host system (i.e., for C1). 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, on the other hand, is used to estimate how a future write bandwidth is affected by the current LSB-page write (i.e., for C2). 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 appropriate page type for incoming writes 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[2]. 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 of storage systems. Using q avoids such a dramatic drop in the write bandwidth by limiting the number of consecutive LSB-page writes, thus forcing to use MSB pages when $q = 0$. Once the quota q is expended, `flexFTL` works in a similar fashion as an FPS-based FTL by alter-

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

nating LSB-page writes and MSB-page writes[3]. 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 (i.e., when a high bandwidth is not necessary), 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.

3.3.3 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 [5]. Figure 12(a) illustrates how the backup procedure works in `flexFTL`. The backup procedure is closely connected to the proposed 2PO scheme.

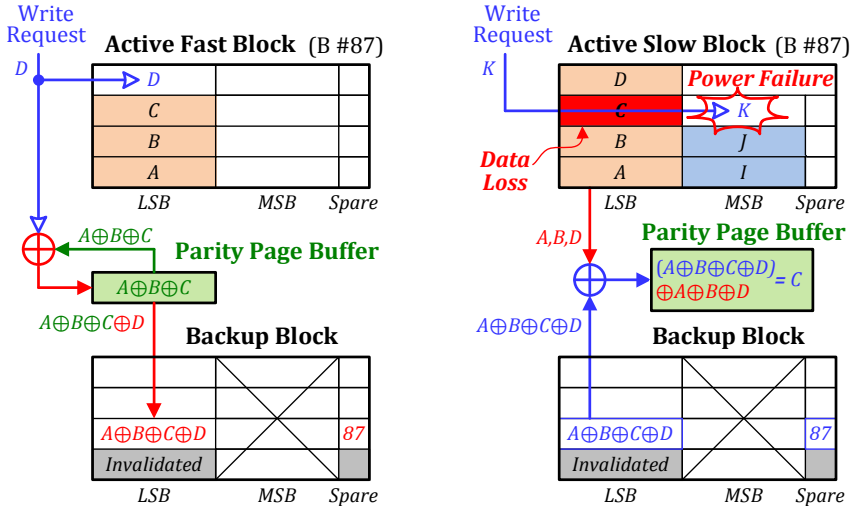
[3]Note that, nevertheless, `flexFTL` can achieve a higher average write bandwidth over FPS-based FTLs, thanks to the per-block parity-page backup scheme described in Section 3.3.3

While the LSB pages of the active fast block are written, using the parity-page buffer, `flexFTL` computes the accumulated parity values (i.e., XORed values) of all the LSB pages written in the active fast block. For example, in Figure 12(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[4]. 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[5].

The error recovery procedure takes reverse steps of the backup procedure as shown in Figure 12(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 current active block. In order to check a data loss in the active slow block, we read all the LSB pages of the active block while recomputing the accumulated parity values on the parity page buffer. If all the LSB pages

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

[5]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 12: Examples of backup and recovery procedures.

were successfully read, we have recomputed the parity page for the active 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 XORing the saved parity page (in the backup block) and the recomputed parity values, we can recover the lost page. Note that at most one LSB page's data can be lost due to a sudden power-off because only one MSB page in a block can be programmed at the same time, and thus our backup and recovery procedure can always

guarantee the data durability.

The per-block parity page backup scheme requires to compute parity values, so it may increase the latency of LSB-page writes and the reboot time after a sudden power-off. However, the overhead of computing parity values for incoming writes on LSB pages is insignificant over the NAND program time which is about 1 2 ms on average. The increase of the reboot time after a sudden power-off is relatively high since, when the sudden power-off occurred during an MSB-page write, the error recovery procedure must recompute parity values for all the active blocks. Even if we were using LSB pages at the moment of the sudden power-off, 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 recomputing 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 192 LSB pages, the extra read overhead is less than 200 ms (i.e., 16 $[chips] \times 1\ [\frac{blocks}{chip}] \times 192\ [\frac{pages}{block}] \times 40\ [\frac{\mu s}{page}] = 122.88$ 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.

3.4 Experimental Results

3.4.1 Experimental Settings

In order to evaluate the effectiveness of `flexFTL`, we have implemented it using open development/evaluation platforms [25, 26]. Our evaluation platform can support up to the 512-GB capacity, but for fast evaluation, we set the capacity of the target storage system to 16 GB. The storage system is based on 2x-nm 2D MLC NAND devices, and Table 1 summaries the detail configuration of the target storage system and NAND timing parameters.

For our evaluations, we used five distinct I/O workloads, which were generated from Sysbench [27] and Filebench [28]. As summarized in Table 2, 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

Table 1: A summary of storage configurations used for evaluations.

Storage configuration		NAND time parameters		
# of buses	8	Read latency	LSB	30 μs
# of chips per bus	4		MSB	60 μs
# of blocks per chip	512	Program latency	LSB	600 μs
# of pages per block	256		MSB	2000 μs
Page size	4 KB	Erase latency		2 ms

Table 2: Descriptions of workloads used for evaluations.

	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

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.

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 [5], 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[6]. `RtfFTL`, which is based on the *return-to-fast* scheme proposed in [8], performs successive LSB-page writes for incoming write requests under the FPS scheme, by maintaining multiple active blocks per chip. 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

[6] As shown in Figure 6(b), at most two LSB pages can share a parity backup page before programming their paired MSB pages.

set u_{high} and u_{low} to 80% and 10%, respectively. The initial value of q is set to 5% of total LSB pages. The background garbage collector in `rtfFTL` aggressively consumes MSB pages in idle times so as to sustain successive LSB-page writes. On the other hand, in rest three FTLs, a background garbage collector is invoked during storage idle times when the number of free blocks is less than 10% of the total capacity.

3.4.2 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 13 shows IOPS values of four different FTLs under each workload, normalized to `pageFTL`. As shown in Figure 13, `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.

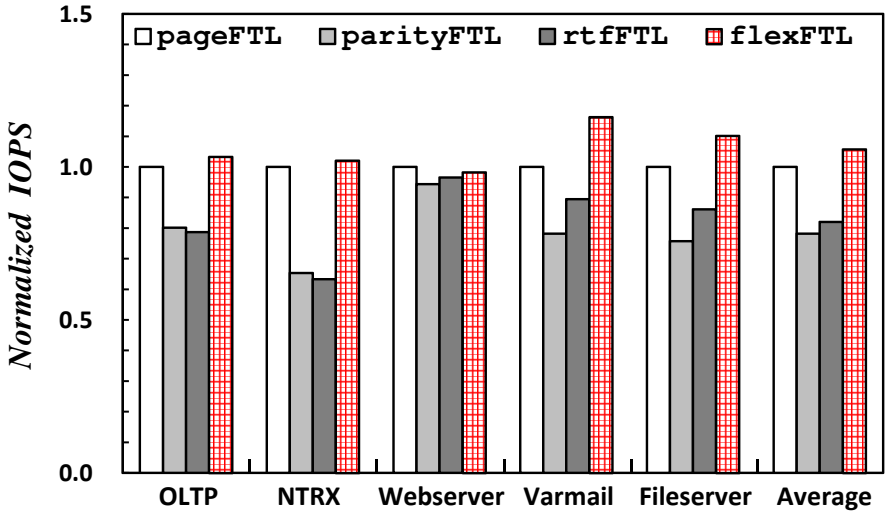


Figure 13: A performance comparison of `flexFTL` over FPS-based FTLs.

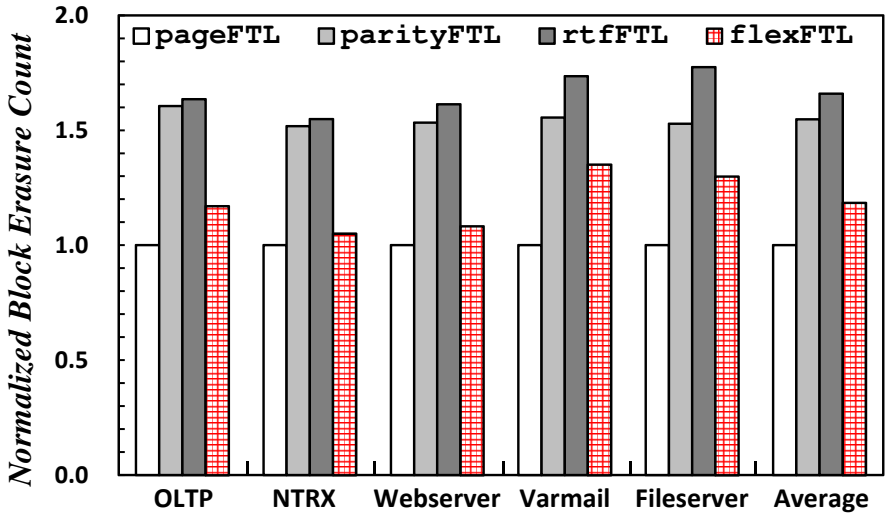


Figure 14: A lifetime comparison of `flexFTL` over FPS-based FTLs.

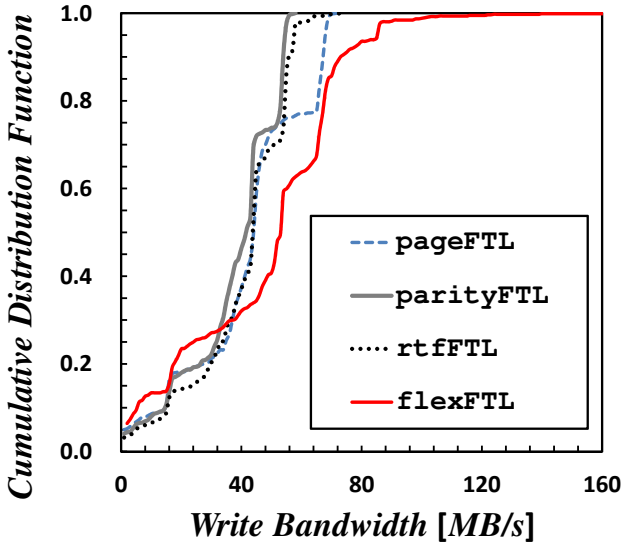


Figure 15: CDF curves of write bandwidth for Varmail workload.

Figure 14 shows normalized block erasure counts of four different FTLs under each workload. As shown in Figure 14, `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 particular, `parityFTL` exhibited the highest increase of block erasure counts over `pageFTL` in every workload due to premature background GC to sustain the LSB-page pull.

In order to understand how `flexFTL` can better meet high write-bandwidth requirements over the other FTLs, we ploted the cumulative distribution function (CDF) curves of write bandwidth for Var-

mail. As shown in Figure 15, 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.

Chapter 4

Subpage-Parallel Read for Large-Page NAND Devices

4.1 Motivation

While the capacity of a NAND chip has increased from 16 Gb to 128 Gb, the NAND page size has increased by eight times as well from 2 KB to 16 KB [2, 3, 4]. In order to maximally increase the capacity of a NAND chip, a large NAND page is a (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. 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 wasting the raw bandwidth of NAND storage systems. (In this paper, we call such a read *amplified*).

Our key observations are that 1) the root cause of the amplified read problem is that reading a part of a NAND page is not *size-*

proportional[1], and 2) a small read can be significantly faster than a large read in NAND flash memory. Based on our observations, at the device level, we propose a new page read operation, called a subpage-parallel read (SPread), which can read multiple subpages at the same time while skipping unneeded subpages. By avoiding reads for unnecessary subpages, the read latency of an SPread operation is proportional to the requested data size without read amplification.

4.1.1 Page-Read Mechanism of NAND Flash Memory

To better understand the feasibility of a *fast small read* in large-page NAND devices, it is necessary to know the mechanism of page reads in details. Figure 16 describes the procedure of reading a page from a NAND chip, which is composed of two phases: 1) sensing memory cells and 2) transferring read data. For a host read, the flash controller issues a NAND read command to the NAND device. An array of memory cells inside the NAND device is organized in rows and columns. The cells at the same row are connected to a wordline (WL), and each cell in the wordline is linked to a different bitline (BL). A WL is used to select a group of cells composing a NAND page, while BLs are used for sensing the status of the cells in the selected WL. The NAND read command specifies a target WL and BLs with given column and row addresses in multiple command slots.

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

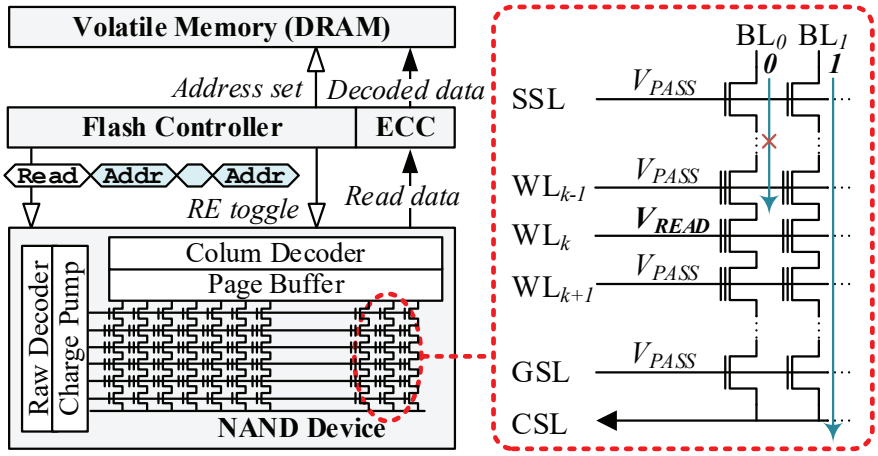


Figure 16: An illustration of the page-read mechanism in NAND flash memory.

For the first phase in reading data, BLs related to a target WL are pre-charged *in parallel*. After pre-charging them, the NAND device performs sensing data by applying appropriate voltages to WLs. Depending on the status of a cell, a corresponding BL either remains charged or is being discharged. In Figure 16, BL_0 remains charged since the cell at WL_k is programmed, while BL_1 whose related cell is erased sinks current. Two different sensing results are interpreted as a logical bit value '0' or '1', respectively. Sensed bit values are temporarily stored in the page buffer. As the second phase, the flash controller moves bit values from the page buffer to storage DRAM. This is done by toggling the read enable (RE) signal for each byte. To prevent data loss from bit-flip errors, when data were being written, they were encoded with an error correction code (ECC) in a codeword unit (typically 512 B - 2 KB) [29]. Thus, the raw data read from

the device must be decoded with the ECC parity.

The latency of a page read is mostly decided by the above two phases. First, the time t_R spent for sensing data from the NAND device increases in proportion to the number of BLs. This is because, with more BLs, discharging them takes much longer (See Section 4.2.3). Second, the time t_{DMA} for data transfer linearly increases according the read size. This is due to the fact that more RE toggles are required to fetch data from the NAND device. Performing the ECC would increase read latency as well, but its impact is relatively small since an ECC engine is implemented in hardware and it works in a pipelined manner with other reads and writes.

4.1.2 Performance Impact of Amplified Reads

As described before, as the size of a NAND page becomes larger, the latency of reading a single page tends to increase [30]. 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 [2, 3]. 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

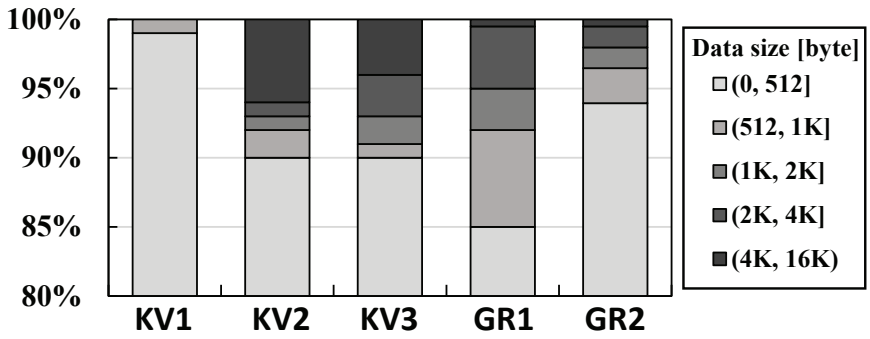


Figure 17: Distributions of the read size in key-value store and graphic processing applications.

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 HPC systems, a key-value store [31] and a graph processing application [12]. We first realized that the HPC applications themselves generate lots of small random reads to storage devices. Figure 17 shows distributions of read data sizes of the two applications. As shown in Figure 17, 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 [12, 31], 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)

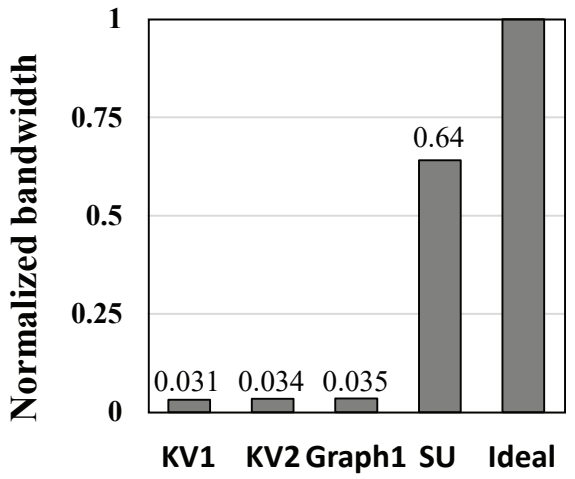
scheme that maps 4-KB logical pages to a larger physical page, say 16-KB[2]. The FGM scheme buffers four 4-KB logical pages and writes them to a 16-KB NAND page together. By doing this, FGM allows us to avoid expensive read-modify-write (RMW) operations in case where a small random update comes [10]. 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 [32] to generate similar distributions of read data sizes as in

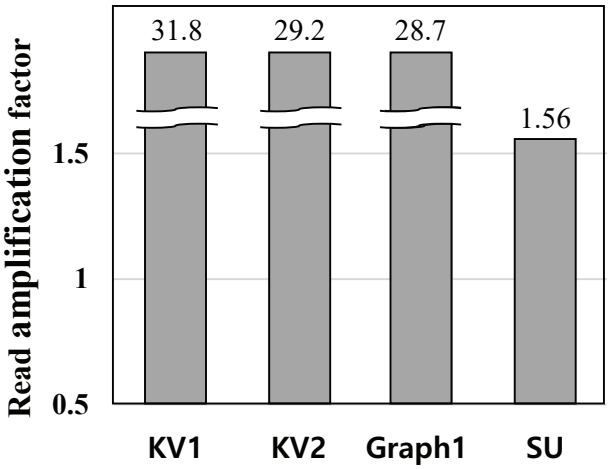
[2]The size of (small) logical pages is typically set to 4 KB, to be equal to the default block size of many modern file systems such as Linux ext4 and FAT32.

Figure 17. Figure 18(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 simas the NAND page size becomes larger, ilar 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 18(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 18(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 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 am-



(a) Normailized read bandwidth.



(b) Read amplification factor.

Figure 18: Impact of amplified reads on I/O performance.

plification 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.

4.2 Design and Implementation of SPread

4.2.1 Design Requirement

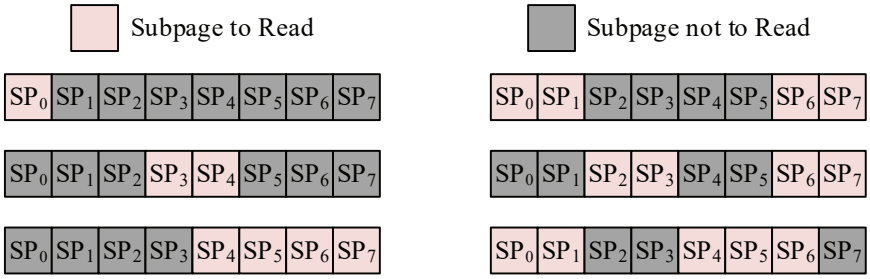
Based on insights from Section 4.1, we propose a new type of a NAND device that supports selective subpage reads from a NAND page, called SPread. An SPread-enabled NAND device exposes a reconfigurable subpage read interface that allows us to read *one* or *more subpages in parallel* from a large-size NAND page. As described in Section 4.1, the read latency is decided by two factors, t_R and t_{DMA}. By sensing a limited number of cells from selected BLs and transferring fewer bit values to the flash controller, SPread shortens both t_R and t_{DMA} for small reads, 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 ECC code word. As explained in Section 4.1.1, 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[3], which is small enough to minimize the read amplification by amplified reads.

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 19 depicts example cases of when multiple subpage reads happen on a single page simultaneously. In Figure 19, 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 19(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

[3]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.



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

Figure 19: Examples of parallel subpage read patterns

size. However, the problem gets more complicated when subpages are severely fragmented over the NAND page as shown in Figure 19(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. Note that recent NAND devices already support many NAND commands for advanced features, such as erase suspend, program reset, and read voltage calibration, so we do not have enough room to support many new commands. The latter one would reduce the design complexity of SPread, but it may lose a lot of optimization gains.

4.2.2 Design of SPread-Enabled NAND Device

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 providing them a *valid bitmap*. The valid bitmap indicates the information of all the demanded subpages within a NAND page. Figure 20 illustrates how the proposed SPread command works with 16-KB NAND pages and 2-KB ECC words. As shown in Figure 20, 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 BLs, while the others are inhibited. The additional selective pre-charging logic, an *SP selector* in Figure 20, does this task by simply pulling down the BLs of inhibited subpages. For SPread, the elapsed times for

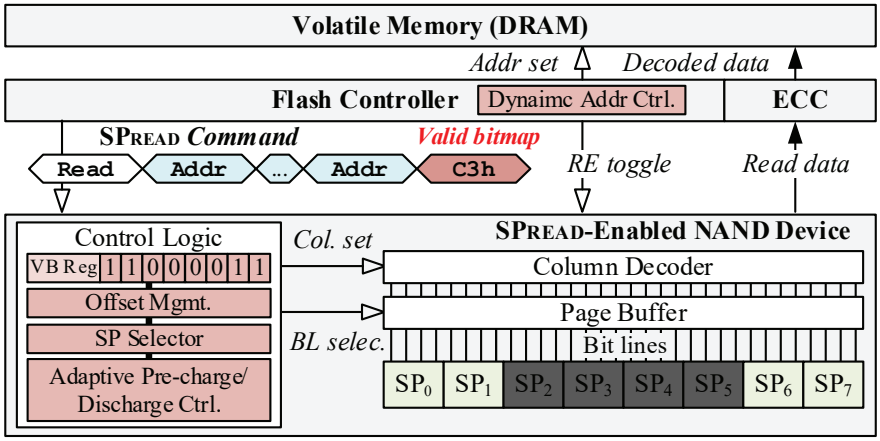


Figure 20: An operational overview of SPREAD.

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 20. 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 already maintains the valid bitmap inside, so it is able to specify the column offsets for fetching the required bytes from the page

buffer, skipping unnecessary subpages bytes. 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 (in the flash controller) as well as the DMA slave engine (in the NAND device). 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 toggling signals.

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 [33] 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.

4.2.3 SPread Latency Modeling

In order to evaluate the effectiveness of the proposed SPread, it is ideal to develop a new SPread-enabled NAND device. However, due to practical limitations in developing real NAND devices in academia, we have estimated the SPread latency t_{SPR} based on

NAND device physics using known NAND device parameter values. We assumed that our NAND devices are based on the 1x-nm process. In our latency modeling, our goal is to estimate the read latency $t_{SPR}(n)$ of SPREAD when n-KB data are read. Since few NAND flash manufacturers disclose their devices' t_R values for 16-KB reads, we relied on the known t_R for 4-KB reads [3] in our estimation.

As described in Sections 4.1.1 and 4.2.2, as with a normal page read operation, SPREAD reads the stored data through two steps, the sensing step and data transfer step. Therefore, we can represent $t_{SPR}(n)$ as follows:

$$t_{SPR}(n) = t_R(n) + t_{DMA}(n). \tag{4.1}$$

Since the sensed data are transferred one byte per RE toggle, if the NAND chip interface operates in the clock frequency f, $t_{DMA}(n)$ can be computed as follows:

$$t_{DMA}(n) = (1/f) \times n \times 10^3. \tag{4.2}$$

The sensing step can be further divided into two sub-steps, precharging BLs and discharging BLs. Therefore, we can model $t_R(n)$ as a sum of $t_{PRE}(n)$ and $t_{DISCH}(n)$. In order to model other minor steps required during a read operation, we introduce a *constant* overhead term t_C to $t_R(n)$ [34]. We assume that execution times of these minor steps are independent from the data size n. (For example, t_C includes

the time for discharging WLs, which takes a constant time.) Therefore, $t_R(n)$ can be modeled as follows:

$$t_R(n) = t_{PRE}(n) + t_{DISCH}(n) + t_C. \tag{4.3}$$

In the pre-charging phase, $t_{PRE}(n)$ is linearly proportional to the capacitance C_{BL} of a BL. C_{BL} of the pre-charged BL can be increased by about 30% if either of its adjacent BLs is not charged due to the parasitic capacitance C_{PARA} caused with the uncharged adjacent BL [35]. Since $t_{PRE}(n)$ is largely decided by whether C_{PARA} exists or not during the pre-charging phase, we distinguish $t_{PRE}(16)$ for a full-page (i.e., 16-KB) read from t^*_{PRE} for all other small-size reads. (Note that except for a 16-KB read, all other reads have the same t_{PRE}, which we denote as t^*_{PRE}.) When a full-page read is performed, since all BLs are pre-charged and no C_{PARA} occurs between adjacent BLs, $t_{PRE}(16)$ is modeled as $0.7 \times t^*_{PRE}$.

In the discharging phase, $t_{DISCH}(n)$ depends on the number of pre-charged BLs: the more BLs are charged, the longer t_{DISCH} is. This is because, as shown in Figure 16, all BLs are connected to a single common source line (CSL) *in parallel*. Therefore, how fast BLs are discharged is up to how fast a CSL can sink current I_{CSL}. Since I_{CSL} is the sum of each BL current, it is linearly proportional to the number of pre-charged BL. When $t_{DISCH}(m)$ is known, $t_{DISCH}(n)$

can be modeled as follows:

$$t_{DISCH}(n) = n/m \times t_{DISCH}(m). \tag{4.4}$$

For the latency model given above, in order to compute t^*_{PRE}, $t_{DISCH}(m)$, and t_C in 1x-nm process, we exploit the known $t_{PRE}(4) = 10\ \mu s$ in the 3x-nm process. Based on 1) a fact that C_{BL} is roughly doubled when the process is shrunk from 3x nm to 1x nm [34] and 2) $t_{PRE}(n)$ is proportional to C_{BL}, we can estimate $t_{PRE}(4)$ (i.e., t^*_{PRE}) in the 1x-nm process to be 20 μs. Since a typical ratio between $t_{PRE}(4)$ and $t_{DISCH}(4)$ is known to 1:1 (i.e., $t_{DISCH}(4) = 20\ \mu s$ in 1x-nm process) [34], using the known $t_R(4) = 45\mu s$ in 1x-nm process [3], we get 5 μs for t_C from equation (4.3). Combining equations (4.3) and (4.4), we estimate $t_R(n)$ as follows:

$$t_R(n)\ [\mu s] = \begin{cases} 99 & n = 16 \\ 25 + 5 \times n & otherwise. \end{cases} \tag{4.5}$$

Based on equations (4.2) and (4.5), we estimate $t_{SPR}(n)$ for different n's as summarized in Table 3. We assumed the I/O bus frequency 800 MB/s, typically used with 1x-nm NAND devices [3].

Table 3: 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

4.3 spFTL: SPread-Aware FTL

We have developed an SPread-aware FTL, `spFTL`, which leverages the proposed SPread command. Figure 21 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 21 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 21, logically contiguous pages may map to multiple physical pages under the FGM scheme; four subpages whose logical addresses are `F0h`, `F1h`, `F2h` and `F3h` were written together, but two of them `F1h` and `F2h` were overwritten with new data at different times

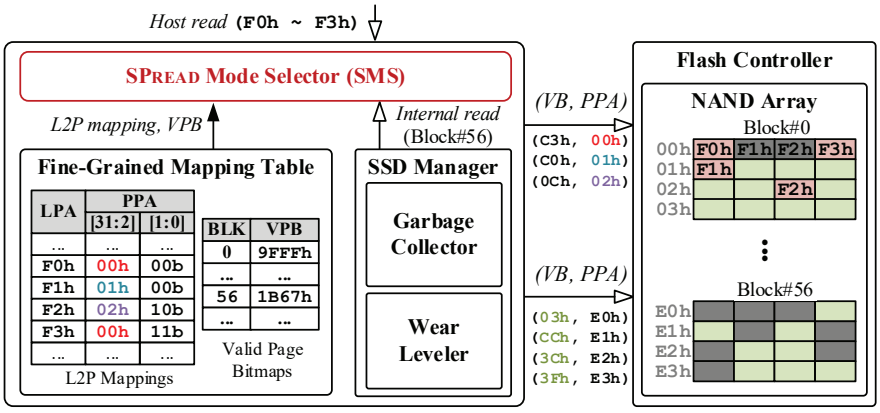


Figure 21: An organizational overview of `spFTL`.

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 21. When the garbage collector or wear-leveler issues internal reads for a target block (e.g., Block#56 in Figure 21), 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.

4.4 Experimental Results

4.4.1 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 [36]. 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 modeling [3], full-page write latency (t_{PROG}) and block erasure latency (t_{BER}) were set to 640 μs and 3.5 ms, respectively. The latency model described in Section 4.2.3 was used for modeling a 16-KB page read latency (99 μs) and SPread latencies with the other sizes.

The five distinct I/O workloads were used for our evaluation. Table 4 summaries the characteristics of the workloads in terms of a read request size and a read/write ratio. For evaluating how `spFTL` better

Table 4: Descriptions of I/O traces used for evaluations.

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

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 [37] and LinkBench [38], respectively. To evaluate the performance impact of SPread in more general applications, we used three traces from MSR-Cambridge traces [39]. I/O traces for PRJ, USR, and STG were collected from data center servers managing project directories, user home directories, and web staging, respectively. 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. As explained in Section 4.2.2, `spFTL` employed both the dynamic DMA and SPread, thereby being able to shorten t_{DMA} and t_R.

4.4.2 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 22, `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 23, `spFTL` reduced the average read latency up to 56% and 50% over `pageFTL` and `dmaFTL`,

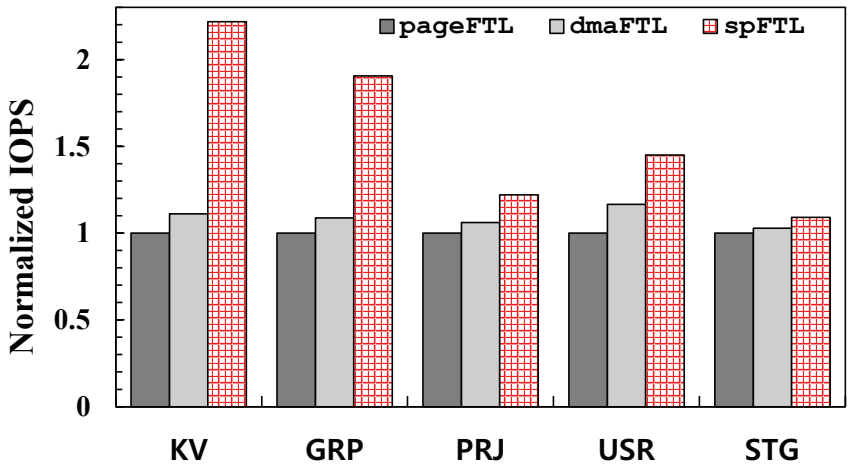


Figure 22: A comparison of normalized IOPS's.

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 6.4x longer than the read latency. However, as shown in Figure 22, `spFTL` achieved performance gains over `pageFTL` by 44% and 10% under USR and STG, respectively, despite their write ratios. As will be explained below,

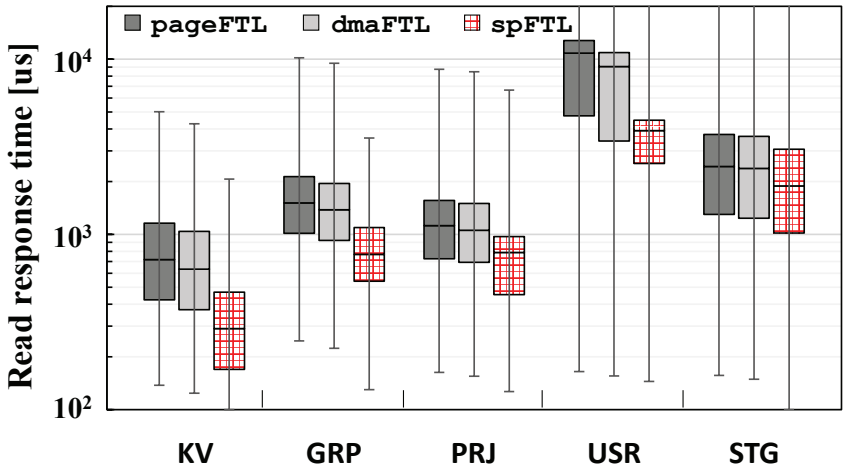


Figure 23: A comparison of read response times.

this was due to the negative effect of internal fragmentation.

To better understand how `spFTL` outperformed the other FTLs in detail, we measured RAF values in `spFTL` and `pageFTL` as shown in Figure 24. 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

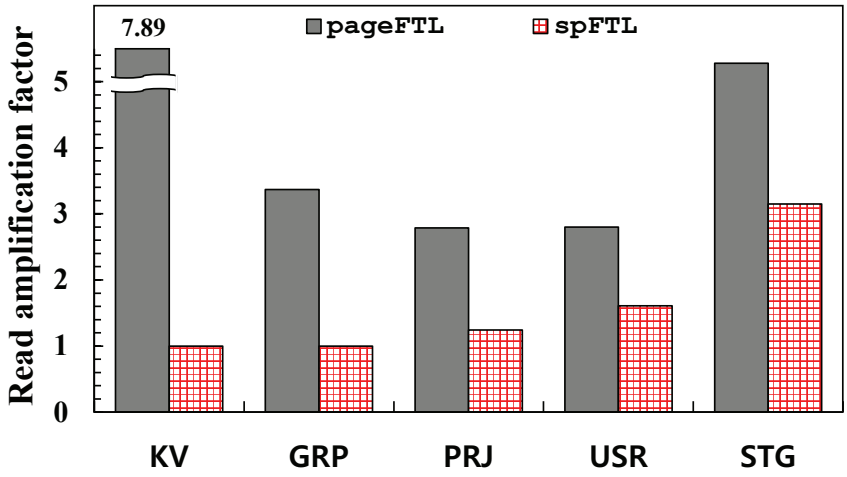


Figure 24: A comparison of read amplification factors.

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 physical 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.

Chapter 5

Lifetime Optimization Based on Integrated Compression Technique

5.1 Motivation

Increasing the number of P/E cycles itself is known to be difficult without the fundamental changes of current semiconductor technologies (e.g., new materials). For this reason, many researchers have focused on developing software/controller-level solutions that can improve the lifetime of SSDs without changing underlying devices. One of popular such solutions is based on a data reduction approach where the amount of data physically written to flash is intelligently reduced. With less amount of written data, SSD lifetimes can be improved by experiencing smaller P/E cycles for a given write traffic. Data deduplication [15], lossless compression [16], and delta compression [19] belong to this solution.

Since individual techniques mentioned above exploit different properties of incoming data, combining multiple data reduction techniques has received a lot of attention in order to achieve high data reduction efficiency over a wide range of data [40]. Although it is obvious that

the more data reduction technique are combined, the less amount data would be stored in NAND devices, how to efficiently integrate them is the most important issue because performance and resource overheads of each technique can be accumulated.

In this chapter, we propose a novel integrated data reduction technique, called dedup-assisted compression (DAC). As its name implies, DAC is based on deduplication and lossless compression, but its approach is totally different from their naive combination. DAC aims at bridging a big gap between deduplication and lossless compression. Unlike deduplication that only considers exactly matched values among blocks and lossless compression that eliminates exactly matched/repeated bit patterns within a block, DAC takes into account data similarity across entire storage space. While handling a wider spectrum of input workloads (where neither of the individual techniques can effectively deal with), our DAC-enabled storages also require negligible resources by utilizing existing hardware modules.

5.2 Overall Architecture of Target SSDs

Figure 25 depicts an overall architecture of our baseline SSD with the DAC scheme. Similar to recent high-end SSDs employing deduplication and compression, our baseline SSD has a specialized hardware controller, called DAC hardware module (DHM), which is composed of two sub-modules, a strong hash function accelerator and a compression/decompression accelerator. A simple XOR logic is only a new one

that is added to the hardware controller. In spite of architectural similarity, DHM works differently from the existing SSD controller in that it is designed to find similar references and to lower data entropy for better compression (see Section 5.3).

A DAC-aware FTL (DAC-FTL) is a new FTL design for DAC. It handles read and write requests from the host as usual FTLs, but its primary goal is to reduce write traffic by utilizing DHM. DAC-FTL maintains a fingerprint store for efficient search of similar data, in addition to an enhanced mapping table. Two buffers, reference and write buffers, are required for fast read operations and for preventing internal fragmentation, respectively (see Section 5.4).

5.3 DAC: Dedup-Assisted Compression

In this section, we describe the mechanism of the proposed dedup-assisted compression algorithm, particularly focusing on its two unique features, (i) reference search and (ii) compression with a reference.

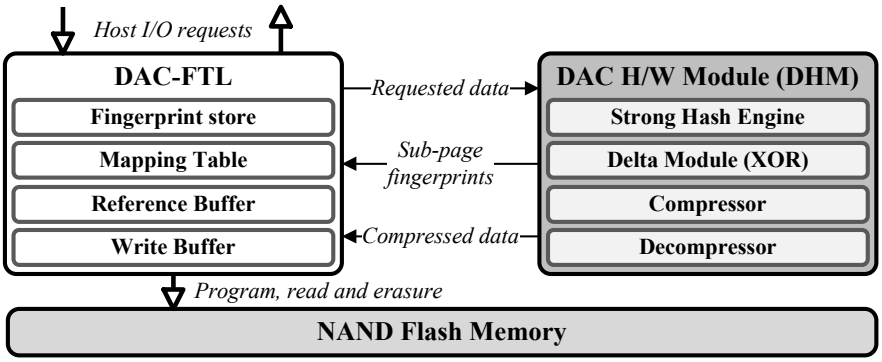


Figure 25: An organizational overview of our target SSDs.

5.3.1 Reference Data Search

Existing deduplication techniques aim at finding reference data that are exactly matched to requested data. To this end, a strong hash algorithm is widely used to generate a unique fingerprint for a specific bit pattern with an extremely low collision rate. One important property of the strong hash algorithm is that it creates very different fingerprints with a long distance if bit patterns are similar. This is a desirable characteristic for deduplication and security applications, but it is undesirable for DAC because our goal is to find a similar reference. That is, a fingerprint from a strong hash engine is not a feasible metric to decide similarity among candidates.

There is another category of hash functions like locality-sensitive hashing [41, 42] that creates fingerprints with a short distance if bit patterns of data are similar. Using such hash functions is beyond the scope of this study because we want to maximally reuse existing hardware modules with minimal changes at the controller level, so as to provide good compatibility with existing SSD controllers. However, the adoption or the development of new hash algorithms suitable for DAC would be interesting future work.

Instead of using or developing new hash functions, DAC makes use of existing strong hash functions for similarity detection. DAC splits a flash page into several sub-pages. For example, if a page size and a sub-page size are assumed to be 16 KB and 4 KB, respectively, there are four sub-pages per page. Using a strong hash function, it then

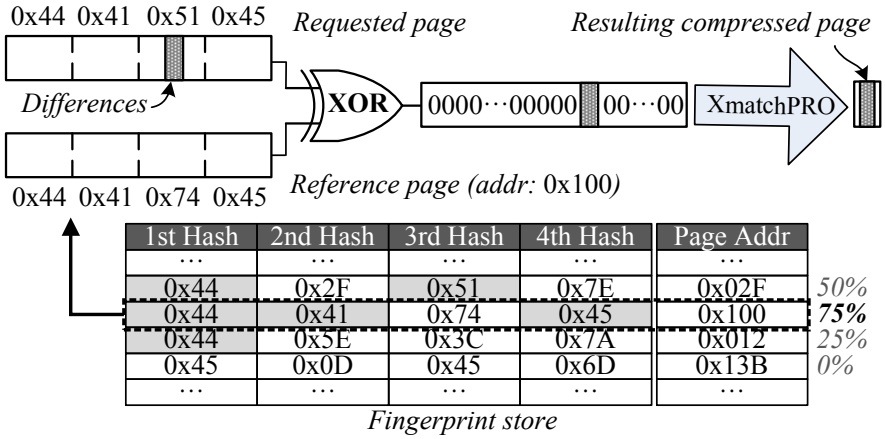


Figure 26: The proposed dedup-assisted compression mechanism.

calculates fingerprints of all the sub-pages belonging to a requested page. To find a similar reference, DAC compares four fingerprints with those in a fingerprint store which keeps fingerprints of all the candidates. If there is a candidate page whose fingerprints are all matched, its content is identical to the requested one, showing the best similarity. If nothing is matched, it means that there are no similar references. As a result, based on the number of fingerprints matched, DAC estimates the degree of similarity. As expected, among all the candidates, the most similar one is selected as a reference.

Figure 26 shows an example of how DAC finds a similar reference. The sub-page fingerprints of a requested page are `0x44`, `0x41`, `0x51`, and `0x45`, respectively. Using those values, DAC searches for the most similar one in the fingerprint store. The page `0x100` is selected as a reference page because it has the largest number of matched sub-pages.

The fingerprint store contains a large number of candidates, so searching for a similar reference by comparing fingerprints would take a very long time. This issue has been intensively investigated by previous deduplication studies, and they showed that reference search can be quickly completed in a constant time. Only the difference between conventional deduplication and DAC is that DAC requires more than one fingerprint comparison. However, the impact of additional fingerprints search on performance is actually negligible due to long I/O latencies of NAND flash.

5.3.2 Data Compression with a Reference

Once a similar reference is found, DAC conducts XORing between a reference page and a requested page. This can be done quickly by using hardware-based XOR logics. As depicted in Figure 26, the reference and the requested pages are very similar, so resulting data mostly has '0', exhibiting extremely low data entropy. Only the exception is a narrow range of region where different bit patterns from the reference are observed. For such data which consists of a series of '0' values, even a simple lossless compression algorithm exploiting run-length encoding can achieve an extremely high compression ratio.

While any lossless algorithms can be used for DAC, we select the XmatchPRO algorithm [43] in this study, which takes advantage of both the run-length and the dictionary-based data encoding. The run-length encoding feature of XmatchPRO is effective to compress successive bit patterns. In the case where similar references are found

as shown in Figure 26, it converts several kilobytes of data to few bytes. XmatchPRO also makes use of the dictionary-based encoding, so it achieves a fairly good compression ratio even for data which has no similar reference. In our observation, XmatchPRO exhibits comparable performance as other LZ-variant algorithms for small size data like 4-16 KB flash pages.

5.4 Design of DAC-Aware FTL

5.4.1 Request Handling in DAC-FTL

DAC-FTL is responsible for handling incoming read and write requests, performing data compression by communicating with the DHM controller. Figure 27 shows request handling procedures of DAC-FTL. When a write request arrives, DAC-FTL passes the requested data to DHM to get fingerprints for four sub-pages using the strong hash accelerator of DHM, as described in Section 5.3. If there is no reference page, it merely compresses the requested data without a reference using the DHM's XmatchPRO accelerator. If a similar reference is found, DAC-FTL reads a reference page from a flash array and then delivers it to DHM. If the reference is compressed, DHM decompresses it immediately. Note that decompressing the reference data does not require additional reference reads because DAC-FTL never allows a page compressed with a reference to be used as a reference for other pages. It helps us avoid reading another reference page for the current reference page. By XORing the uncompressed refer-

ence page and the requested page, DAC gets the low-entropy data, which is then compressed by DHM.

The size of the compressed data is usually smaller than that of a flash page, so writing it to flash directly causes a fragmentation problem, wasting valuable flash space. For this reason, DAC-FTL temporally keeps the compressed data on a write buffer whose capacity is the same as a flash page. When the write buffer becomes full, DAC-FTL flushes the buffered data to flash at once. It would be possible that the compressed data becomes larger than its original size (e.g., if its data entropy is so high with no reference page). In that case, DAC-FTL writes the original (i.e., uncompressed) data to flash directly.

In DAC-FTL, multiple logical pages are packed together to a single physical page. Therefore, a conventional mapping table that maps one logical page to one physical page, i.e., one-to-one mapping, does not work with DAC-FTL. Instead, many-to-one mapping where multiple logical pages point to a physical page is required. Each entry of the mapping table may need to have a physical page number for a logical page, the offset of the logical page on the physical page and the length of compressed data. Maintaining this information in the mapping table requires a large DRAM space. Thus, DAC-FTL keeps all the information about logical pages at the beginning of a physical page where they are stored. This header information is automatically fetched while reading a compressed page, so additional read operations for headers are not necessary.

Updating the fingerprint store is the final step of a write process.

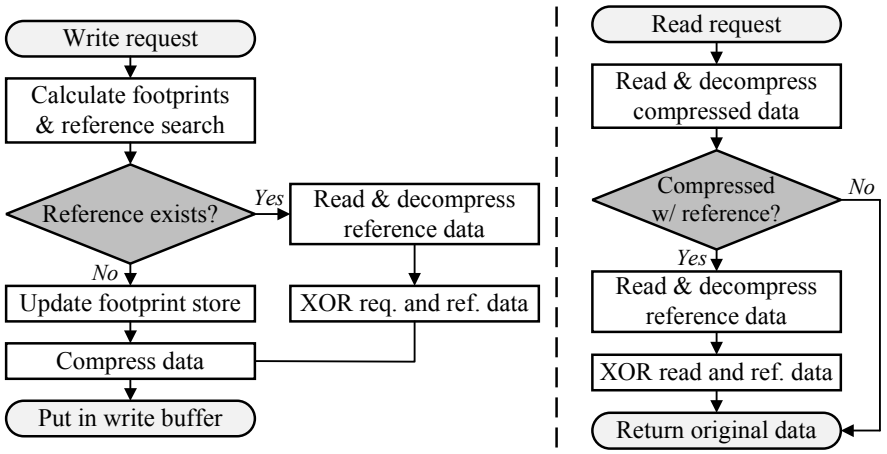


Figure 27: Request handling procedures of DAC-FTL.

DAC-FTL adds the fingerprints of the requested data to the fingerprint store *only when* it is compressed without a reference. This is a reasonable choice for the following two reasons; (1) if the requested data has no reference, it means that the contents of that data is unique and similar data was not written before; (2) as we briefly mentioned before, it avoids us to read another reference page when it is selected as a reference page for incoming data in the future.

For a read request, DAC-FTL first reads a physical page from the flash array. If the read data is not compressed, it returns to the host immediately (not shown in Figure 27). If the data is compressed without a reference, DAC-FTL performs decompression using DHM and delivers uncompressed data to the host. Finally, if the data are compressed with reference data, DAC-FTL reads a reference from flash and decompresses it. Then, the uncompressed requested is XORed with the decompressed reference. The resulting data is sent to the

host.

5.4.2 Overhead Mitigation in DAC-FTL

DAC-FTL may require a large amount of memory for the fingerprint store. Strong hash functions commonly employed for data deduplication generates tens of bytes hash values per page (e.g., 16 bytes for MD5 [44]). If we assume to maintain all the fingerprints for 4 KB pages in 512 GB flash storage, the memory requirement is 2 GB only for fingerprints. Moreover, since DAC-FTL should maintain four fingerprints for each page, the amount of memory for the fingerprint store increases to 8 GB, which is infeasibly large.

DAC-FTL takes two approaches to reduce the memory requirement. Firstly, DAC-FTL uses the first 1/4 of 16 bytes fingerprints from MD5. It may result in more hash collisions, increasing a probability that different data could have the same fingerprint. However, data losses never occur because DAC does not perform data deduplication, always conducting lossless compression even when there is a reference page with all the matched fingerprints. It may decrease the potential benefit of DAC by storing redundant data which can be skipped with deduplication. However, its impact on overall write traffic reduction is negligible since every 4 KB page can be compressed to only 24 Bytes (about 0.5% of the original size) by DAC if it has a reference page whose data is exactly the same as the requested data. In order to further reduce the memory overhead, secondly, DAC-FTL maintains only a limited number of fingerprints in DRAM and manages them

in an LRU fashion. This decision is made based on our observation that references to fingerprints have a very skewed distribution with high temporal locality; less than 5% of written pages are being used as reference pages for more than 80% of requested pages. Consequently, its effect on a compression rate is negligible.

Unlike data deduplication and lossless compression, DAC-FTL requires to read reference pages for compression and decompression of the requested page. Those additional reads could badly affect overall I/O performance, especially for read requests from host; DAC-FTL requires two additional reference page reads to service a single page request, which doubles read latencies. To mitigate read overheads, DAC-FTL employs a read buffer whose size is 32 MB that caches popular references. Since reference data has high temporal locality mentioned before, many page reads for reference data can be served from a small read buffer.

5.5 Experimental Results

5.5.1 Experimental Settings

In order to evaluate the effectiveness of the proposed `DAC`, we have implemented DAC-FTL on FlashBench [25], which is a storage emulation environment of flash-based SSDs. The SSD emulator was 512 GB with eight channels with eight ways, and its page size was 4 KB and the number of pages per block was 128. Hardware modules for a strong hash function (MD5) and lossless compression (XmathPRO)

were emulated by software in FlashBench.

For our evaluation, we used four block I/O traces collected from a high-end PC in various scenarios. All the traces included actual data as well. PC recorded I/O activities in general PC usages (e.g., documenting, installing programs, etc.), and SYNTH was collected from hardware synthesizing procedures. WEB and IOT captured I/O activities while browsing World-Wide-Web and storing data generated from IOT sensor devices, respectively.

We compared five different SSD configurations with different data reduction techniques. For `Dedup` and `Comp`, the FTL performed either deduplication or lossless compression, respectively. In `Dedup+Comp`, deduplication was first performed and lossless compression was applied if deduplication failed. `DAC` compressed requested data as described in Section 5.4. The size of the fingerprint store was set to keep 0.5% of all the fingerprints. For example, the fingerprint store manages only 5K fingerprints in an LRU fashion for a trace with 1M fingerprints. $\texttt{DAC}_{\texttt{unlimited}}$ was identical to `DAC` except its fingerprint store was unlimited.

5.5.2 Evaluation Results

Figure 28 shows the amount of written pages for each trace under five different configurations. Figure 28 is normalized to `Dedup`. In terms of a data reduction ratio, `DAC` outperforms the other SSD configurations for every trace. Even compared with `Dedup+Comp` that employs both deduplication and lossless compression, `DAC` reduces write

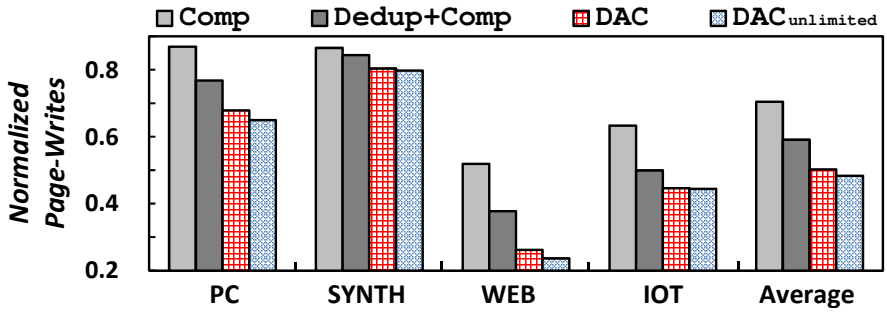


Figure 28: A comparison of normalized page-writes for different configurations of data reduction techniques.

traffic by up to 30% and by 15% on average. This result clearly shows that DAC finds similar data patterns more efficiently than the naive combination of deduplication and lossless compression, and prevents them from being written to SSDs. $\texttt{DAC}_{\texttt{unlimited}}$ shows better performance than DAC, further reducing the amount of written data by lower than 2% on average. However, this benefit is not so attractive, considering its high memory consumption. DAC only requires 0.5% of that $\texttt{DAC}_{\texttt{unlimited}}$ requires for the fingerprint store, but exhibits similar write traffic reductions, achieving excellent lifetime improvement as well.

In order to better understand how the proposed DAC effectively reduces the write traffic, we additionally measured 1) the number of 4-KiB host writes compressed with references and 2) the size of compressed data with DAC and the normal lossless-compression algorithms. Table 5 shows the portion of host writes compressed with references and their average compression ratio with DAC and normal

Table 5: Portion of DAC-compressed host writes and their average compression ratio.

		PC	SYNTH	WEB	IOT
DAC-compressed Portion		0.41	0.18	0.97	0.63
Avg. Comp. ratio	DAC	0.066	0.031	0.049	0.031
	normal	0.32	0.17	0.15	0.54

compression. As shown in Table 5, except for SYNTH, a number of host writes were compressed with references in DAC-FTL, and their resulting data were significantly reduced by DAC over the normal compression algorithm. In particular, DAC-FTL performed compressions with references on 97% of incoming data, and further reduced the write traffic on NAND devices by 67% over the normal compression.

Finally, we evaluated the effect of a read buffer on read performance. Figure 29 shows the number of additional reads for reference pages with different buffer sizes ranging from 4 MB to 64 MB. This graph is normalized by that of DAC with no read buffer. As shown in

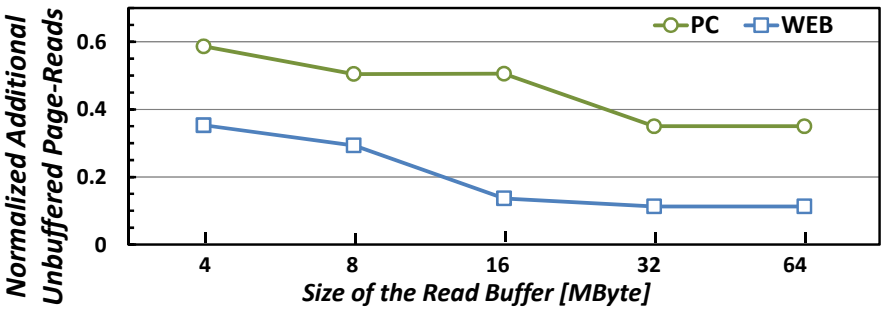


Figure 29: A comparison of normalized additional unbuffered page-reads for varying size of read buffer.

Figure 29, the use of a read buffer is effective in reducing reference reads; only with a 64-MB reference buffer, extra reads are reduced by more than 60% (for PC) and 90% (for WEB), respectively. Note that even though almost incoming data are compressed with references in WEB, DAC-FTL effectively reduced additional reads by leveraging the high localities of reference data.

Chapter 6

Conclusions

6.1 Summary

NAND flash-based storage systems have been widely adopted in modern computing systems, owing to the continuous reduction of the cost-per-bit value of NAND flash memory. Contrary to the improvement of the device density, the performance and lifetime of NAND devices have been continuously degraded by aggressive capacity-oriented design decisions such as semiconductor process scaling, multi-leveling, and large operation units. In order for NAND flash-based storage systems to satisfy the high-performance and long-lifetime requirement of modern computing systems as well as the large-capacity requirement, this performance problem should be properly addressed.

In this dissertation, we proposed several system-level techniques which aim at alleviating the performance and lifetime degradation originated from capacity-oriented design decisions. First, we presented a new page-program sequence for MLC NAND flash memory, called RPS, which opens new optimization opportunities to upper system layers. By relaxing unnecessary constraints of the existing program order, the RPS scheme allows an FTL to flexibly use heterogeneous MLC pages, thus effectively overcoming the limitations of existing optimiza-

tion techniques for MLC NAND flash-based storage systems. We validated the proposed RPS scheme using state-of-the-art 3D NAND flash chips, and our validation results show that the RPS scheme can provide the same level of NAND reliability over the existing FPS scheme.

Based on the RPS scheme, we designed `flexFTL`, which fully exploits the performance asymmetry between MLC NAND pages to improve the effective write-performance of MLC NAND flash-based storage systems. It rapidly handles intensive host writes using fast LSB pages, while using MSB pages to serve host writes sporadically issued. By doing so, `flexFTL` can better meet varying write-performance requirements of host systems without sacrificing the storage lifetime. In addition, the 2PO scheme combined with the per-block parity-page backup scheme enables the `flexFTL` to remove most overheads of paired LSB-page backup operations, thus improving both the performance and lifetime of MLC NAND flash-based storage systems.

In order to evaluate the effectiveness of `flexFTL`, we implemented `flexFTL` in an open develop/evaluation platform, and evaluated it using various enterprise workloads. 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.

Second, we introduced a novel design of large-page NAND flash-based storage systems. We proposed a new NAND read operation, called SPread, which allows us to selectively reads only desired sub-pages in parallel, thus significantly mitigating the amplified read problem in large-page NAND flash-based storage systems. By slightly ex-

tending the existing data path and control of NAND read operations, SPread can quickly handle small reads without read amplification. In order to take advantages of SPread-enabled NAND devices at the storage level, we developed an SPread-aware FTL, `spFTL`, which effectively leverages SPread in dealing with internal reads in GC procedures as well as host reads.

In order to evaluate the effectiveness of the proposed `spFTL`, we carried out a simulation study using various workloads from a variety of server systems. Our evaluation results show that `spFTL` can increase the IOPS and read latency by up to 122% and 56%, respectively. Moreover, our `spFTL` is able to improve overall I/O performance even under workloads with lots of writes, by increasing the GC efficiency.

Finally, we proposed a new integrated write-traffic reduction technique, called DAC. With a synergetic integration of individual techniques, DAC maximally reduced the write traffic to NAND devices by effectively reusing hardware resources of individual techniques, thus requiring trivial hardware and memory resources for integration. We evaluated the effectiveness of DAC using content-included workloads collected in various real-world computing environments. Our evaluation results showed that showed that the proposed scheme achieved up to 30% higher data reduction ratio even over a combination of existing data reduction techniques.

6.2 Future Work

6.2.1 Development of Optimization Techniques for 3D NAND Flash Memory

3D NAND flash technology [4, 24, 45] has been recently proposed, and widely adopted as a key enabling technique for the continuous growth in the capacity of NAND devices. By stacking memory cells vertically, 3D NAND flash memory can provide high device density without aggressive semiconductor process scaling, thus minimizing the performance and lifetime degradation due to small manufacturing process technology nodes under 20-nm [46]. However, 3D NAND devices are also expected to eventually suffer from the performance and lifetime degradation as capacity-oriented design decisions are more aggressively adopted to meet high storage requirements on the capacity in modern computing systems. For example, most NAND manufacturers have been introducing 4-bit MLC (i.e., quad-level cell (QLC)) NAND devices [47, 48, 49], and using a manufacturing process technology node as small as 20 nm [50].

In order to address the performance and lifetime problems in 3D NAND devices, we will investigate new physical characteristics of 3D NAND flash memory and develop various optimization techniques aware of them, as we have done for planar NAND devices in this dissertation. Since 3D NAND flash memory has a different cell design and a different chip organization over 2D NAND flash memory, the physical characteristics of 3D NAND flash memory are quite different from

those of 2D NAND flash memory. For example, due to the vertical etching process, the program latency of a wordline varies depending on the wordline's location within vertical layers [51]. Furthermore, in 3D NAND devices with charge trap (CT)-type cell structures [46], the cell-to-cell interference has little impact on the NAND reliability, thus allowing pages in a block to be used in a much more flexible fashion over 2D NAND devices. Therefore, the effective write performance of 3D NAND flash-based storage systems can be improved in a similar manner over the adaptive page allocation techniques proposed in Section 3.3.2.

6.2.2 System-Level Extensions of Proposed Techniques

In this dissertation, we have proposed several optimization techniques that exploit the physical characteristic of underlying NAND devices at the storage firmware level. Although the proposed techniques can achieve significant performance and lifetime gains over existing techniques, we can further improve their effectiveness by vertically integrating the optimization hints from different storage system layers.

First, we can further increase performance gains of the current version of `flexFTL` by leveraging host-side hints about future write requests. The page the adaptive page allocator introduced in Section 3.3.2 is using a rather simple heuristic in estimating the performance requirement of host systems to choosing a page type for given requests. If `flexFTL` can more accurately estimate the amount of fu-

ture burst writes, for example, by using a page cache-based future write predictor [52], a background garbage collector can reclaim free blocks more efficiently so that fast LSB pages can be maximally used for future write requests.

The read performance of large-page NAND flash-based storage systems can be further improved as well, by making upper system layers to manage data blocks in SPread-aware fashion. In order to effectively handle small writes, modern file systems and applications are typically using data blocks larger than the minimum size SPread supports, issuing large requests for reading small data. However, the small write problem can be addressed by issuing multiple requests in a batched manner even when a finer-granularity data block is used. We have a plan to develop a new data block management technique at applications and file systems, that efficiently decouples read and write units so as to better meet the high performance requirements for small reads while not aggravating the small write problem.

6.2.3 Leveraging Locality-Sensitive Hash to Maximize Write Traffic Reduction

In order to further reduce write traffic to NAND devices over a naive combination of deduplication and lossless compression techniques, DAC additionally leverages the similarity between incoming data and the previously stored. As explained Section 5.3.1, DAC maintains partial fingerprints of stored data in a subpage unit so as to find the most similar data that have the largest number of matched sub-

pages. However, since it uses partial fingerprints based on a strong hash function which is designed to find *exactly the same* data, an actually similar data cannot be used as a reference if the data are slightly different in multiple subpages.

In such a case, it would be more effective for DAC to exploit another hash functions such as locality-sensitive hashing (LSH) [41, 42] that generates the same hash values if two sets of input data have similar bit patterns. Since LSH families does not explicitly compute the intersection and union of target data sets, they allow us to find a reference of incoming data, even if the reference has no subpages matched to those of the incoming data. In our preliminary experiments, DAC was able to find more reference data for incoming writes with a simple LSH function, thus further reducing write traffic to NAND devices over when using a strong hash function.

However, there is a technical challenge to employ LSH functions in DAC, since most LSH functions introduce significant computational overheads over the strong hash function currently used. If we use a light-weight LSH function to reduce the computation overheads, the number of hash collisions significantly increases, and it rather decreases the compression efficiency by compressing incoming data with a reference which is not actually similar. To address this, we have a plan to explore various LSH algorithms to find the most effective in DAC, and design a new hardware acceleration module to minimize its performance overheads.

Bibliography

[1] D. C. Daly, L. C. Fujino, and K. C. Smith, "Through the Looking Glass - The 2018 Edition: Trends in Solid-State Circuits from the 65th ISSCC," *IEEE Solid-State Circuits Magazine*, vol. 10, no. 1, pp. 30–46, 2018.

[2] A. L. Shimpi, "Micron Announces 16nm 128Gb MLC NAND, SSDs in 2014," https://www.anandtech.com/show/7147/micron-announces-16nm-128gb-mlc-nand-ssds-in-2014.

[3] S. Lee, J.-Y. Lee, I.-H. Park, J. Park, S.-W. Yun, M.-S. Kim, J.-H. Lee, M. Kim, K. Lee, T. Kim, B. Cho, D. Cho, S. Yun, J.-N. Im, H. Yim, K.-H. Kang, S. Jeon, S.-Jo, Y.-L. Ahn, S.-M. Joe, S. Kim, D.-K. Woo, J. Park, H.-W. Park, Y. Kim, J. Park, Y. Choi, M. Hirano, J.-D. Ihm, B. Jeong, S.-K. Lee, M. Kim, H. Lee, S. Seo, H. Jeon, C.-H. Kim, H. Kim, J. Kim, Y. Yim, H. Kim, D.-S. Byeon, H.-J. Yang, K.-T. Park, K.-H. Kyung, and J.-H. Choi, "A 128Gb 2b/Cell NAND Flash Memory in 14nm Technology with tPROG=640μs and 800MB/s I/O Rate," in *Proceedings of the IEEE International Solid-State Circuits Conference* (ISSCC), 2016.

[4] D. Kang, W. Jeong, C. Kim, D.-H. Kim, Y. S. Cho, K.-T. Kang, J. Ryu, K.-M. Kang, S. Lee, W. Kim, H. Lee, J. Yu, N. Choi, D.-S. Jang, J.-D., Ihm, D. Kim, M.-S. Kim, A.-S. Park, J.-I. Son,

I.-M. Kim, P. Kwak, B.-K. Jung, D.-S. Lee, H. Kim, H.-J. Yang, D.-S. Byeon, K.-T. Park, K.-H. Kyung, and J.-H. Choi, "256gb 3b/cell V-Nand Flash Memory with 48 Stacked WL Layers," in *Proceedings of the IEEE International Solid-State Circuits Conference* (ISSCC), 2016.

[5] J. Lee and D. Shin, "Adaptive Paired Page Prebackup Scheme for MLC NAND Flash Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 7, pp. 1110–1114, 2014.

[6] S. Lee and J. Kim, "Improving Performance and Capacity of Flash Storage Devices by Exploiting Heterogeneity of MLC Flash Memory," *IEEE Transactions on Computers*, vol. 63, no. 10, pp. 2445–2458, 2014.

[7] F. Roohparvar, "Single Level Cell Programming in a Multiple Level Cell Non-Volatile Memory Device," In Unite States Patent, Number 7,366,013, Arpril 2008.

[8] L. M. Grupp, J. D. Davis, and S. Swanson, "The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs," in *Proceedings of the USENIX Annual Techinal Conference* (ATC), 2013.

[9] C.-C. Ho, Y.-M. Chang, Y.-H. Chang, and T.-W. Kuo, "An SLC-Like Programming Scheme for MLC Flash Memory," *ACM Transactions on Storage*, vol. 14, no. 1, article 11, 2018.

[10] M. Kim, J. Lee, S. Lee, J. Park, and J. Kim, "Improving Performance and Lifetime of Large-Page NAND Storages Using Erase-Free Subpage Programming," in *Proceedings of the Annual Design Automation Conference* (DAC), 2017.

[11] X. Zhang, J. Li, and H. Wang, "Reducing Solid-State Storage Device Write Stress through Opportunistic In-Place Delta Compression," in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2017.

[12] H. Liu, and H. H. Huang, "Graphene: Fine-Grained IO Management for Graph Computing," in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2017.

[13] S. S. Hahn, S. Lee, C. Ji, L.-P. Chang, I. Yee, L. Shi, C. J. Xue, and J. Kim, "Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter," in *Proceedings of the USENIX Annual Techinal Conference* (ATC), 2017.

[14] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging Value Locality in Optimizing NAND Flash-Based SSDs," in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2017.

[15] F. Chen, T. Luo, and X. Zhang, "CAFTL: A Context-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives," in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2011.

[16] S. Lee, J. Park, K. Fleming, Arvind, and J. Kim, "Improving Performance and Lifetime of Solid-State Drives Using Hardware-Accelerated Compression," *IEEE Transactions on Consummer Electronics*, vol. 57, no. 4, pp 1732–1739, 2011.

[17] K. Yim, H. Bahn, and K. Koh, "A Flash Compression Layer for Smartmedia Card Systems," *IEEE Transactions on Consummer Electronics*, vol. 50, no. 1, pp 192–197, 2004.

[18] T. Park, and J.-S. Kim, "Compression Support for Flash Translation Layer," in *Proceedings of the International Workshop on Software Suppot for Portable Stroage* (IWSSPS), 2010.

[19] G. Wu, and X. He, "Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality," in *Proceedings of the ACM European Conference on Computer Systems* (EuroSys), 2012.

[20] G. Naso, L. Botticchio, M. Castelli, C. Cerafogil, M. Cichocki, P. Conenna, A. D'Alessandro, L. De Santis, D. Di Cicco, W. Di Francesco, M. L. Gallese, G. Gallo, M. Incarnati, C. Lattaro, A. Macerola, G. marotta, V. Moschiano, D. Orlandi, F. Paolini, S. Perugini, L. Pilolli, P. Pistilli, G. Rizzo, F. Rori, M. Rossini, G. Santin, E. Sirizotti, A. Smaniotto, U. Siciliani, M. Tiburzi, R. Meyer, A. Goda, B. Filipiak, T. Vali, M. Helm, and R. Ghodsi, "A 128Gb 3b/Cell NAND Flash Design Using 20nm Planar-Cell Technology," in *Proceedings of the IEEE International Solid-State Circuits Conference* (ISSCC), 2013.

[21] C. Kim, J. Ryu, T. Lee, H. Kim, J. Lim, J. Jeong, S. Seo, H. Jeon, B. Kim, I. Lee, D. Lee, P. Kwak, S. Cho, Y. Yim, C. Cho, W. Jeong, J.-M. Han, D. Song, K. Kyung, Y.-H. Lim, and Y.-H. Jun. "A 21 nm High Performance 64 Gb MLC NAND Flash Memory with 400 MB/s Asynchronous Toggle DDR Interface," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 4, pp. 981–989, 2012.

[22] H.-W. Tseng, L. Grupp, and S. Swanson, "Understanding the Impact of Power Loss on Flash Memory," in *Proceedings of the Annual Design Automation Conference* (DAC), 2011.

[23] K.-T. Park, M. Kang, D. Kim, S.-W. Hwang, B. Y. Choi, Y.-T. Lee, C. Kim, and K. Kim, "A Zeroing Cell-to-Cell Interference Page Architecture with Temporary LSB Storing and Parallel MSB Program Scheme for MLC NAND Flash Memories," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 4, pp. 919–928, 2008.

[24] J. Im, W. Jeong, D. Kim, S. Man, D. Shim, M. Choi, H. Yoon, D. Kim, Y. Kim, H. W. Park, D. Kwak, S. Park, S. Yoon, W. Hahn, J. Ryu, S. Shim, K. Kang, S. Choi, J. Ihm, Y. Min, I. Kim, D. Lee, J. Cho, O. Kown, J. Lee, M. Kim, S. Joo, J. Jnag, S. Hwang, D. Byeon, H. Yang, K. Park, K. Kyung, and J. Choi, "A 128Gb 3b Cell V-NAND Flash Memory with 1Gb/s I/O Rate," in *Proceedings of the IEEE International Solid-State Circuits Conference* (ISSCC), 2015.

[25] S. Lee, J. Park, J. Kim, "FlashBench: A Workbench for a Rapid Development of Flash-Based Storage Devices," in *Proceedings of the IEEE International Symposium on Rapid System Prototyping* (RSP), 2012.

[26] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "BlueDBM: An Appliance for Big Data Analytics," in *Proceedings of the Annual International Symposium on Computer Architecture* (ISCA), 2015.

[27] Sysbench, http://github.com/akopytov/sysbench.

[28] Filebench, http://filebench.sourceforge.net.

[29] K. Zhao, W. Zhao, H. Sun, T. Zhang, X. Zhang, and N. Zheng, "LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives," in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2013.

[30] L. M. Grupp, J. D. Davis, and S. Swanson, "The Bleak Future of Nand Flash Memory," in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2012.

[31] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, pp 53–64, 2012.

[32] Fio-Flexible I/O Tester Synthetic Benchmark, https://github.com/axboe/fio.

[33] JEDEC Standard JEDS230, "NAND Flash Interface Interoperability," JEDEC Solid State Technology Association, 2014.

[34] R. Micheloni, L. Crippa, and A. Marelli, *Inside NAND Flash Memories*, Springer, 2010.

[35] J. Brewer, and M. Gill, *Nonvolatile Memory Technologies with Emphasis on Flash*, IEEE Press, 2008.

[36] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind, "Application-Managed Flash," in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2016.

[37] RocksDB Git Repository, https://github.com/facebook/rocksdb.

[38] T.G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: a Database Benchmark Based on the Facebook Social Graph," in *Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD), 2013.

[39] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies* (FAST), 2008.

[40] S. Lee, T. Kim, J. Park, and J. Kim, "An Integrated Approach for Managing the Lifetime of Flash-Based SSDs," in *Proceedings of the Conference on Design, Automation and Test in Europe* (DATE), 2013.

[41] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *Proceedings of the Very Large Database Conference* (VLDB), 1999.

[42] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," in *Proceedings of the Annual ACM Symposium on Theory of Computing* (STOC), 1998.

[43] J. L. Nunez and S. Jones, "The X-MatchPRO 100 Mbytes/second FPGA-Based Lossless Data Compressor," in *Proceedings of the Conference on Design, Automation and Test in Europe* (DATE), 2000.

[44] R. Rivest, "The MD5 Message-Digest Algorithm," *RFC Editor*, 1992.

[45] K. Park, S. Nam, D. Kim, P. Kwak, D. Lee, Y. Choi, M. Choi, D. Kwak, D. Kim, M. Kim, H. W. Park, S. Shim, K. Kang, S. Park, K. Lee, H. Yoon, K. Ko, D. Shim, Y. Ahn, J. Ryu, D. Kim, K. Yun, J. Kwon, S. Shin, D. Byeon, K. Choi, J. Han, K. Kyung, J. Choi, and K. Kim, "Three-Dimensional 128 Gb MLC Vertical NAND Flash Memory with 24-WL Stacked Lay-

ers and 50 MB/s High-Speed Programming," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 204–213, 2015.

[46] Sansung Electronics Co., Ltd., "Next generation Samsung 3 Bit V-NAND Technology," https://www.samsungforpartners.com/pdf/resources/3bit_V-NAND_technology_White_Paper-1.pdf, white paper, 2015.

[47] N. Shibata, K. Kanda, T. Shimizu, J. Nakai, O. Nagao, N. Kobayashi, M. Miakashi, Y. Nagadomi, T. Hashimoto, H. Date, M. Sato, T. Nakagawa, H. Takamoto, J. Zhou, R. Tachibana, T. Takagiwa, T. Sugimoto, M. Ogawa, Y. Ochi, K. Kawaguchi, M. Kojima, T. Ogawa, T. Hashiguchi, R. Fukuda, M. Masuda, K. Kawakami, T. Someya, Y. Kajitani, Y. Matsumoto, N. Morozumi, J. Sato, N. Raghunathan, Y. L. Koh, S. Chen, J. lee, H. Nush, H. Sugawara, K. Hosono, T. Hisada, T. Kaneko, and H. Nakamura, "A 1.33 Tb 4-Bit/Cell 3D-Flash Memory on a 96-Work-Line-Layer Technology," in *Proceedings of the IEEE International Solid-State Circuits Conference* (ISSCC), 2018.

[48] B. Tallis, "The Samsung 860 QVO (1 TB, 4 TB) SSD Review: First Consumer SATA QLC," https://www.anandtech.com/show/13633/the-samsung-860-qvo-ssd-review, 2018.

[49] A. Shilov, "SK Hynix Begins Sampling of 96-Layer 1 Tb 3D QLC NAND," https://www.anandtech.com/show/14318/sk-hynix-begins-sampling-of-96-layer-1-tb-qlc-nand, 2019.

[50] TechInsights, "Intel/Micron 64L 3D NAND Analysis," https://www.techinsights.com/blog/intelmicron-64l-3d-nand-analysis, 2017.

[51] S.-H. Chen, Y.-T. Chen, H.-W. Wei, and W.-K. Shih, "Boosting the Performance of 3D Charge Trap NAND Flash with Asymmetric Feature Process Size Characteristic," in *Proceedings of the Annual Design Automation Conference* (DAC), 2017.

[52] S. S. Hahn, S. Lee, J. Kim, "To Collect or Not to Collect: Just-in-Time Garbage Collection for High-Performance SSDs with Long Lifetimes," in *Proceedings of the Annual Design Automation Conference* (DAC), 2015.

초 록

반도체 공정의 미세화, 다치화 기술, 큰 입출력 단위로 대표되는 용량 중심 설계 기술은 낸드 플래시 메모리의 단위 용량 당 가격을 지속적으로 감소시킴으로써 다양한 컴퓨팅 환경에 낸드 플래시 저장장치의 폭넓은 적용에 크게 기여하였다. 위와 같은 용량 중심 설계 기술은 디바이스의 집적도를 향상시키는 데는 매우 효율적이지만, 디바이스의 성능 및 수명 측면에는 막대한 악영향을 가져왔다. 반면, 향후 성능 및 수명에 대한 요구사항 수준 또한 지속적으로 증가할 것임을 고려할 때, 성능 및 수명 하락을 완화하기 위한 기존 기법의 최적화 정도를 뛰어넘을 수 있는 새로운 기법의 개발이 필요하다.

본 논문에서는 용량 중심 설계 기술의 부작용을 최소화하여 대용량 낸드 플래시 기반 저장장치의 성능 및 내구성을 개선하기 위한 다양한 최적화 기술을 제안한다. 기존 기법들의 최적화 한계를 극복하기 위해, 본 논문에서 제안한 기법들은 기존에 당연히 받아들여졌던 디바이스의 특성 및 최적화 기법들의 특성을 재고찰함으로써 용량 중심 설계의 부작용 및 최적화 정도를 제한하는 근본원인을 제거한다. 이를 통해 성능 및 수명 개선정도를 극대화 하여, 대용량 낸드 플래시 기반 저장장치가 현대 응용의 성능, 수명, 그리고 용량 측면의 다양하고 높은 저장장치 요구수준을 동시에 만족시킬 수 있도록 한다.

첫째로, 본 논문에서는 멀티레벨셀(multi-level cell, MLC) 낸드 플래시 메모리를 위한 새로운 페이지 프로그램 순서인 Relaxed Program Sequence (RPS)를 제안한다. RPS는 MLC 낸드 플래시 메모리를 위한 최적화 기법들의 효율성을 크게 저해하는 기존 페이지 프로그램 순서에서의 불필요한 제약사항을 제거한다. 결과적으로, 기존 페이지 프로그램 순서의 목적인 데이터 신뢰성을 보장하면서도 상위 시스템 계층에서 이종 MLC 낸드 페이지들을 유연하게 사용하여 다치화 기술 적용으로 인한 성능 및 수명 하락을 최소화 할 수 있게 한다. 본 논문에서는 제안한 RPS 기반의 `flexFTL` 은 MLC 낸드 플래시 메모리에서 필요한 페이지 백업 동작의 오버헤드를 최소화 하면서도, 일반적인 MLC 낸드 플래시 기반 저장장치가 달성할 수 없는 높은 쓰기성능을 제공한다.

둘째로, 본 논문에서는 큰 입출력 단위를 갖는 낸드 플래시 메모리에서의 새로운 병렬적 부분페이지 읽기 명령(Subpage-Parallel Read, SPread)을 제안한다. SPread 는 일반적인 낸드 플래시 메모리의 입출력 단위인 페이지보다 더 작은 읽기를 더 빠른 시간에 처리하게 함으로써 상위 시스템과 낸드 디바이스의 입출력 단위 비대칭으로 발생하는 읽기 증폭 문제를 근본적으로 해결한다. 또한, 일반적인 호스트 응용으로 부터의 매우 작은 읽기뿐만 아니라 저장장치 내부 관리로 인한 단편화된 읽기 명령을 다수의 부분 페이지를 병렬적으로 읽음으로써 효과적으로 처리할 수 있게 한다. 이를 통해 저장장치의 성능에 막대한 영향을 주는 가비지 컬렉션의 효율을 개선함으로써 다수의 쓰기 명령을 인가하는 작업부하에서도 전반적인 성능 향상을 달성할 수 있다.

셋째로, 본 논문에서는 개별적 데이터 압축 기법을 효율적으로 통합하여 저장장치의 쓰기 감소량을 더욱 높일 수 있는 새로운 압축 기법을 제안한다. 낸드 플래시 메모리의 제한된 수명은 실제 디바이스로 인가되는 쓰기량을 줄임으로써 보다 직접적으로 큰 개선을 달성할 수 있다. 본 논문에서 제안한 기법은 개별적 기법을 독립적으로 인식하고 단순 통합하는 기존 접근 방식에서 벗어나 개별적 기법들의 효과를 동반 상승시킬 수 있는 새로운 통합 방식을 제안한다. 특히, 제안된 기법은 기존 개별 기법들의 하드웨어 자원 및 자료구조를 대부분 재사용함으로써 성능/자원 측면의 추가적 부하를 최소화하였다.

본 논문에서 제안한 기법들은 저장장치 프로토타입 및 공개 낸드 플래시 저장장치 개발/평가 환경에 구현되었으며, 다양한 응용의 작업부하를 대상으로 그 유용성을 검증하였다. 실험 결과, 본 논문에서 제안한 기법들은 기존의 최적화 기법들의 개선정도에 비해 대용량 낸드 플래시 기반 저장장치의 성능 및 수명을 크게 개선할 수 있음을 확인하였다.

키워드: 낸드 플래시 메모리, 플래시 변환 계층, 낸드 플래시 기반 저장장치, 내장형 시스템, 성능 최적화, 수명 최적화
학번: 2011-23362

감사의 글

제 박사학위논문이 논문이 완성되기까지 정말 많은 분들의 도움이 있었습니다. 이 지면을 빌어 그분들께 감사의 인사를 드리고자 합니다.

먼저 지도교수님이신 김지홍 교수님께 진심으로 감사의 말씀을 드립니다. 교수님의 지도 하에 박사과정을 할 수 있었던 것은 제 인생에 정말 큰 행운이었습니다. 아직도 너무나 부족하지만, 교수님의 세심한 배려와 지도 덕분에 연구자의 삶에 있어 필수적인 많은 것들을 배울 수 있었습니다. 큰 가르침에 조금이라도 답할 수 있도록 앞으로도 최선을 다해 노력하겠습니다. 더불어 논문 심사에 아낌없는 조언을 해주신 유승주 교수님, 이재욱 교수님, 정명수 교수님께도 깊은 감사를 드립니다.

연구실 선배이신 이성진 교수님께도 특별한 감사의 말씀을 드립니다. 때로는 친한 형님으로, 때로는 선배 연구자로서 제게 주신 아낌없는 격려와 연구 방향에 대한 조언은, 제가 박사과정을 무사히 마칠 수 있는데 말로 다 표현할 수 큰 도움이 되었습니다. 앞으로도 계속 함께 연구를 진행하면서, 언젠가는 제가 받은 도움에 조금이라도 보답할 수 있도록 열심히 노력하겠습니다.

긴 박사학위 과정동안 많은 연구실 동료분들을 만났고, 많은 도움을 받았습니다. 송욱 박사님, 김태진 박사, 성노섭, 박경진, 임제헌, 곽민우, 여러분들과 함께 할 수 있어 박사학위 과정이 그리 고된 기억만은 아니었습니다. 감사합니다. 또한 제 논문에 기술적 조언을 비롯하여 많은 도움을 주신 삼성전자의 정재용 박사님, 송영선, 김명석, 홍두원, 심영섭 수석님들께도 감사의 말씀을 드립니다. 함께 연구하면서 도움을 준 후배들인 천명준, 신슬기, 김윤아, 조건희 학생에게도 감사드립니다. 연구실 선배로서 도움을 준 것도 없이 떠나는 것 같아 마음이 무겁습니다. 여러분들의 앞길에 항상 행운이 함께 하길 기원하며, 조금이라도 도움이 될 수 있도록 노력하겠습니다.

물심양면으로 저를 지지해주신 장모님, 장인어른, 처조모님, 그리고 처제에게도 진심으로 감사드립니다. 귀하게 자란 딸, 손녀, 언니가 저로 인해 고생하는 것에 대해 항상 죄송한 마음입니다. 묵묵히 믿어주시고 지원해주신 덕분에 무사히 박사과정을 마칠 수 있었습니다. 아직도 험난한 길이 남아있지만, 아내를 행복하게 할 수 있도록, 그리고 받은 은혜에 보답할 수 있도록 최선을 다하겠습니다.

제게 세상에서 두 번째로 소중한 부모님께도 글로 다 할 수 없는 감사를 드립니다. 못난 아들을 언제나 믿어주시고, 아낌없이 지원해주시고, 끊임없이 기도해주신 어머님, 아버님. 세상에서 가장 좋은 부모님을 주신 하느님께 진심으로 감사드립니다. 제가 지금까지 작게나마 이룬 것들은 모두 두 분으로 부터 비롯됐습니다. 두 분께 받은 것이 태산이라면, 제가 갚을 수 있는 것은 고작 조그만한 돌 하나겠지요. 하지만 우공(愚公)이 산을 옮기는 마음으로, 조금이라도 더 큰 돌을 옮길 수 있도록 언제나 최선을 다하겠습니다. 감사합니다. 그리고 사랑합니다. 아울러 언제나 제 편이 되주신 우리 가족들, 형, 형수님, 막내이모, 막내이모부, 막내고모, 막내고모부께도 진심으로 감사드립니다.

마지막으로, 제 삶의 이유인 아내 이상민에게 이 논문을 바칩니다. 제가 지금껏 한 노력과 앞으로 할 노력은 모두 당신을 위함입니다.

2019년 8월

박 지 성