2013

Performance and Reliability Study and Exploration of NAND Flash-based Solid State Drives

Guanying Wu
Virginia Commonwealth University

Performance and Reliability Study and Exploration of NAND Flash-based Solid State Drives

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

by

Guanying Wu
B. S., Zhejiang University, Hangzhou, China. July, 2007.
M. S., Tennessee Technological University, Cookeville, TN, USA. December, 2009.

Director: Dr. Xubin He, Associate Professor
Department of Electrical and Computer Engineering

Virginia Commonwealth University
Richmond, Virginia
August 2013

Acknowledgements

I would like to express the deepest appreciation to my committee chair Dr. Xubin He, for his constant support and guidance at VCU as well as TTU in these years. Thanks to him, I was able to develop my knowledge and skills in my field of study. In particular, I am grateful I had his support to pursue the research in Solid State Technologies, which is interesting, promising, as well as challenging. I also would like to thank Dr. Preetam Ghosh, Dr. Robert H. Klenke, Dr. Weijun Xiao, and Dr. Meng Yu for serving on my advisory committee. They were very much kind and thoughtful to me. Meanwhile, our research group, The Storage Technology and Architecture Research (STAR) Lab, has provided a joyful and incentive environment, from which I have benefited significantly in both my study and life.

I would like to say thank you to my parents. I had to sacrifice the time to be around them in the past five years and I am wishing to make it up as soon as I can. I am especially grateful to my wife, who had been patient with me in my most miserable days. You could not be more wonderful.

Contents

List of Tables

List of Figures

Abstract

PERFORMANCE AND RELIABILITY STUDY AND EXPLORATION OF NAND FLASH-BASED SOLID STATE DRIVES

By Guanying Wu

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Virginia Commonwealth University.

Virginia Commonwealth University, 2013

Major Director: Dr. Xubin He, Associate Professor, Department of Electrical and Computer Engineering

The research that stems from my doctoral dissertation focuses on addressing essential challenges in developing techniques that utilize solid-state memory technologies (with emphasis on NAND flash memory) from device, circuit, architecture, and system perspectives in order to exploit their true potential for improving I/O performance in high-performance computing systems. These challenges include not only the performance quirks arising from the physical nature of NAND flash memory, e.g., the inability to modify data in-place,

read/write performance asymmetry, and slow and constrained erase functionality, but also the reliability drawbacks that limits solid state drives (SSDs) from widely deployed.

To address these challenges, I have proposed, analyzed, and evaluated the I/O scheduling schemes, strategies for storage space virtualization, and data protection methods, to boost the performance and reliability of SSDs.

Key Words: Solid state devices; NAND flash memory; Data Storage; Performance; Reliability.

Chapter 1

Introduction

Solid State Drives (SSD's) have shown promise to be a candidate to replace traditional hard disk drives. The benefits of SSD's over HDD's include better durability, higher performance, and lower power consumption, but due to certain physical characteristics of NAND flash, which comprise SSDs, there are some challenging areas of improvement and further research. In this section, I will begin with an introduction to the subject of my research, i.e., NAND flash memory and SSDs, followed by a statement of key problems to address as well as a summary of proposed approaches.

1.1 Background

In this section, I will briefly overview the related background of my research, i.e., state-of-the-art techniques adopted in NAND flash memory and SSD architecture.

1.1.1 NAND Flash Memory

In general, the data retention of NAND flash memory is done by the charge trapped in the floating gate of the flash cell, and the amount of charge determines the logical level of a certain cell. According to the maximum number of levels defined when the data are retrieved, there are two primary types of NAND flash memory: Single-level cell (SLC) and Multi-level cell (MLC). As one would expect, single-level cell flash stores one bit per transistor, while multi-level cell flash stores multiple bits per transistor. MLC is one of the efforts made for increasing the storage density of the flash.

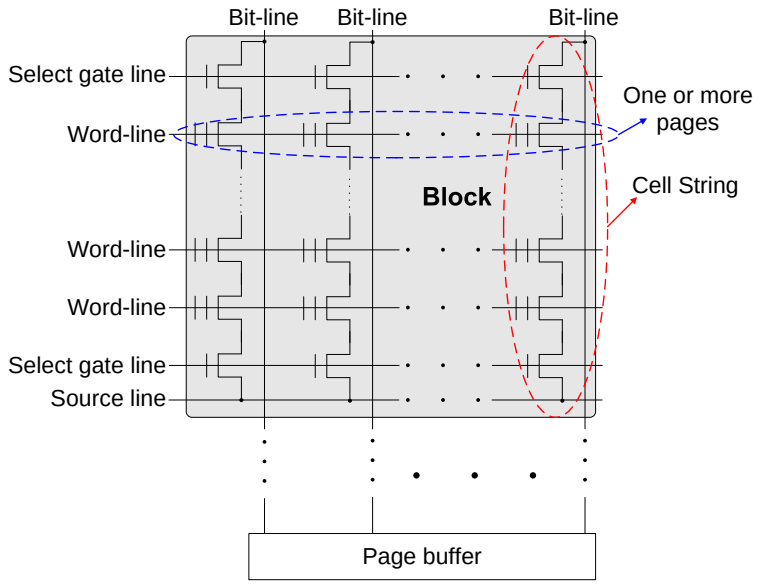


Figure 1.1: NAND flash memory structure.

To further push the storage density envelope, NAND flash memory cells are organized in an array→page→block hierarchy (Figure 1.1), where a NAND flash memory array is partitioned into blocks, and each block contains a number of pages. Within each block, each memory cell string typically contains 64 to 256 memory cells, and all the memory cells driven by the same word-line are programmed and sensed at the same time. All the memory cells within the same block must be erased at the same time. Data are programmed

and fetched in the unit of page. The read operation consists of sensing and loading the data from cells to the page buffer and transmitting the data from page buffer to a flash controller.

The write operation consists of receiving the data page to be written from the flash controller, loading the page buffer with the data, and then writing on the flash page using ISPP (Incremental Step Pulse Program [10]). The erase operation simply takes a long erase pulse (in micro seconds) to reset the cells in the target flash block. Typical access latency values of these operations are listed in Table 1.1.

Table 1.1: Values from [3] for a Samsung 4 GB Flash Module.

Page Read to Register	25 μs
Page Program from Register	200 μs
Block Erase	1.5 ms

1.1.2 NAND Flash Program/Erase Algorithm

Compared to the read operation which simply applies the predefined voltage bias on the cell and detects whether the cell is turned on or not, the P/E operations are more complex in that the charging/discharging process should be precisely controlled to achieve a pre-defined amount of charges in the cells [8]. One state-of-the-art technique known as "Incremental Step Pulse Program"(ISPP) is used for the flash programming [10]. It consists of a series of program and verify iterations. For each iteration, the program pulse voltage, V_{pp}, is increased by ΔV_{pp}, which is normally a few tenth of a volt [77]. ISPP is illustrated in Fig. 1.2. For the erase operation, the duration of the discharge/erase voltage applied on the flash cells is ensured to remove the charges in all cells of one flash block. Therefore, the P/E latency of NAND flash is much higher than the read latency. The execution of ISPP and the erase process is implemented in the flash chip with an analog block and a control logic

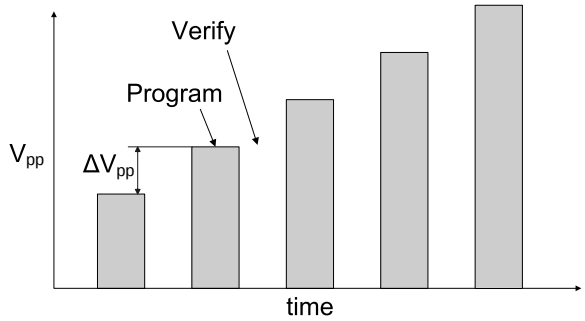


Figure 1.2: Control-gate voltage pulses in program-and-verify operation.

block. The analog block is responsible for regulating and pumping the voltage for program or erase operations. The control logic block is responsible for interpreting the interface commands, generating the control signals for the flash cell array and the analog block, and executing the program and erase algorithms. As shown in Figure 1.3, the write state machine consists of three components: an *algorithm controller* to execute the algorithms for the two types of operations, several *counters* to keep track of the number of ISPP iterations, and a *status register* to record the results from the verify operation. Both

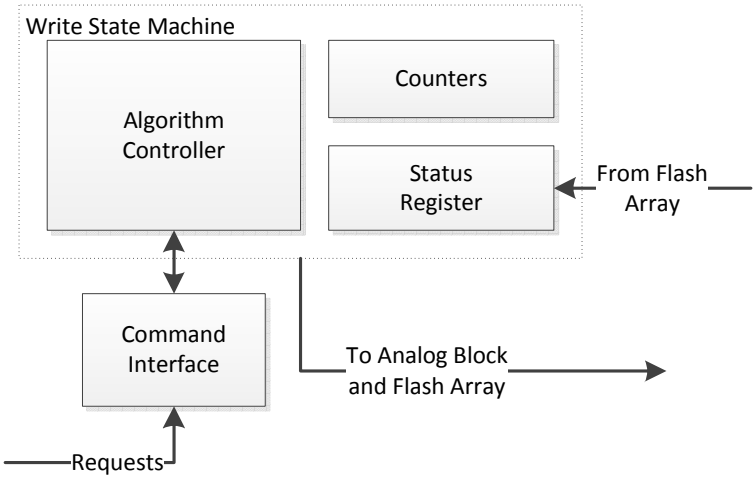


Figure 1.3: Control Logic Block [10]

program and erase operations require a precise timing control, i.e., the program or erase voltage pulse that applies on the cell must be maintained for the predefined time period, which is determined by the physical feature of the flash.

1.1.3 NAND Flash-based SSDs

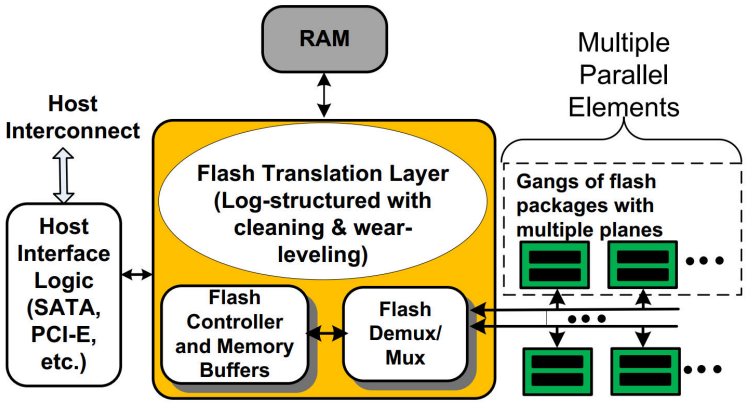


Figure 1.4: Typical SSD Architecture [60].

The NAND flash by itself exhibits relatively poor performance [78, 75]. The high performance of an SSD comes from leveraging a hierarchy of parallelism. At the lowest level is the *page*, which is the basic unit of I/O read and write requests in SSDs. Erase operations operate at the *block* level, which are sequential groups of pages. A typical value for the size of a block is 64 to 256 pages. Further up the hierarchy is the plane, and on a single die there could be several planes. Planes operate semi-independently, offering potential speed-ups if data is striped across several planes. Additionally, certain copy operations can operate between planes without crossing the I/O pins. An upper level of abstraction, the chip interfaces, free the SSD controller from the analog processes of the basic operations, i.e., read, program, and erase, with a set of defined commands. NAND interface standards includes ONFI [56], BA-NAND [56], OneNAND [62], LBA-NAND [71], etc. Each chip is connected via the data buses to the central control unit of an SSD, which is typically implemented in one micro-processor coupled with RAMs. The RAM space is often utilized to cache the write requests and mapping table entries.

SSDs hides the underlying details of the chip interfaces and exports the storage space as a standard block-level disk via a software layer called *Flash Translation Layer* (FTL), running on the in-drive micro-processor. The typical SSD architecture is illustrated in Figure 1.4 [60]. FTL is a key component of an SSD in that it not only is responsible for managing the "logical to physical" address mapping but also works as a flash memory allocator, wear-leveler, and garbage collection engine.

Mapping Schemes

The mapping schemes of FTL's can be classified into two types: page-level mapping, with which a logical page can be placed onto any physical page; or block-level mapping, with which the logical page LBA is translated to a physical block address and the offset of that page in the block. Since with block-level mapping, one logical block corresponds to one physical block, we refer *a logical block on a physical block* as a *data block*. As the most commonly used mapping scheme, *Log-block FTL's* [61] reserve a number of physical blocks that are not externally visible for logging pages of updated data. In log-block FTL's, block-level mapping is used for the data blocks, while page-level mapping is for the log blocks. According to the block association policy (how many data blocks can share a log block), there are mainly three schemes, *block-associative sector translation* (BAST) [38], *fully-associative sector translation* (FAST) [41], and *set-associative sector translation* (SAST) [32]. In BAST, a log block is assigned exclusively to one data block; in FAST, a log block can be shared among several data blocks; SAST assigns a set of data blocks to a set of log blocks.

Garbage Collection Process

In the context of log-block FTL's, when free log blocks are not sufficient, the *garbage collection* process is executed, which merges clean pages on both the log block and data block together to form a data block full of clean pages. Normally this process involves the following routine: read clean pages from the log block and the corresponding data block(s) and form a data block in the buffer; erase the data block(s) and log block; program the data on a clean physical block (block that contains no data at all). Sometimes the process can be quite simplified: if we consider a log block that contains all the clean pages of an old data block, the log block can just replace the old data block; the old data block can be erased, making one clean physical block. We refer to the normal process as *full merge* and the simplified one as *switch merge*. A *Partial merge* happens when the log block contains only (but not all) clean pages of one data block, and the garbage collection process only requires that the rest of the clean pages get copied from the data block to the log block. Afterwards, the log block is then marked as the new data block and the old data block gets erased.

To make a quantitative view of the overhead of different merge routines, Table 1.2 compares the numbers of clean page reading, page programming, and block erase, which are involved in garbage collection routine of the BAST FTL. The former two are in the order of number of pages, and the last one is in number of blocks.

Table 1.2: Overhead difference among full merge, partial merge and switch merge. N stands for the number of pages per block; N_c means the number of clean pages in the data block.

	Full merge	Partial merge	Switch merge
Clean page reading	N	N_c	0
Page programming	N	N_c	0
Block erase	2	1	1

1.2 Related Work: SSD Performance and Reliability

To improve the performance and reliability of flash-based SSDs, many designs have been proposed in the literature working with the file system, FTL, cache scheme, etc.

File systems: Early flash file systems such as YAFFS [52] and JFFS2 [25] are designed for embedded systems and work on the raw flash. On the contrary, DFS [30] is implemented over the virtualized flash interface offered by Fusion-IO driver. By leveraging this interface, it avoids the complexity of physical block management of traditional file systems.

FTLs: For block-level mapping, several FTL schemes have been proposed to use a number of physical blocks to log the updates. Examples include FAST [41], BAST [38], SAST [32], and LAST [43]. The garbage collection of these schemes involves three types of merge operations, *full*, *partial*, and *switch* merge. The block-level mapping FTL schemes leverage the spacial or temporal locality in write workloads to reduce the overhead introduced in the merge operations. For page level mapping, DFTL [23] is proposed to cache the frequently used mapping table in the in-disk SRAM so as to improve the address translation performance as well as reduce the mapping table updates in the flash; μ-FTL [44] adopts the μ-tree on the mapping table to reduce the memory footprint. Two-level FTL [73] is proposed to dynamically switch between page-level and block-level mapping. Content-aware FTLs (CAFTL) [15][22] implement the deduplication technique as FTL in SSDs. ΔFTL [74] exploits another dimension of locality, the content locality, to improve the lifetime of SSDs.

Cache schemes: A few in-disk cache schemes like BPLRU [37], FAB [29], and BPAC [76] are proposed to improve the sequentiality of the write workload sent to the FTL, so as to reduce the merge operation overhead on the FTLs. CFLRU [59] which works

as an OS level scheduling policy, chooses to prioritize the clean cache elements when doing replacements so that the write commitments can be reduced or avoided. Taking advantage of fast sequential performance of HDDs, Griffin [66] and I-CASH [79] are proposed to extend the SSD lifetime by caching SSDs with HDDs. FlashTier [63] describes a system architecture built upon flash-based cache geared with dedicated interface for caching.

Heterogeneous material: Utilizing advantages of PCRAM, such as the in-place update ability and faster access, Sun *et al.* [69] describe a hybrid architecture to log the updates on PCRAM for flash. FlexFS [42], on the other hand, combines MLC and SLC as trading off the capacity and erase cycle.

Wear-leveling Techniques: Dynamic wear-leveling techniques, such as [65], try to recycle blocks of small erase counts. To address the problem of blocks containing cold data, static wear-leveling techniques [14] try to evenly distribute the wear over the entire SSD.

Read/Write Speed vs. Reliability Trade-offs: NAND flash memory manufacturers must reserve enough redundant bits in the flash pages to ensure the worst case reliability at the end of their lifetime. Y. Pan et al. proposed to trade the such reliability over-provisioning (at the early age of the flash memory) for faster write speed by increasing ΔV_{pp} [58]. S. Lee et al. proposed to exploit the self-recovery mechanics of NAND flash memory to dynamically throttle the write performance so as to prolong the SSD lifetime [40]. In [47] R. Liu et al. proposed to trade the retention time of NAND flash for faster write or shorter ECCs.

1.3 Problem Statement

SSD Read Performance Issues: The read access latency is a critical metric of SSDs' performance, attributed to 1) raw access time including on-chip NAND flash memory sensing latency, flash-to-controller data transfer latency, and ECC decoding latency; 2) the queuing delay.

SSD Reliability Issues: The limited lifetime of SSDs is a major drawback that hinders their deployment in reliability sensitive environments. Pointed out in the literature, "endurance and retention of SSDs is not yet proven in the field" and "integrating SSDs into commercial systems is painfully slow". The reliability problem of SSDs mainly comes from the following facts. Flash memory must be erased before it can be written and it may only be programmed/erased for a limited times (5K to 100K) [21]. In addition, the out-of-place writes result in invalid pages to be discarded by garbage collection (GC). Extra writes are introduced in GC operations to move valid pages to a clean block [3] which further aggravates the lifetime problem of SSDs.

1.4 Research Approaches

On the SSD read performance issues, two approaches (DiffECC discussed in Chapter 2 and Program/Erase Suspension discussed in Chapter 3) are proposed to reduce the latency from two perspectives, i.e., the raw read latency and queuing delay, respectively. To enhance SSD reliability, my work (Delta-FTL discussed in Chapter 4 falls into the area of FTL design. Delta-FTL leverages the content locality to prolong SSD lifetime via the idea of delta-encoding.

Chapter 2

Exploiting Workload Dynamics to Improve SSD Read Latency via Differentiated Error Correction Codes

2.1 Introduction

As pointed out in [39], the read access latency is another critical metric of SSDs. SSD read access latency mainly consists of on-chip NAND flash memory sensing latency, flash-to-controller data transfer latency, and ECC (Error Correction Code) decoding latency. There is an inherent trade-off between storage density and read access latency. The storage density can be improved by using a larger NAND flash page size. Moreover, if each entire page is protected by a single ECC, the coding redundancy can be minimized, leading to a higher effective storage density; however, the use of larger page size and a longer ECC

codeword inevitably increases the read access latency, in particular the flash-to-controller data transfer latency and ECC decoding latency.

This work presents a cross-layer design strategy that can reduce the average SSD read access latency when large NAND flash memory page size is used. This design strategy is motivated by an inherent NAND flash memory device write speed vs. raw storage reliability trade-off: if we can intentionally slow down NAND flash memory internal write operation, which can enable a finer-grained control of memory cell programming states, the raw NAND flash memory storage reliability will accordingly improve. Therefore, by leveraging run-time workload variability, if the SSD controller can opportunistically slow down the NAND flash memory write operation through appropriate use of data buffering, it can opportunistically use different ECC coding schemes to reduce the read access latency. In particular, if NAND flash memory is allowed to write one page of data with a slower-than-normal speed and hence better-than-normal raw storage reliability, this page can be partitioned into several segments and each segment is protected by a shorter and weaker ECC. As a result, when this page is being read, since each small segment can be decoded independently, the flash-to-controller data transfer and ECC decoding can be largely overlapped, leading to a dramatically reduced flash read latency. The data access workload variation naturally allows us to take advantage of the bandwidth at the idle time to slow down the write speed of the SSDs in order to opportunistically improve SSD read response speed, as discussed above. In this work, we propose a disk level scheduling method to smooth the write workload and opportunistically slow down certain write operations.

It should be pointed out that this proposed design approach does not sacrifice the SSD write speed performance. The objective is to *opportunistically* slow down the NAND

flash memory device write operations when the device is idle, because of the data access workload variation in the run time. Moreover, for writes, the OS page cache works on scheduling the actual commitment on the disks and hiding the write latency. With the aid of on-disk write buffer, the disk can adopt the *write-back* scheme which reports write completion as soon as the data are buffered. These factors can be naturally leveraged to improve the probability of opportunistic write slow down.

In the rest of this chapter, **DiffECC**, a novel cross-layer co-design to improve SSD read performance using differentiated ECC schemes, is proposed, discussed, and evaluated in detail.

2.2 Background

In this section, a model of bit error rate of NAND flash memory is introduced, followed by a brief discussion about the error correction coding schemes (with emphasis on BCH code) used to protect NAND flash from bit errors.

2.2.1 NAND Flash Error Rate

Ideally, threshold voltage distributions of different storage states should be sufficiently far away from each other to ensure a high raw storage reliability. In practice, due to various affects such as background pattern dependency, noises, and cell-to-cell interference [18], the threshold voltage distributions may be very close to each other or even overlap, leading to non-negligible raw bit error rates. In the following, we present an MLC cell threshold voltage distribution model that will be used for quantitative performance evaluation and comparison in this work. The erase state tends to have a wide Gaussian-like distribution

[70], i.e., the probability density function (PDF) of the threshold voltage distribution can be approximated as

$$p_0(x) = \frac{1}{\sigma_0\sqrt{2\pi}} \cdot e^{-\frac{(x-\mu)^2}{2\sigma_0^2}}$$

where σ_0 is the standard deviation and μ is the mean threshold voltage of the erase state. All the other states tend to have the same threshold voltage distribution, as illustrated in Fig. 2.1. The model consists of two parts, an uniform distribution in the middle and Gaussian distribution tail on both sides [70]. The width of the uniform distribution equals

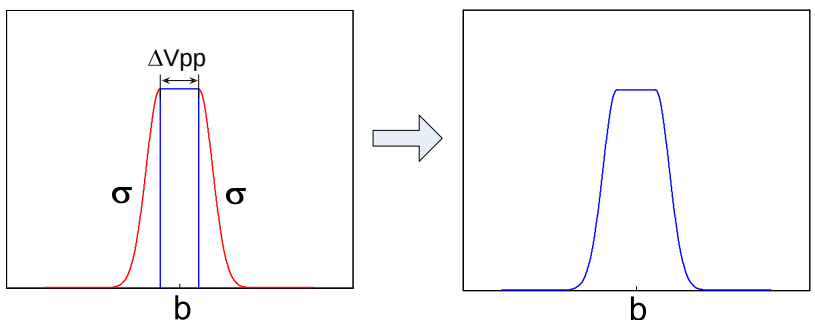


Figure 2.1: Threshold voltage distribution model NAND flash memory (except the erase state).

to the program step voltage ΔV_{pp}, and the standard deviation of the Gaussian distribution is denoted as σ. The Gaussian distribution on both sides models the overall effect of background pattern dependency, noises, and cell-to-cell interference. Let P_0 and P_1 denote the probabilities of the uniform distribution and the Gaussian distribution, respectively. We

have the overall PDF $f_{pr}(x)$ as

$$f_{pr}(x) = \begin{cases} \frac{c}{\sigma\sqrt{2\pi}}, b - 0.5\Delta V_{pp} \leq x \leq b + 0.5\Delta V_{pp} \\ \\ \frac{c}{\sigma\sqrt{2\pi}} e^{-\frac{(x-b-0.5\Delta V_{pp})^2}{2\sigma^2}}, x > b + 0.5\Delta V_{pp} \\ \frac{c}{\sigma\sqrt{2\pi}} e^{-\frac{(x-b+0.5\Delta V_{pp})^2}{2\sigma^2}}, x < b - 0.5\Delta V_{pp} \end{cases}$$

where b is the mean of the threshold voltage (i.e., the center of the distribution as shown in Fig. 2.1), and the constant c can be solved based on $P_0 + P_1 = \int_{-\infty}^{+\infty} f_{pr}(x)dx = 1$. It is clear that, as we reduce the program step voltage ΔV_{pp}, adjacent states will have less probability to overlap. Hence the raw storage reliability will improve, while the memory write latency will accordingly increase. This suggests that there is an inherent trade-off between NAND flash memory write latency and raw storage reliability.

The use of a larger page size can increase the effective NAND flash memory storage density from two perspectives: (i) A larger page size enables more memory cells share the same word-line, leading to a more compact memory cell array layout and hence higher storage density; (ii) Given the same raw bit error rate, ECC with a longer codeword tends to use less coding redundancy (i.e., higher code rate). A larger page size enables the use of ECC with longer codeword length, leading to less coding redundancy and hence higher effective storage density. However, a large page size apparently will result in a longer time to transmit the data from the flash die to the controller. Meanwhile, the ECC decoding latency may increase as the codeword length increases. As a result, the read response time of SSD will inevitably increase.

2.2.2 Error Correction Code Schemes

As the storage density continues to grow, NAND flash memory uses increasingly powerful ECC on each individual page to ensure storage reliability [8]. A more powerful ECC with stronger error correction capabilities, tends to demand more redundant bits, which causes an increase in requirements for storage space. Therefore, designers always select an ECC that provides just enough error correction capability to satisfy the given reliability specifications. Moreover, *with the same code rate, the longer the codeword is, the better the coding efficiency is*. Hence, as pointed out earlier, ECC with a longer codeword length requires less coding redundancy and thus leads to a higher storage density. In current design practice, binary BCH code is being widely used in NAND flash memories [20][17][68].

Binary BCH code construction and encoding/decoding are based on binary Galois Fields [46]. A binary Galois Filed with degree of m is represented as GF(2^m). For any $m \geq 3$ and and $t < 2^{m-1}$, there exists a primitive binary BCH code over GF(2^m), which has the codeword length $n = 2^m - 1$ and information bit length $k \geq 2^m - m \cdot t$ and can correct up to (or slightly more than) t errors. A primitive t-error-correcting (n, k, t) BCH code can be shortened (i.e., eliminate a certain number, say s, of information bits) to construct a t-error-correcting $(n - s, k - s, t)$ BCH code with less information bits and code length but the same redundancy. Given the raw bit error rate p_{raw}, an (n, k, t) binary BCH code can achieve a codeword error rate of

$$P_e = \sum_{i=t+1}^{n} \binom{n}{m} p_{raw}^i (1 - p_{raw}^i)^{n-i}$$

Binary BCH encoding can be realized efficiently using linear shift registers, while binary BCH decoding is much more complex and the computational complexity is proportional

to t^2. Readers can refer to [9] and [46] for a more detailed discussion of various BCH decoding algorithms.

2.3 Analysis and Modeling

In this section, we first demonstrate how write speed would affect the raw reliability by an example; then we discuss about reducing the read latency via page segmentation and weaker/shorter ECC; finally, we explore the potential of applying the differentiated ECC scheme through an analysis of read-world disk I/O workloads.

2.3.1 Write Speed vs. Raw Reliability Trade-off

A slowing down write speed which is reflected by a smaller program step voltage ΔV_{pp} can improve the raw NAND flash memory reliability due to the narrowed V_{th} distribution. Let us consider 2bits/cell NAND flash memory as an example. We set the program step voltage ΔV_{pp} to 0.4 as a baseline configuration and normalize the distance between the mean of two adjacent threshold voltage windows as 1. Given the value of program step voltage ΔV_{pp}, the BCH code decoding failure rate (i.e., the page error rate) will depend on the standard deviations of the erased state (i.e., σ_0) and the other three programmed states (i.e., σ). We fix the normalized value of σ_0 as 0.1 and carry out simulations to evaluate the sector error rate vs. normalized σ, as shown in Fig. 2.2. The results clearly show that only 5% reduction of the ΔV_{pp} can achieve a noticeable performance improvement under the same BCH code.

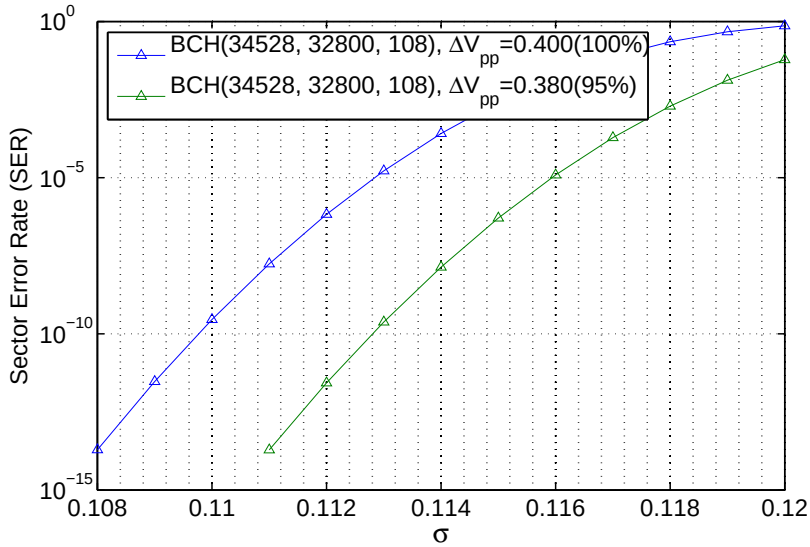


Figure 2.2: Simulation of SER under two different program step voltage ΔV_{pp} and hence different NAND flash memory write speed.

2.3.2 Read Access Latency

Read access latency includes the on-chip NAND flash sensing latency, flash-to-controller data transfer latency, and ECC decoding latency. The on-chip NAND flash sensing latency is typically a few tens of μs. Assuming the NAND flash chip connects with the SSD controller through a 100MHz 8-bit I/O bus, it takes at least 40.96μs and 20.48μs to transfer one page from NAND flash chip to controller when the page size is 4 KB and 2 KB, respectively. Typically, *ECC decoding delay is linearly proportional to its codeword length*. Hence, the use of large page size will inevitably result in a longer ECC decoding delay. For example, assuming the use of parallel BCH code decoder architecture presented in [68], we estimate that the overall BCH decoding latency is 41.2μs and 22.3μs when using one BCH code to protect 4 KB and 2 KB user data, respectively. Therefore, if we assume the on-chip NAND flash page sensing latency is 25μs, the overall read access latency is 107.16μs and 67.78μs when the page size is 4 KB and 2 KB, respectively. It suggests that the overall read latency may increase 58% when we increase the page size from 2 KB to

4 KB. As pointed out earlier, the use of large page size is beneficial from storage density perspective. To improve the read response time while increasing page size, straightforward solutions include the use of higher speed I/O bus and/or higher throughput BCH decoder. However, increasing the bus speed and/or decoder throughput will greatly increase the power consumption and silicon cost.

NAND flash memory has an inherent write speed vs. raw storage reliability trade-off. Therefore, if we could exploit the run-time workload variability and use the on-chip buffer in SSD controller to opportunistically slow down the NAND flash write operations, we can opportunistically increase the NAND flash raw storage reliability and hence use shorter and weaker ECC, which can directly reduce overall read access latency. This intuition leads to the basic idea of this work, i.e., by opportunistically slowing down NAND flash memory write operations, we can use different ECC coding schemes in order to reduce average overall SSD read access latency.

In particular, given better NAND flash memory raw storage reliability from slow programming speed, we can partition one large page into a few smaller segments, each one is protected with one shorter (thus weaker) ECC. Figure 2.3(a) illustrates the baseline mode of data and ECC storage in the flash page where the entire data area is encoded with ECC as a whole; Figure 2.3(b) shows an example of the proposed page segmentation: the data is split into 4 segments and each segment is encoded with ECC individually.

With segmentation, the flash-to-controller data transfer and ECC decoding can be largely overlapped, i.e., once the controller receives the first segment, it starts to decode while the following segments are being transferred. This can largely reduce the overall read access time of a flash page. We illustrate this idea in Figure 2.4: after the flash sensing latency (T_{read_raw}), the baseline mode would take a long bus transfer period (T_{bus})

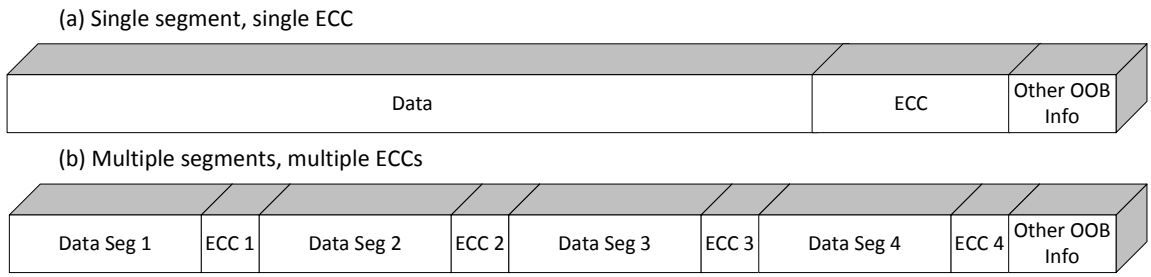


Figure 2.3: Data and ECC storage in the flash page: single segment and single ECC vs. multiple segments and multiple ECC.

and ECC decoding period (T_{ecc_decode}) without being able to overlap these two periods. However, with segmentation, because T_{bus} of current segment transfer and T_{ecc_decode} of previous segment decoding are independent of each other and thus can be overlapped, we may achieve reduced read latency compared to the baseline mode.

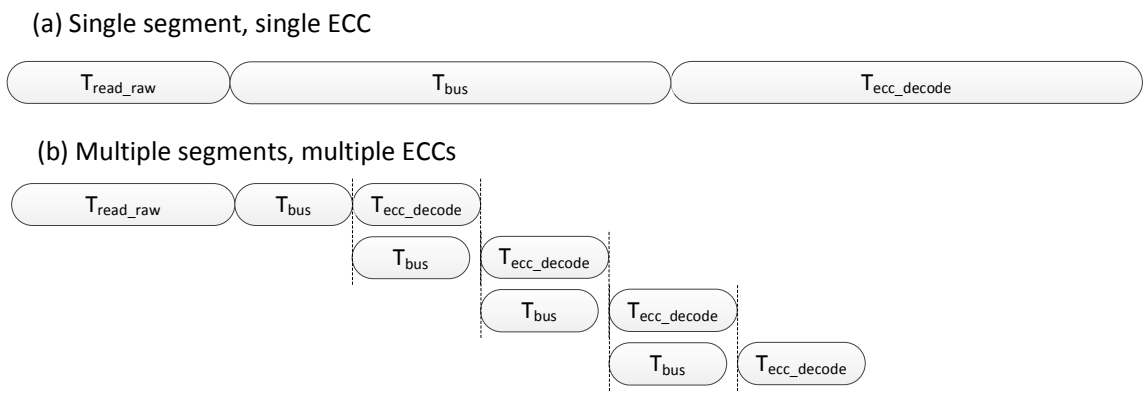


Figure 2.4: Read latency reduction: pipelining bus transfer and ECC decoding via page segmentation.

2.3.3 Server Workload Analysis

The workloads of the servers are often time-varying, while to build a server, the hardware/software configurations are determined by the needs of the peak performance, which is often affected by the request burstness. For example, Kavalanekar *et al.* [35] characterized the block I/O workloads of the Production Windows Servers, and marked

Table 2.1: Disk Traces Information

Parameter	F1	F2	C3	C8	DAP	MSN
Reads(10^6)	1.23	3.04	0.75	0.56	0.61	0.40
Read %	23.1	82.3	35.3	27.4	56.2	75.0
Compression Ratio	X10	X30	X3	X3	X8	X50
ARR	1240	3223	576	666	135	1245
Idle %	52.2	51.6	57.4	56.0	63.9	66.2
WR_{max} %	93.8	73.0	91.5	97.3	99.2	45.6

that most of them show a high level of burstiness, which is measured in self-similarity. Self-similarity means bursts occur at a wide range of time scales, e.g., the diurnal pattern is a day-scale factor; the Internet traffic, which incurs congestions, is a minute-scale factor; the operating system, which periodically flush the dirty data, is a millisecond to second scale factor. In addition, the I/O behavior of the application contributes to the variations at various time scales. Meeting the peak performance needs makes the bandwidth of the hardwares under fully-exploited, especially for the self-similar workloads which have concentrated bursts.

To learn about the workload stress on the SSDs, we have conducted trace-driven simulation experiments with an SSD simulator based on the Microsoft Research SSD extension [3] for Disksim 4.0. The simulated SSD is configured realistically to match a typical SSD: there are 16 flash chips, each of which owns a dedicated channel to the flash controller. Each chip has four planes that are organized in a RAID-0 fashion; the size of one plane is 1 GB assuming the flash is used as 2-bit MLC (page size is 4 KB). To maximize the concurrency, each individual plane has its own allocation pool [3]. The garbage collection processes are executed in the background so as to minimizing the interference upon the foreground requests. In addition, the percentage of flash space overprovisioning is set as 30%, which doubles the value suggested in [3]. Considering the limited working-set size of the workloads used in this work, 30% over-provisioning is believed to be sufficient to

avoid garbage collection processes to be executed too frequently. The garbage collection threshold is set as 10%, which means if the clean space goes below 10% of the exported space, the garbage collection processes are triggered. Here we only report the results with buffer size of 64 MB. The SSD is connected to the host via PCI-E of 2.0 GB/s.

We played back a few real-world disk I/O traces in our simulation experiments. *Financial 1* and *Financial 2*(F1, F2) [67] were obtained from OLTP applications running at two large financial institutions; the *Display Ads Platform and payload servers*(DAP) and *MSN storage metadata*(MSN) traces were from the Production Windows Servers and described in [35](note that we only extracted the trace entries of the first disk of MSN-CFS); the *Cello99* [27] trace pool is collected from the "Cello" server that runs HP-UX 10.20. Because the entire *Cello99* is huge, we randomly use one day traces (07/17/99) of two disks (C3 and C8). These disk I/O traces are originally collected on HDD systems. To produce more stressful workloads for SSDs, we deliberately compressed the simulated time of these traces so that the system idle time is reduced from originally 98% to about 50∼60%. Some basic information of these traces can be found in Table 2.1, where "ARR" stands for average request rate(requests per second); "compression ratio" means the ratio of simulation time compression done for each traces.

There are a few applications that take advantage of the idle slots, such as disk scrubbing [57], system checkpointing, data backup/mining, etc., which normally run between the midnight to daybreak. Our design differs from these applications in that it works on a smaller time scale, in particular we buffer the writes and dispatch them over the idle slots between foreground requests. With the original traces, we collected the idle slot time CDF in Figure 2.5. Except for DAP, the rest traces all have over 40% of idle slots longer than 5 ms. In comparison to HDDs, which can service a random access latency of a

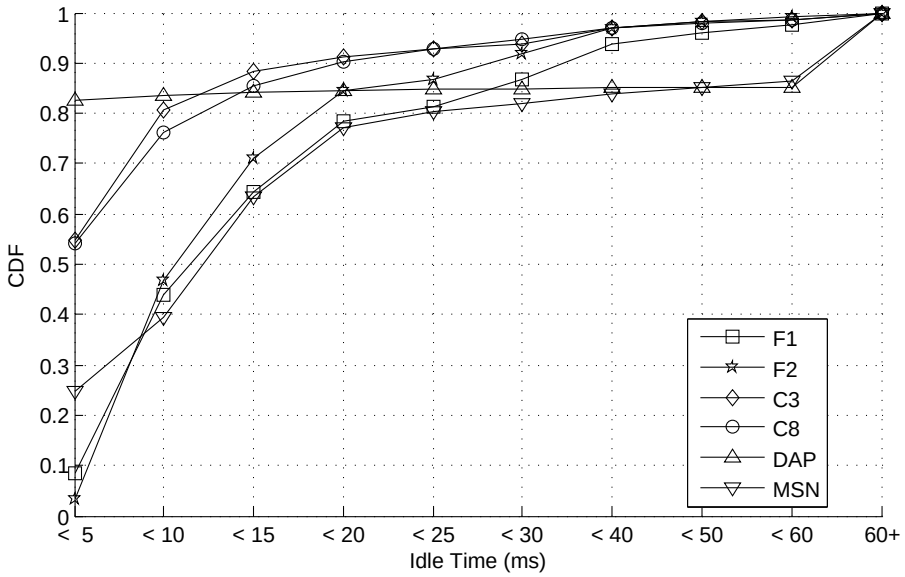


Figure 2.5: The CDF of idle slot time of six traces.

few milliseconds, SSDs can do dozens of page writes or hundreds of pages reads with the bandwidth of a single flash chip. Or we can buffer writes and slowly program them with weak ECC in the idle slots, then while read accesses occur on those pages, we may have a reduced read latency.

So, what is the chance of reading a weak ECC coded page? Ideally, assuming all page writes could be programmed slowly and coded with weak ECC, the chance is then determined by the overlap of the working-sets of reads and writes. We simulated this case with the above six traces, assuming the data are initially programmed in strong ECC. The percentage of weak ECC reads is denoted as WR_{max} in Table 2.1. For the listed six traces, WR_{max} ranges from 45.6% to 99.2%, suggesting a promising potential gain on read latency performance with our approach.

2.4 Architecture and Design of DiffECC

In this section, we first outline the architecture of our proposed design and then depict its major components including the differentiated ECC schemes and the write request scheduling policy in detail.

2.4.1 System Overview

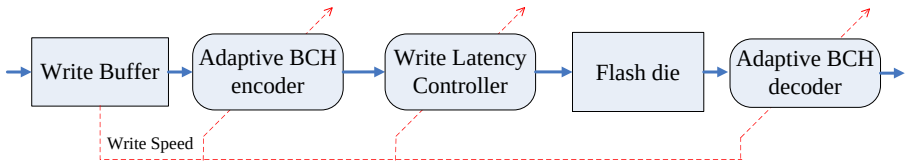


Figure 2.6: Proposed system structure.

Fig. 2.6 shows the system structure of the proposed design approach. In order to smooth the workload, an on-disk write buffer with capacity m is managed by our I/O scheduling policy. We dynamically adjust the code length of BCH encode/decoder and the write speed, according to the scheduling policy. The adaptive BCH encoder will accordingly partition the page of length L into $N(v_c)$ segments and encode each segment with BCH codes at length of $L/N(v_c)$. $N(v_c) = 1$ represents no partition is performed and the system writes at full speed. We further make the following assumptions: the data on the SSD programmed at the speed of v_c are read with the probability of $\alpha(v_c)$; the decoding latency of the BCH decoder at length of $L/N(v_c)$ is $T_{dec}(v_c)$; the bus transfer time is slightly less than ECC decoding time (as illustrated in Figure 2.4); and the average number of requested segments on the target page is $n(v_c)$ ($n(v_c) <= N(v_c)$). We can estimate the average read latency as:

$$T_{read\_raw} + \sum_{v_c} \alpha(v_c)(\frac{T_{bus}}{N(v_c)} + T_{dec}(v_c) * n(v_c)) \tag{2.1}$$

where T_{bus} is the bus delay when the whole page is transmitted through the bus from the NAND flash chip to the controller, and T_{read_raw} is the delay when the data are sensed from the cells and transferred to the page buffer. We also note that, since this design strategy is applied to each page independently, it is completely transparent to the operating systems and users. Pointed out in Section 2.4 that ECC decoding delay is linearly proportional to its codeword length. Thus, $T_{dec}(v_c)$ in Expression 2.1 is approximately the value of ECC decoding time of the baseline mode (no segmentation and strong ECC) devided by $N(v_c)$.

The analytical modeling for the expected read performance discussed above is straightforward. However, the actual performance under real-world workloads is difficult to estimate, due to a few reasons. First, as we discussed in Section 2.3, the $\alpha(v_c)$ is workload specific. Second, although doing 100% slow writes does not involve in much overhead with the HDD traces, in practice, heavier workload may be expected. So we can not always assume a 100% slow write. Third, bursts come in the term of high request rate or batched requests, which are often queued. With a queuing system, Equation 2.1 only outlines the physical access latency improvement of the reads. For example, if there are many writes ahead of a read, the long queuing time would trivialize the latency reduction resulted from weak ECC, especially, for the "Read After Write" access pattern, the read has to be arranged after the write if no cache exists. The slow writes may additionally increase the queuing time.

2.4.2 Differentiated ECC Schemes: Trading-off between Write Speed and Read Latency

In order to achieve the best possible performance, $N(v_c)$ should be able to vary within a large range and accordingly ECCs with many different code lengths should be supported by the controller. However, such an ideal case may incur prohibitive amount of implementation complexity and hence may not be practicable. In this work, to minimize the implementation overhead, we set that the system can only switch between four modes, a normal mode (also used as the baseline mode) and three slow write modes. For the baseline normal mode, we set $N(v_c) = 1$ and use a (34528, 32800, 108) BCH code in order to minimize the coding redundancy. For the slow write modes, we set $N(v_c)$ as 2, 4, and 8, considering the trade-off between required write speed slow down and read response latency improvement. The BCH coding parameters of each modes are listed in Table 2.2.

Table 2.2: BCH Parameters for Each Mode

$N(v_c)$	Segment Size	BCH(n,k,t)
1	4KB	(34528, 32800, 108)
2	2KB	(17264, 16400, 57)
4	1KB	(8632, 8200, 30)
8	512B	(4316, 4100, 16)

We assume the use of 2 bits/cell NAND flash memory. We set the program step voltage ΔV_{pp} to 0.4 and normalize the distance between the mean of two adjacent threshold voltage windows as 1. Given the value of program step voltage ΔV_{pp}, the sector error rate (SER) will depend on the standard deviations of the erased state (i.e., σ_0) and the other three programmed states (i.e., σ). In the following simulation, we fix the normalized value of σ_0 as 0.1 and evaluate the SER vs. normalized σ. For the slow write modes, under the same σ_0, we run the exhaustive simulation to choose a just slow enough ΔV_{pp} to ensure the performance is not degraded at SER of 10^{-15}. Based on the configuration mentioned above,

we demonstrate the derivation of ΔV_{pp} of mode $N(v_c) = 8$ as an example in Fig. 2.7. We can observe that ΔV_{pp} =0.265 (66.25% of the baseline, corresponding to a write latency overhead of 50.9%) is just able to compensate the performance loss caused by the using of short BCH code (4316, 4100, 16) instead of the (34528, 32800, 108) BCH code.

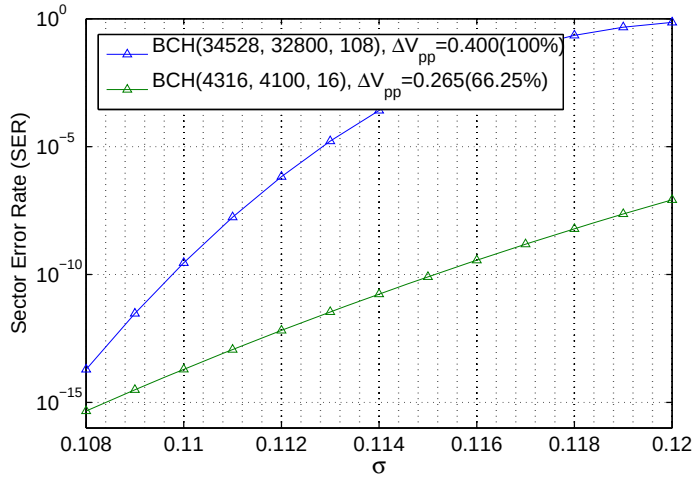


Figure 2.7: SER for two different modes.

Targeting at the throughput of 3.2Gbps and based on the hardware structure of [68], we further carry out the ASIC design of BCH decoders for the above BCH codes. TSMC 65nm standard CMOS cell and SRAM libraries are used in the estimation. We summarize the BCH decoding latency values of the four modes (as well as the write latency, bus transfer latency, etc.) in Table 2.3. This work assumes 25μs of cell to buffer read latency (T_{read_raw}) and 100 MB/s bus bandwidth.

Table 2.3: Latency results for different modes

$N(v_c)$	Write Latency(μs)	Read Latency (μs)			
		BCH	Bus	Cell to buffer	Total
1	500	41.20	40.96	25	107.16
2	538	21.71	20.48	25	67.19
4	606	11.25	10.24	25	46.49
8	754	5.78	5.12	25	35.9

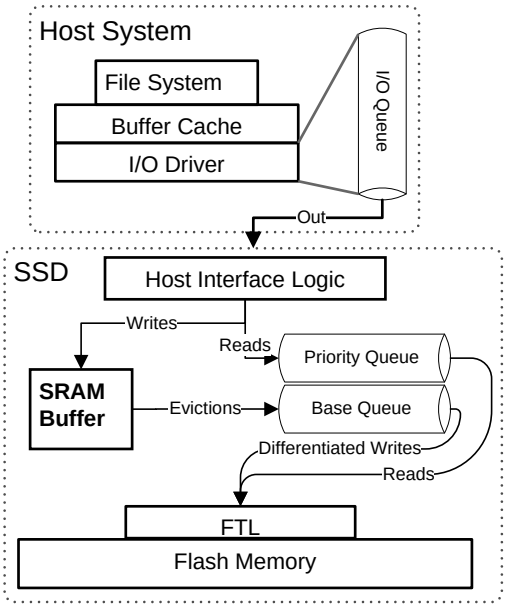


Figure 2.8: Overview of the I/O queuing system.

2.4.3 Buffer Queue Scheduling Policy

The goal of our scheduling policy is to take advantage of the idle time bandwidth of SSDs so as to slow down the write speed. Intuitively, we choose to utilize the on-disk SRAM cache to buffer the write requests, and synchronize them on the flash with high or low write speed according to the on-line load of the SSD.

Typical SSDs are equipped with an SRAM buffer, which is responsible for buffering/re-scheduling the write pages and temporally hold the read data which are to be passed on to the OS. The size of the buffer normally ranges from 32 MB to 128 MB according to the class of the product. Because of the fast random read speed of flash memory and the relatively small size of the SRAM, the cache is dedicated to writes exclusively. The write scheme of the buffer is set as “write-back”, which means completions are reported as soon as the data are buffered. While accommodating the new data and buffer is out of space, the replacements take place. The victim (dirty) pages are inserted into an I/O queue in which they wait to be programmed on the flash. After the actual synchronization on the flash, the

buffer space of the victims is freed and can accept new data. In this way, the maximum number of write pages holding in the queue is determined by the size of the buffer, e.g., with a 32 MB buffer, the number is 65536 assuming the sector size is 512B.

Given a workload, of what portion the writes can be done in slow modes? Ideally, taking out the bandwidth the SSD spends on the reads, the slow and fast writes can share the rest. For simplicity, let us assume there are two write modes in the following discussion, one slow mode ($N(v_c) > 1$) and one normal mode ($N(v_c) = 1$). The maximum throughput of slow write is denoted as Θ_s requests/s and Θ_n for the normal write, and we have $\Theta_s < \Theta_n$. According to different average write request rates(ARR_w), there are two scenarios.

- Case 1: If $ARR_w < \Theta_s$, ideally, the workload can be perfectly smoothed so that the slow write percentage is 100%.
- Case 2: $\Theta_s < ARR_w < \Theta_n$, a portion of the writes could be done in slow mode. For example, assuming $\Theta_s = 100, \Theta_n = 200, ARR_w = 150$, the maximum percentage of slow writes can be 1/2.
- Case 3: If $ARR_w = \Theta_n$, the system can only accept normal writes.

However, slow writes may involve a few overheads. First, from the host's point of view, the write throughput, which can be represented by the write requests holding on its side, could be compromised. Second, within the SSD, the slow writes occupy the flash chip for a longer time than fast writes do, so the probability of reads getting held is higher. To minimize these overheads, we consider the immediate load of the SSD regarding to the available resources.

The I/O driver of the host system and the on-disk buffer consist a queuing system as shown in Figure 2.8. The host system issues reads/writes through the I/O driver, which

queues the outstanding(in-service or to-be-serviced) requests; the SSD caches writes in its SRAM and queues the buffer evictions and the reads in its own queues. In order to minimize the queuing time resulted from writes, we put reads in the priority-queue while putting the buffer evictions in the base-queue.

The immediate load of an SSD can be estimated by the size of its pending-request queues and the idle slot time. Assuming at a time point there are N_r pages waiting to be read (the priority queue) and N_w pages to write (the base queue). The time to fulfil the read requests is $T_r = N_r * L_r$, where L_r is the page read latency (since it is difficult to estimate or predict the read latency reduction resulted from slow write modes, we assume the worst case where the reads occur in the baseline mode). The time for doing the writes at slow speed is $T_s = N_w * L_{ws}$, where L_{ws} means the time of writing a page at slow speed, and the time of fast writes is $T_f = N_w * L_{wf}$. We note that, the queued requests are to share the bandwidth of the SSD with the foreground requests in the near future. So how much time we can expect to have for dealing with these postponed requests without interfere with the foreground request? By the recent history information, i.e., recently the average length of the idle slots is T_{idle_avg}, then if $T_s < (T_{idle\_avg} - T_r)$, we can expect that there is a low probability that slowing down the programming of all the buffered writes will increase the length of the host side queue. Furthermore, similar to the discussion about the ideal cases, if $T_f < (T_{idle\_avg} - T_r) < T_s$, we can try to output a part of queued writes at slow speed and the rest at fast speed.

2.5 Evaluation

We have implemented and evaluated our design (denoted as DiffECC) based on a series of comprehensive trace-driven simulation experiments. In this section, we present the experimental results of comparing DiffECC with the baseline. In addition, we evaluate the overhead DiffECC may potentially introduce on the write performance. Particularly, the read and write performance is measured in terms of response time and throughput, respectively.

2.5.1 Simulation Methodology

We modified the the Microsoft Research SSD extension [3] simulator to support our design, where the access latency numbers of the raw flash are taken from the above subsection and the average idle time is sampled every 5 seconds of simulated time. The initial state of each flash page is still assumed to be strong ECC coded ($N(v_c) = 1$). We use the six disk traces analyzed in Section 2.3, i.e., F1, F2, C3, C8, DAP, and MSN, which are considered covering a wide spectrum of workload dynamics. Our design is compared with the baseline, which uses the same buffer size (as well as the same allocation policy and write scheme) and adopts 100% fast/strong ECC write mode ($N(v_c) = 1$). In addition, we tune the cache size from 32 MB to 128 MB. We collected the experimental results of a few metrics, i.e., the average read/write latency, the average number of write requests held in I/O driver queue (Q_{w_avg}), and the average idle time (T_{idle_avg}). For reference, we listed the results of the baseline under 32 MB buffer in Table 2.4.

Table 2.4: The Baseline Results under 32 MB Buffer (in ms)

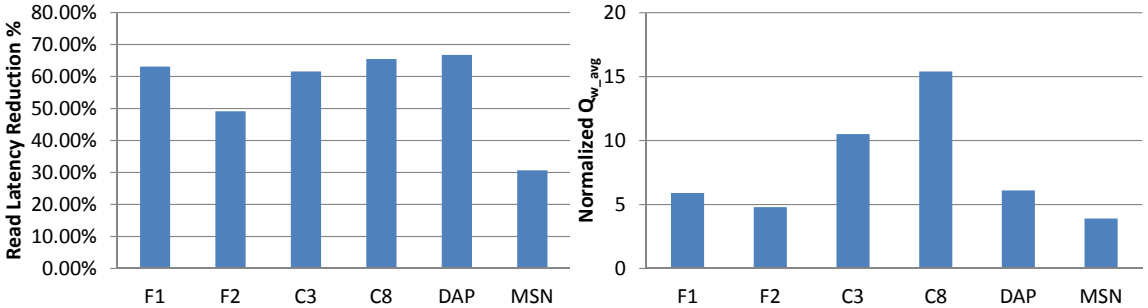
Metric	F1	F2	C3	C8	DAP	MSN
Read Latency	0.44	0.27	0.52	6.30	5.74	8.47
Write Latency	1.58	1.03	0.56	4.54	11.74	25.21
Q_{w_avg}	1.59	0.98	19.88	9.91	69.23	8.43
T_{idle_avg}	1.99	0.71	10.8	5.51	65.19	4.41

2.5.2 The Optimistic Case of DiffECC

DiffECC may achieve the optimistic performance gain on the read latency if we are allowed to carry out all write requests in the slowest mode, i.e., $N(v_c) = 8$. Here we examine this performance gain upper-bound by forcing 100% $N(v_c) = 8$ mode writes in the simulation experiments. As the metric of read performance improvement, the read latency reduction percentage against the baseline, which adopts 100% $N(v_c) = 1$ mode writes, is presented in Fig. 2.9(a). The results comply with the preliminary discussion about the upper-bound of weak-ECC read percentage in Sec. 2.3.3 (WR_{max} in Tab. 2.1) and the model of average read latency outlined in Sec. 3.4.1 (Expression 2.1). For example, DAP and MSN traces, having the most and least WR_{max}, achieve the maximum and minimum read latency reduction percentage (66.8% and 30.7%), respectively. As the in-drive cache is dedicated to writes and thus the cache size makes little difference for the read performance, we only demonstrate the results under 32 MB cache size here.

However, forcing 100% slowest write mode definitely results in overhead on the write performance. For example, the write latency of $N(v_c) = 8$ mode exceeds that of $N(v_c) = 1$ by 50.9%, which could be further amplified by the queuing effect. In our experiments, DiffECC doubles the average write latency at most cases. However, our concern is the write throughput, which should avoid being compromised by slow writes. We choose to use one metric to evaluate the overhead of slow writes on the write throughput: the average number of writes held in I/O driver queue (Q_{w_avg}). The normalized Q_{w_avg} against the

baseline is presented in Fig. 2.9(b). Under F1, F2, DAP, and MSN, the uncontrolled slow writes result in a Q_{w_avg} of about 5 times of the baseline; it is even worse as 10 and 15 times under C3 and C8, respectively, due to the batched write access pattern in these two traces. Therefore, the uncontrolled slow writes compromise the write throughput and we must avoid such overhead by selectively switching among different write modes via the scheduling policy described in Sec. 2.4.3.



(a) The percentage of read latency reduction.

(b) The average run-time write queue length. Normalized to the baseline.

Figure 2.9: The read and write performance of the optimistic case: 100% $N(v_c) = 8$ writes.

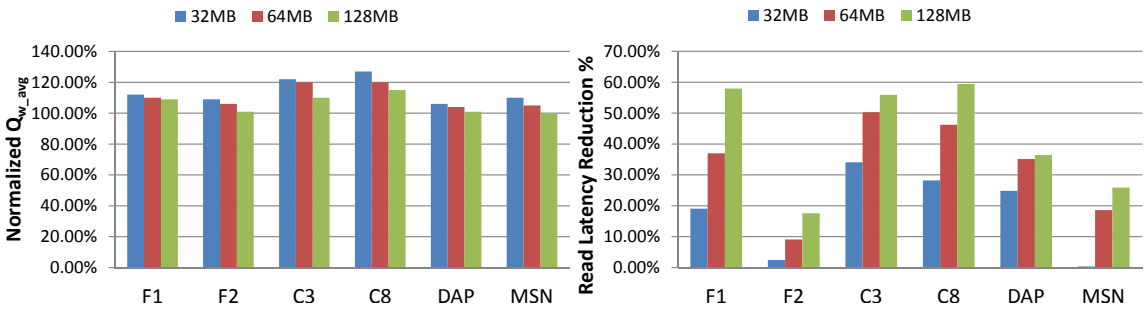
2.5.3 The Controlled Mode-switching of DiffECC

DiffECC switches among the proposed four write modes automatically regarding to the immediate load of the drive. As discussed in Sec. 2.4.3, we estimate the immediate load by the number of pending requests and the expected idle slot time. In order to avoid interfering with the future foreground requests (especially for the reads), we are required to service the pending requests in the idle slots. At the mean time, we try to adopt slow write modes as much as possible to boost the read performance. To achieve this goal, we first estimate the time to fulfill the pending writes in each mode. For example, with N_w page writes, the mode "$N(v_c) = 1$" would take the time of $T_{m1} = L_{m1} * N_w$, the mode "$N(v_c) = 2$" would

take $T_{m2} = L_{m2} * N_w$, and so on. L_{m1} and L_{m2} represent the write latency values of the two modes, respectively. If the expected time for servicing the writes ($T_{idle\_avg} - T_r$, i.e., excluding the time to service the reads from the expected idle slot time) falls in between two adjacent modes, for example, T_{m2} and T_{m4}, we would write N_{m2} pages in the mode "$N(v_c) = 2$" and N_{m4} pages in the mode "$N(v_c) = 4$", where $N_{m2} + N_{m4} = N_w$ and $N_{m2} * L_{m2} + N_{m4} * L_{m4} = T_{idle\_avg} - T_r$.

With the above mode-switching policy, DiffECC successfully eliminates the overhead on write throughput. As shown in Fig. 2.10(a), Q_{w_avg} of DiffECC (normalized to that of the baseline) is increased by mostly less than 20% (except for C8 under 32MB cache, that is because of the caching effect, i.e., more write hits, resulting longer idle slot time, which helps smoothing the write workload further). As illustrated in Table 2.4, the highest Q_{w_avg} we observed in the baseline results is about 70. Given a flash page size of 4KB, the overhead of less than 20% on Q_{w_avg} means the I/O driver needs an extra amount of 56KB ($70 * 20\% * 4KB$) memory to temporally hold the content of queued writes. Allocating and managing such a small piece of memory like 56KB is trivial compared to hundreds of MB of buffer cache in the host side. Again, DiffECC causes higher overhead on C3 and C8 traces due to the same reason mentioned in the previous subsection. It is worth noting that as the cache size increases from 32 MB to 128 MB, we observe less overhead.

However, comparing to the optimistic situation where we adopt 100% slowest writes, the controlled mode-switching of DiffECC achieves less read latency performance gain. As shown in Fig. 2.10(b), F1, C3, C8, and MSN approach the performance gain upper-bound (Fig. 2.9) as the cache size increases from 32 MB to 128 MB, while under F2 and DAP traces, DiffECC achieves relatively poorer gain. Particularly, DiffECC achieves the maximum read latency reduction of 59.4% under C8 with 128 MB cache; the average



(a) The average run-time write queue length. Normalized to the baseline.

(b) The percentage of read latency reduction.

Figure 2.10: The read and write performance of DiffECC with controlled mode-switching.

reduction percentage under 32 MB, 64 MB, and 128 MB is 18.1%, 32.8%, and 42.2%, respectively. Generally speaking, the read latency performance of DiffECC is determined by the available idle slots and the number of pending requests, which would determine the ratio of each write modes used (thus the ratio of reads in each mode).

To have more insight about the observed read performance gain, we collect the percentage of writes and reads in each mode and tune the cache size from 32 MB to 128 MB in Fig. 2.11 and Fig. 2.12, respectively. First of all, comparing the results of writes and reads, we observe apparent resemblance between them in most traces except for MSN. This is because of the extent of overlap between the working-set of writes and reads. We have examined the extent of overlap in Table 2.1 using WR_{max} as the metric: MSN has the lowest WR_{max} while the others are much more closer to 1. Second, looking at Fig. 2.11, as the cache size increases from 32 MB to 128 MB, we have more and more percentage of slow write modes. That is due to the increased idle time resulted from less write workload stress, which is in-turn caused by more write cache hits. F1 has more dramatic changes (from baseline-mode dominated at 32 MB to slowest-mode dominated at 128 MB) than the others due to a higher temporal locality existing in the writes. Third, with more slow mode writes, we observe more number of corresponding reads in these modes, which explains the

read latency reduction performance in Fig. 2.10(b). For example, under F2 the dominate read mode is constantly $N(v_c) = 1$ with all three cache sizes and under MSN the $N(v_c) = 1$ and $N(v_c) = 2$ modes outnumber the rests. Thus, DiffECC achieves less performance gain under such two traces than the others.

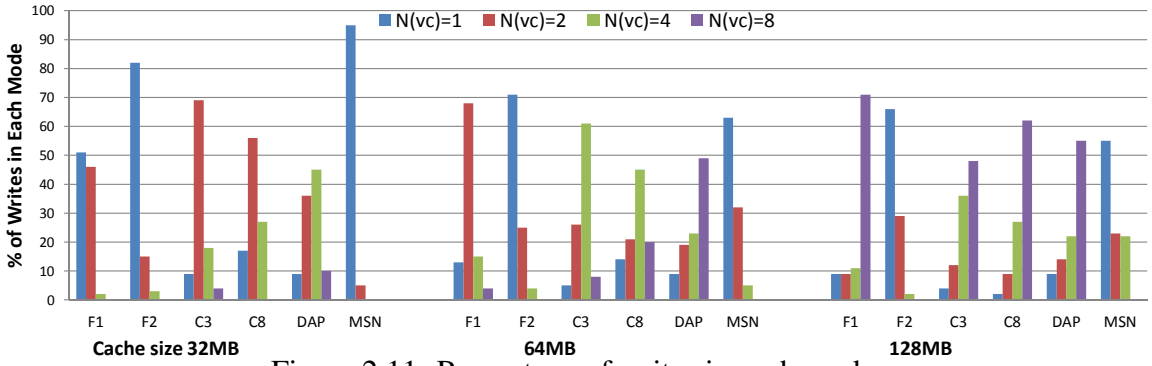


Figure 2.11: Percentage of writes in each mode.

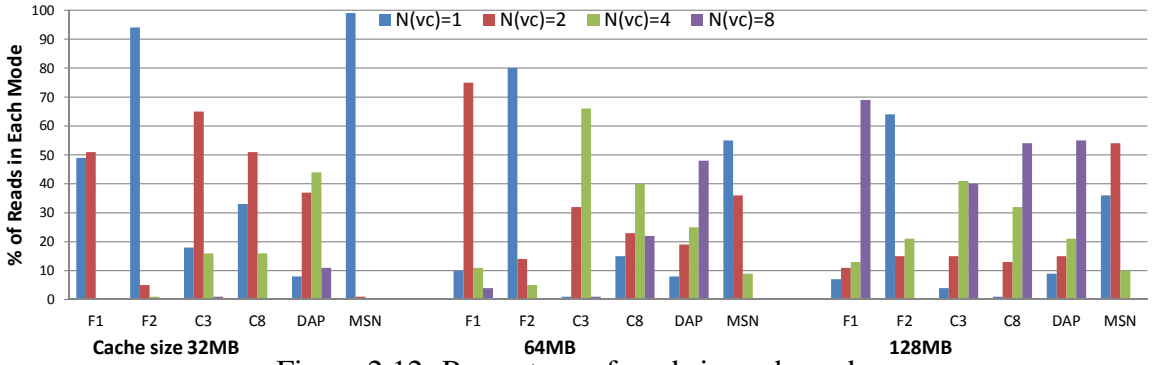


Figure 2.12: Percentage of reads in each mode.

To conclude the evaluation of DiffECC, we learned that un-controlled slow writes have negative affects on the write throughput performance; using an workload adaptive method of switching between slow and fast write modes, DiffECC successfully achieves a balance among the slow write ratio, write throughput, and read latency.

2.6 Summary

In this work, motivated by the NAND flash memory device write speed vs. raw storage reliability trade-off and run-time data access workload dynamics, we propose a cross-layer co-design approach that can jointly exploit these features to opportunistically reduce SSD read response latency. The key is to apply opportunistic memory write slowdown to enable the use of shorter and weaker ECCs, leading to largely reduced SSD read latency. A disk-level scheduling scheme has been developed to smooth the write workload to effectively enable opportunistic memory write slowdown. To demonstrate its effectiveness, we use 2 bits/cell NAND flash memory with BCH-based error correction codes as a test vehicle. We choose four different BCH coding systems. Extensive simulations over various workloads show that this cross-layer co-design solution can reduce the average SSD read latency by up to 59.4% at a cost of trivial overhead on the write throughput performance.

Chapter 3

Reducing SSD Access Latency via NAND Flash Program and Erase Suspension

3.1 Introduction

In NAND flash memory, once a page program or block erase (P/E) command is issued to a NAND flash chip, the subsequent read requests have to wait until the time-consuming P/E operation to complete. Preliminary results show that the lengthy P/E operations increase the read latency by 2x on average. This increased read latency caused by the contention may significantly degrade the overall system performance. Inspired by the internal mechanism of NAND flash P/E algorithms, we propose a low-overhead P/E suspension scheme, which suspends the on-going P/E to service pending reads and resumes the suspended P/E afterwards. Having reads enjoy the highest priority, we further extend our approach by

making writes be able to preempt the erase operations in order to improve the write latency performance.

3.2 Motivation

In this section, we demonstrate how the read vs. P/E contention increases the read latency under various workloads. We have modified *MS-add-on* simulator [3] based on Disksim 4.0. Specifically, under the workloads of a variety of popular disk traces, we compare the read latency of two scheduling policies, FIFO and read priority scheduling (RPS), to show the limitation of RPS. Furthermore, with RPS, we set the latency of program and erase operation to be equal to that of read and *zero* to justify the impact of P/E on the read latency.

3.2.1 A Simple Demonstration of Contention Effect

First of all, we illustrate the contention effect between reads and writes with an simple example. Figure 3.1 shows the timing diagram of one flash chip servicing three read requests (RD) and one write request (WT), of which the arrival time is marked on the top timeline. The raw latency of read and write is assumed to be one and five time units, respectively. Three scheduling policies, FIFO, RPS, and RPS+Suspension, are analyzed under this workload. With FIFO, both RD2 and RD3 are scheduled for service after the completion of WT1, resulting service latency of 6 and 4 units. RPS schedules RD2 ahead of WT1. However, RD3 has to wait until WT1 is serviced because suspension of write is not allowed. RPS+Suspension is our desired solution for this problem where WT1 is suspended for RD3.

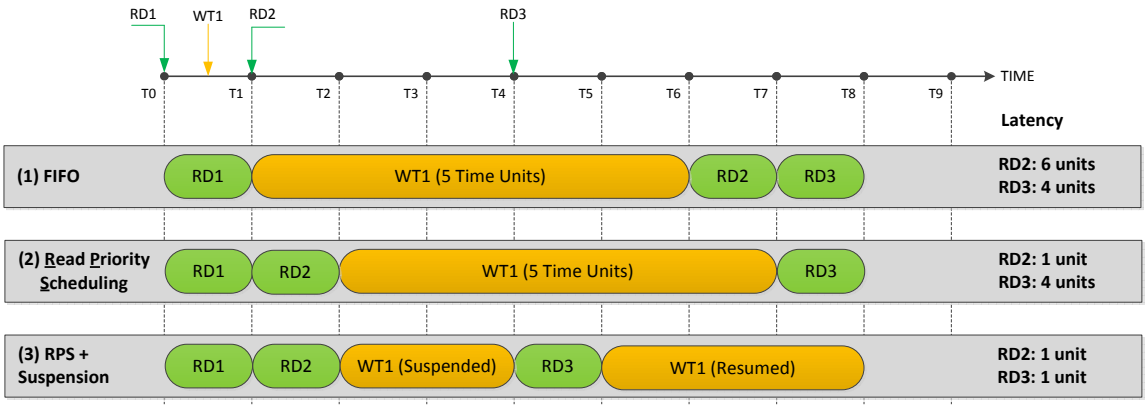


Figure 3.1: Timing diagram illustrating the read latency under the effect of chip contention.

3.2.2 Configurations and Workloads

The simulated SSD is configured as follows: there are 16 flash chips, each of which owns a dedicated channel to the flash controller. Each chip has four planes that are organized in a RAID-0 fashion; the size of one plane is 512 MB or 1 GB assuming the flash is used as SLC or 2-bit MLC, respectively (the page size is 2 KB for SLC or 4 KB for MLC). To maximize the concurrency, each individual plane has its own allocation pool [3]. The garbage collection processes are executed in the background so as to minimize the interference with the foreground requests. In addition, the percentage of flash space over-provisioning is set as 30%, which doubles the value suggested in [3]. Considering the limited working-set size of the workloads used in this work, 30% over-provisioning is believed to be sufficient to avoid frequent execution of garbage collection processes. The write buffer size is 64 MB. The SSD is connected to the host via a PCI-E of 2.0 GB/s. The physical operating parameters of the flash memory is summarized in Table 3.1.

We choose 4 disk I/O traces for our experiments: *Financial 1 and 2* (F1, F2) [67]; *Display Ads Platform and payload servers* (DAP) and *MSN storage metadata* (MSN) traces [35]. Those traces were originally collected on HDDs, to produce more stressful

Table 3.1: Flash Parameters

Symbols	Description	Value	
		SLC	MLC
T_{bus}	The bus latency of transferring one page from/to the chip	20 μs	40 μs
T_{r_phy}	The latency of sensing/reading data from the flash	10 μs	25 μs
T_{w_total}	The total latency of ISPP in flash page program	140 μs	660 μs
N_{w_cycle}	The number of ISPP iterations	5	15
T_{w_cycle}	The time of one ISPP iteration (T_{w_total}/N_{w_cycle})	28 μs	44 μs
$T_{w\_program}$	The duration of program phase of one ISPP iteration	20 μs	20 μs
T_{verify}	The duration of the verify phase	8 μs	24 μs
T_{erase}	The duration of erase pulse	1.5 ms	3.3 ms
$T_{voltage\_reset}$	The time to reset operating voltages of on-going operations	4 μs	
T_{buffer}	The time taken to load the page buffer with data	3 μs	

workloads for SSDs, we compress all these traces so that the system idle time is reduced from 98% to around 70% for each workload. Some basic information of selected traces is summarized in Table 3.2

Table 3.2: Disk Traces Information

Parameter	Reads(10^6)	Read %	Length(h)	Compression Ratio	Idle %
F1	1.23	23.2	12	X9	65.5
F2	3.04	82.3	12	X25	69.1
DAP	0.61	56.2	24	X8	63.9
MSN	0.40	75.0	6	X50	66.2

3.2.3 Experimental Results

In this subsection, we compare the read latency performance under four scenarios: FIFO; RPS; PER (the latency of program and erase is set equal to that of read); and PE0 (the latency of program and erase is set to zero). Note that both PER and PE0 are applied upon RPS in order to study the chip contention and the limitation of RPS. Due to the large range of the numerical values of the experimental results, we normalize them to the

Table 3.3: Numerical Latency Values of FIFO (in ms)

Trace	SLC		MLC	
	Read	Write	Read	Write
F1	0.37	0.87	0.44	1.58
F2	0.24	0.57	0.27	1.03
DAP	1.92	6.85	5.74	11.74
MSN	4.13	4.58	8.47	25.21

corresponding results of FIFO, which are listed in Table 3.3 for reference. The normalized results are plotted in Fig. 3.2, where the left part shows the results of SLC and the right part is for MLC. Compared to FIFO, RPS achieves impressive performance gain, e.g., the gain maximizes at an effective read latency ("effective" refers to the actual latency taking the queuing delay into account) reduction of 38.8% (SLC) and 45.7% (MLC) on average. However, if the latency of P/E is the same as read latency or zero, i.e., in the case of *PER* and *PE0*, the effective read latency can be further reduced. For example, with PE0, the read latency reduction is 64.2% (SLC) and 71.0% (MLC) on average. Thus, even with RPS policy, the chip contention still increases the read latency by about 2x on average.

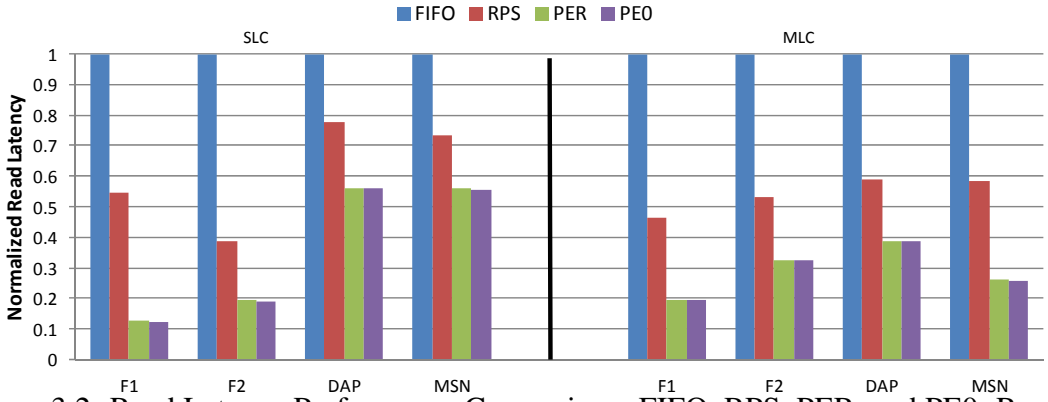


Figure 3.2: Read Latency Performance Comparison: FIFO, RPS, PER, and PE0. Results normalized to FIFO.

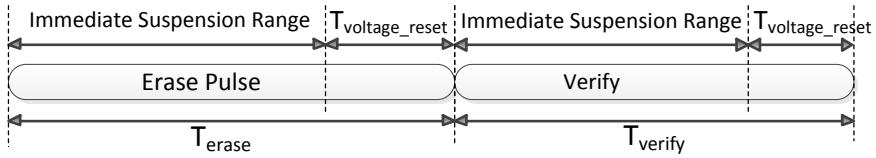
3.3 Design

In this section, the design of the implementation of P/E suspension is proposed in details. To realize the P/E suspension function, we seek for a low-cost solution, with which the user of NAND chip (the on-disk flash controller) only need to exploit this new flexibility by supporting the commands of P/E suspension and resumption while the actual implementation is done inside the chip.

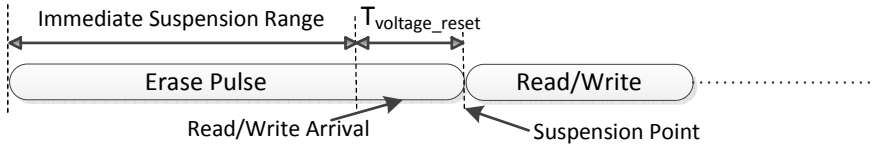
3.3.1 Erase Suspension and Resumption

In NAND flash, the erase process consists of two phases: first, an erase pulse lasting for T_{erase} is applied on the target block; second, a verify operation that takes T_{verify} is performed to check if the preceding erase pulse has successfully erased all bits in the block. Otherwise, the above process is repeated until success, or if the number of iterations reaches the predefined limit, an operation failure is reported. Typically, for NAND flash, since the *over-erasure* is not a concern [10], the erase operation can be done with a single erase pulse.

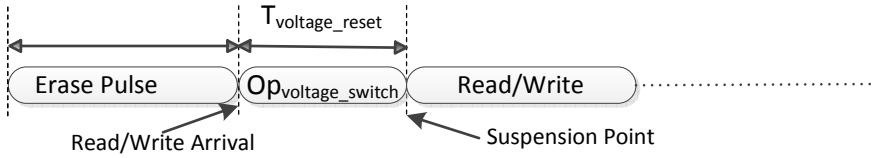
How to suspend an erase operation: suspending either the erase pulse or verify operation requires resetting the status of the corresponding wires that connect the flash cells with the analog block. Specifically, due to the fact that the flash memory works at different voltage bias for different operations, the current voltage bias applied on the wires (and thus on the cell) needs to be reset for the pending read request. This process ($Op_{voltage\_reset}$ for short) takes a period of $T_{voltage\_reset}$. Noting that either the erase pulse or verify operation always has to conduct $Op_{voltage\_reset}$ at the end (as shown in the following diagram of erase operation timeline).



Thus, if the suspension command arrives during $Op_{voltage\_reset}$, the suspension will succeed once $Op_{voltage\_reset}$ is finished (as illustrated in the following diagram of erase suspension timeline).



Otherwise, an $Op_{voltage\_reset}$ is executed immediately and then the read/write request is serviced by the chip (as illustrated in the following diagram).



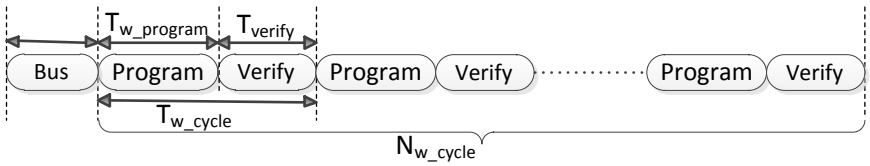
How to resume an erase operation: the resumption means the control logic of NAND flash resumes the suspended erase operation. Therefore, the control logic should keep track of the progress, i.e., whether the suspension happens during the verify phase or the erase pulse. For the first scenario, the verify operation has to be re-done all over again. For the second scenario, the erase pulse time left (T_{erase} minus the progress), for example, 1 ms will be done in the resumption if no more suspension happens. Actually, the task of progress tracking can be easily supported by the existing facilities in the control logic of NAND flash: the pulse width generator is implemented using a counter-like logic [10], which keeps track of the progress of the current pulse.

The overhead on the effective erase latency: resuming the erase pulse requires extra time to set the wires to the corresponding voltage bias, which takes approximately the same amount of time as $T_{voltage\_reset}$. Suspending during the verify phase causes a re-do in the resumption, and thus the overhead is the time of the suspended/cancelled verify operation. In addition, the read service time is included in the effective erase latency.

3.3.2 Program Suspension and Resumption

The process of servicing a program request is: first, the data to be written is transferred through the controller-chip bus and loaded in the page buffer; then the ISPP is executed, in which a total number of N_{w_cycle} iterations consisting of a *program* phase followed by a *verify* phase are conducted on the target flash page. In each ISPP iteration, the program phase is responsible for applying the required program voltage bias on the cells so as to charge them. In the verify phase, the content of the cells is read to verify if the desired amount of charge is stored in each cell: if so, the cell is considered *program-completion*; otherwise, one more ISPP iteration will be conducted on the cell. Due to the fact that all cells in the target flash page are programmed simultaneously, the overall time taken to program the page is actually determined by the cell that needs the most number of ISPP iterations. A major factor that determines the number of ISPP iterations needed is the amount of charge to be stored in the cell, which is in turn determined by the data to be written. For example, for the 2-bit MLC flash, programming a "0" in a cell needs the most number of ISPP iterations, while for "3" (the erased state), no ISPP iteration is needed. Since all flash cells in the page are programmed simultaneously, N_{w_cycle} is determined by the smallest data (2-bit) to be written; nonetheless, we make a rational assumption in

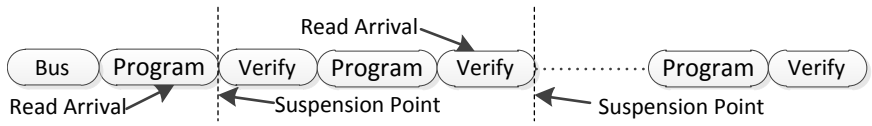
our simulation experiments that N_{w_cycle} is constant and equal to the maximum value. The program process is illustrated in the following diagram.



How to retain the page buffer content: before we move on to suspension, this critical problem has to be solved. For program, the page buffer contains the data to be written. For read, it contains the retrieved data to be transferred to the flash controller. If a write is preempted by a read, the content of the page buffer is certainly replaced. Thus, the resumption of the write demands the page buffer re-stored. Intuitively, the flash controller that is responsible for issuing the suspension and resumption commands may keep a copy of the write page data until the program is finished and upon resumption, the controller re-sends the data to the chip through the controller-chip bus. However, the page transfer consumes a significant amount of time: unlike the NOR flash which does *byte programming*, NAND flash does *page programming*, and the page size is of a few kilobytes. For instance, assuming a 100 MHz bus and 4 KB page size, the bus time T_{bus} is about $40\ \mu s$.

To overcome this overhead, we propose a *Shadow Buffer* in the flash. The shadow buffer serves like a replica of the page buffer and it automatically loads itself with the content of the page buffer upon the arrival of the write request and re-stores the page buffer while resumption. The load and store operation takes the time T_{buffer}. The shadow buffer has parallel connection with the page buffer, and thus the data transfer between them can be done on the fly. T_{buffer} is normally smaller than T_{bus} by one order of magnitude.

How to suspend a program operation: compared to the long width of the erase pulse (T_{erase}), the program and verify phase of the program process is normally two orders of magnitude shorter. Intuitively, the program process can be suspended at the end of the program phase of any ISPP iteration as well as the end of the verify phase. We refer to this strategy as "Inter Phase Suspension" (**IPS**). IPS has in total $N_{w\_cycle} * 2$ potential suspension points as illustrated in the following diagram.



Due to the fact that at the end of the program or verify phase, the status of the wires has already reset ($Op_{voltage\_reset}$), IPS does not introduce any extra overhead, except for the service time of the read or reads that preempt the program. However, the effective read latency should include the time from the arrival of read to the end of the corresponding phase. For simplicity, assuming the arrival time of reads follows the uniform distribution, the probability of encountering the program phase and the verify phase is $T_{w\_program}/(T_{verify} + T_{w\_program})$ and $T_{verify}/(T_{verify} + T_{w\_program})$, respectively. Thus, the average extra latency for the read can be calculated as:

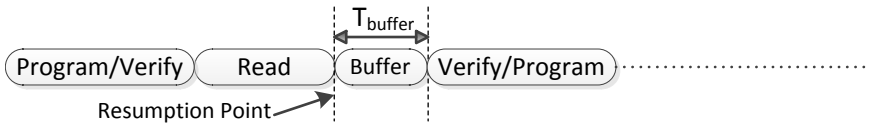
$$\begin{aligned} T_{read\_extra} = & \frac{T_{w\_program}}{(T_{verify}+T_{w\_program})} * \frac{T_{w\_program}}{2} \\ & + \frac{T_{verify}}{(T_{verify}+T_{w\_program})} * \frac{T_{verify}}{2} \end{aligned} \tag{3.1}$$

Substituting the numerical values in Table 3.1, we get $8.29\ \mu s$ (SLC) and $11.09\ \mu s$ (MLC) for T_{read_extra}, which is comparable to the physical access time of the read (T_{r_phy}). To further improve the effective read latency, we propose "Intra Phase Cancelation" (**IPC**). Similar to canceling the verify phase for the erase suspension, IPC cancels an on-going

program or verify phase upon suspension. The reason of canceling instead of pausing the program phase is that the duration of the program phase, $T_{w\_program}$, is short and normally considered atomic (cancelable but not pause-able).

Again, for IPC, if the read arrives when the program or verify phase is conducting $Op_{voltage\_reset}$, the suspension happens actually at the end of the phase, which is the same as IPS; otherwise, $Op_{voltage\_reset}$ is started immediately and the read is then serviced. Thus, IPC achieves a T_{read_extra} no larger than $T_{voltage\_reset}$.

How to resume from IPS: first of all, the page buffer is re-loaded with the content of the shadow buffer. Then, the control logic examines the last ISPP iteration number and the previous phase. If IPS happens at the end of the verify phase, which implies that the information of the status of cells has already been obtained, we may continue with the next ISPP if needed; on the other hand, if the last phase is the program phase, naturally we need to finish the verify operation before moving on to the next ISPP iteration. The resumption process is illustrated in the following diagram.



How to resume from IPC: compared to IPS, the resumption from IPC is more complex. Different from the verify operation, which does not change the charge status of the cell, the program operation puts charge in the cell and thus changes the threshold voltage (V_{th}) of the cell. Therefore, we need to determine whether the canceled program phase has already achieved the desired V_{th} (i.e., whether the data could be considered written in the cell), by a verify operation. If so, no more ISPP iteration is needed on

this cell; otherwise, the previous program operation is executed on the cell again. The later case is illustrated in the following diagram.

Re-doing the program operation would have some affect on the tightness of V_{th}, but with the aid of ECC and a fine-grained ISPP, i.e., small incremental voltage ΔV_{pp}, the IPC has little impact on the data reliability of the NAND flash. The relationship between ΔV_{pp} and the tightness of V_{th} is modeled in [77].

The overhead on the effective write latency: IPS requires re-loading the page buffer, which takes T_{buffer}. For IPC, if the verify phase is canceled, the overhead is the time elapsed of the canceled verify phase plus the read service time and T_{buffer}. In case of program phase, there are two scenarios: if the verify operation reports that the desired V_{th} is achieved, the overhead is the read service time plus T_{buffer}; otherwise, the overhead is the time elapsed of the canceled program phase plus an extra verify phase, in addition to the overhead of the above scenario. Clearly, IPS achieves smaller overhead on the write than IPC but relatively lower read performance.

3.4 Further Discussions

3.4.1 Scheduling Policy

We schedule the requests and suspension/resumption operations according to a priority-based policy. The highest priority is rendered to read requests, which are always scheduled ahead of writes and can preempt the on-going program and erase operations. The write

requests can preempt only the erase operations, giving that there is no read requests pending for service. We allow nested suspension operations, i.e., a read request may preempt a program operation, which has preempted an erase earlier. There are at most 2 operations in the suspension state.

The flash controller determines when to suspend and resume an on-going P/E according to the run-time request list. Intuitively, when a read request is received while P/E is currently being serviced by the chip (i.e., the chip has not yet reported "completion" to the flash controller), the controller issues a suspension command. Then after the response from the chip, the read/write request is committed to the chip. Upon completion of the read/write, the controller issues a resume command if there is no pending request. Otherwise, according to RPS, the pending reads should get serviced before the resumption, followed by the writes assuming an erase operation is in suspension state.

The above policy is a greedy one since the reads *always* preempt the on-going P/E. However, as discussed earlier in this section, the overhead of suspension and resumption as well as the service time of the reads increase the write latency, which in turn increases the chance of data corruption resulted from system halt. Thus, the controller must limit the overhead by stoping giving the reads with the high priority (both RPS and P/E suspension) when the overhead exceeds a predefined threshold. Noting that, although the effective write latency could be prolonged by RPS, the overall throughput is barely compromised since that the total chip bandwidth consumed by the requests remains constant with either FIFO or RPS. In addition, suspending erase operations for writes may delay the garbage collection processes, which would risk the adequacy of available flash space. Therefore, we allow *write-suspend-erase* only when the ratio of clean space is above the pre-defined watermark.

3.4.2 Implementation Issues

The proposed P/E suspension requires the support of the flash controller as well as the on-chip control logic. The flash controller needs to support the new commands in order to interface the scheduling algorithms (as mentioned in the previous subsection) and the chip. The control logic, where our design is deployed, involves a few modifications/add-on's summarized as following:

- The decision-making for the suspension point is needed.
- Extra logic is needed to keep track of the progress of erase pulse and ISPP upon suspension.
- The algorithm of P/E should be modified to support the resumption process, especially for the program.
- The added shadow buffer consumes a portion of chip area and needs support from the P/E algorithm.
- About *data integrity*: In case of *read after write* on the same page and the program/write is being suspended, a recent copy of the data for the suspended write request is kept in the write buffer, which ensures the subsequent reads always get the fresh copy. At any time, there is at most one write suspended in a flash plane, which demands minimum resources to ensure the read/write order. In case that the write buffer failed to render the fresh data, the read operation returns the shadow buffer content.
- The completion of suspension should be reported through the chip pins.

3.4.3 The Overhead on Power Consumption

The overhead of the P/E suspension scheme on power consumption in the flash chip comes from two sources. First, in the scenario of resuming one erase operation, the wires (connecting the flash cells and P/E circuits) are recharged to the voltage for erase operations. The power consumed here is trivial compared to that consumed by the large current to erase the blocks. Second, in the **IPC** technique, the verify phase or program pulse is occasionally repeated upon resumption for writes. The power overhead in this case depends on the number of write suspension operations (as shown in Figure 3.8, where the maximum percentage of suspended writes is about 13% in the case of MLC). As we have barely observed multiple suspensions occurred to a single write process in the experiments, the related power overhead is upper-bounded by a percentage of $9\% * (1/5) = 1.8\%$, for SLC; or $13\% * (1/15) = 0.9\%$, for MLC.

3.5 Evaluation

In this section, the proposed P/E suspension design is simulated with the same configuration and parameters as in Section 3.2. Under the workloads of the four traces used in Section 3.2, we evaluate the read/write latency performance gain and the overhead of P/E suspension. We demonstrate that the proposed design achieves a near-optimal read performance gain and the write performance is significantly improved as well.

3.5.1 Read Performance Gain

First, we compare the average read latency of P/E suspension with RPS, PER and PE0 in Fig. 3.3, where the results are normalized to that of RPS. For P/E suspension, the IPC (Intra

Phase Cancelation), denoted as "PES_IPC", is adopted in Fig. 3.3. PE0, with which the physical latency values of program and erase are set to zero, serves as an optimistic situation where the contention between reads and P/E's is completely eliminated. Fig. 3.3 demonstrates that, compared to RPS, the proposed P/E suspension achieves a significant read performance gain, which is almost equivalent to the optimal case, PE0 (with less than 1% difference). Specifically, on the average of the 6 traces, PES_IPC reduces the read latency by 44.8% for SLC and 46.5% for MLC compared to RPS, and 64.1% for SLC and 70.7% for MLC compared to FIFO. For conciseness, the results of SLC and (then) MLC are listed without explicit specification in the following text.

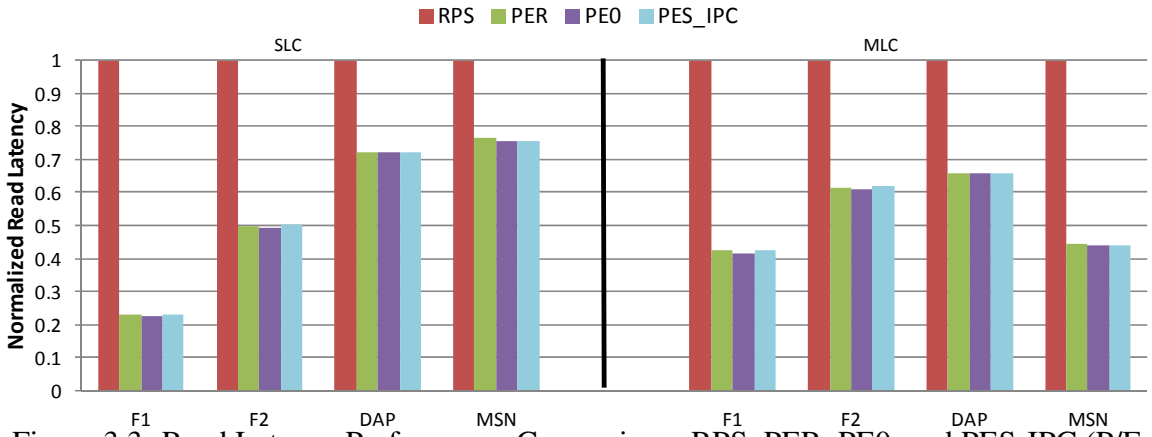


Figure 3.3: Read Latency Performance Comparison: RPS, PER, PE0, and PES_IPC (P/E Suspension using IPC). Normalized to RPS.

We take a closer look at the difference between PE0 and PES_IPC in Fig. 3.4, where PES_IPC is normalized to PE0. With IPC scheme, P/E suspension requires extra time to suspend the on-going write or erase, e.g., the $Op_{voltage\_reset}$. Thus, there is 1.05% and 0.83% on average and at-most 2.64% and 2.04% read latency increase of PES_IPC, compared to PE0. Comparing the results of SLC to that of MLC, SLC has a slightly larger difference from PE0. This is due to the fact that SLC's physical read latency is smaller, and thus more prone to the overhead introduced by $Op_{voltage\_reset}$. In both cases, differences under DAP

and MSN are almost zero because that these two traces are dominated by large-sized read requests.

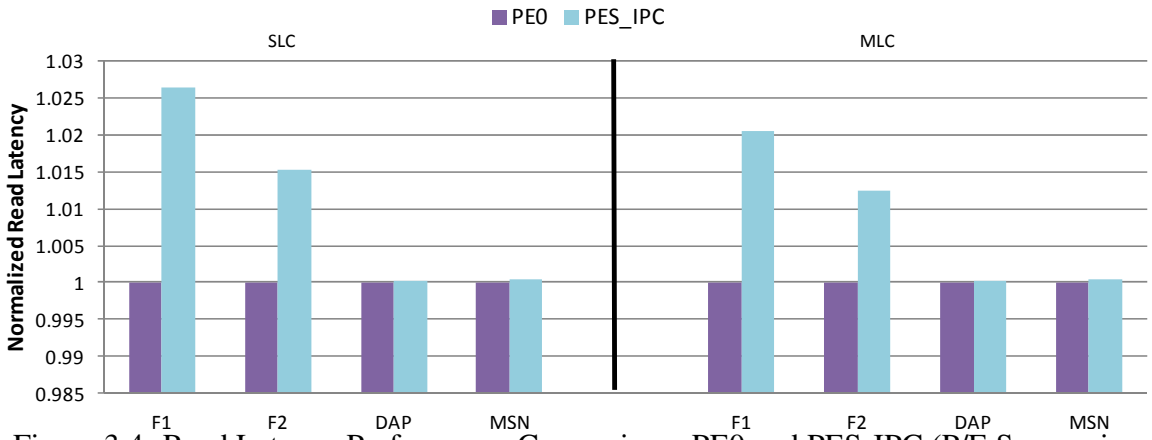


Figure 3.4: Read Latency Performance Comparison: PE0 and PES_IPC (P/E Suspension using IPC). Normalized to PE0.

As stated in Section 3.3, IPC can achieve better read performance but higher write overhead compared to IPS. We evaluate the read performance of IPC and IPS in Fig. 3.5, where the results are normalized to IPC. The read latency of IPS is 8.12% and 3.18% on average and at-most 13.24% and 6.74% (under F1) higher than that of IPC. The difference is resulted from the fact that IPS has extra read latency, which is mostly the time between read request arrivals and the suspension points located at the end of the program or verify phase. The latency difference between IPC and IPS of each trace is roughly proportional to the occurrence rate of suspension operations that happen on the program (noting that for suspension of erase operations, IPC and IPS share the same scheme). We can observe that the latency performance of IPS using SLC is poorer than the MLC case, under all traces, and this is because of the higher sensitivity of SLC's read latency to the overhead caused by the extra latency.

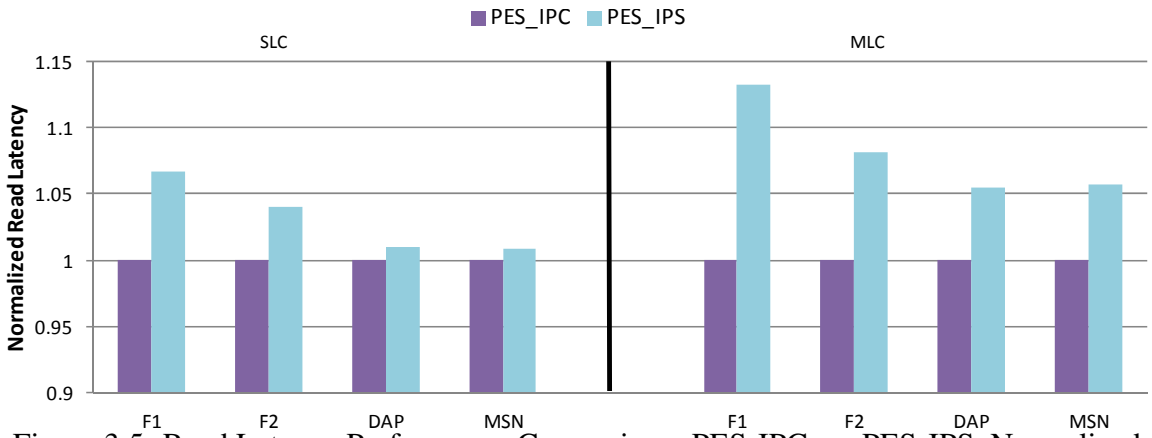


Figure 3.5: Read Latency Performance Comparison: PES_IPC vs. PES_IPS. Normalized to PES_IPC.

3.5.2 Write Performance

In this subsection, we begin with the analysis about the overhead on write latency caused by P/E suspension of read requests. Afterwards, we enable *Write-Suspend-Erase* to eliminate the overhead and boost the write performance.

Without Write-Suspend-Erase: The Overhead on Write Latency

Under the fore-mentioned workloads, either RPS or P/E suspension introduces significant extra chip bandwidth usage and thus the write throughput is barely compromised. Here we use the latency as a metric for the overhead evaluation. First, we compare the average write latency of FIFO, RPS, PES_IPS, and PES_IPC in Fig. 3.6, where the results are normalized to that of FIFO.

As illustrated in Fig. 3.6, write overhead in terms of latency is trivial compared to the read performance gain we achieved with P/E suspension; nontheless, P/E suspension schemes, especially the IPC, has a higher overhead than RPS. Specifically, RPS increases the write latency by 3.96% and 2.81% on average and at-most 6.65% (SLC, MSN) and

3.80% (MLC, DAP), compared to FIFO. PES_IPS increases write latency by 4.49% and 3.14% on average and at-most 6.91% (SLC, MSN) and 4.29%(MLC, DAP), respectively.

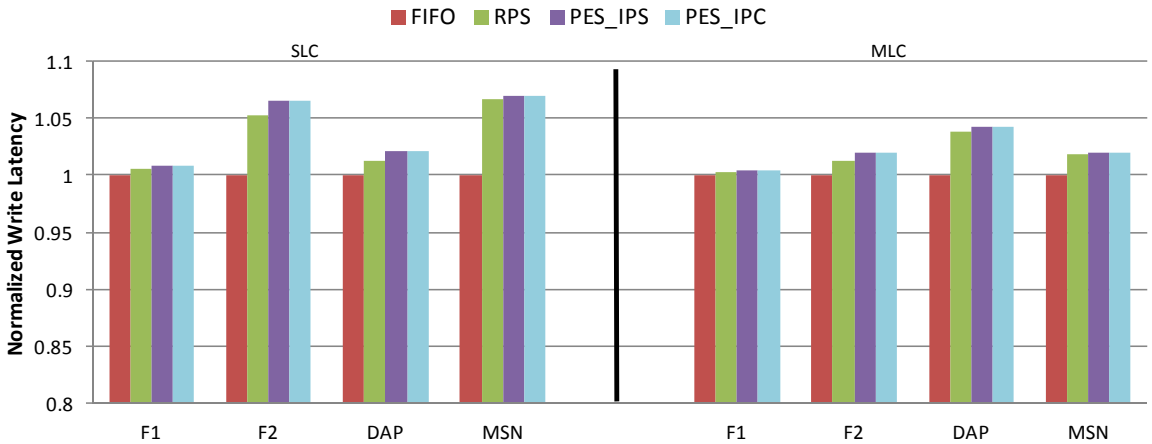


Figure 3.6: Write Latency Performance Comparison: FIFO, RPS, PES_IPC, and PES_IPS. Normalized to FIFO.

Compared to RPS, the overhead of P/E suspension is mostly contributed by the service time of the reads that preempted the on-going write or erase. Thus, although IPC added overhead by extra $Op_{voltage\_reset}$ in addition to the page buffer re-loading, which is the only overhead introduced by IPS, the write latency difference between IPS and IPC is relatively smaller than the difference between IPS and RPS. This point is observed clearly in Fig. 3.6 under all traces.

Furthermore, we observe that the overhead of the two P/E suspension schemes roughly approximates that of RPS under F1, DAP, and MSN, while the deviation is larger under F2. We examine the access pattern of the traces and found that in F2, read requests arrive in larger batches than those in F1, DAP, and MSN. The scheduling policy of proposed P/E suspension actually boost the probability of RPS, i.e., when the read that preempted the P/E has been serviced, the controller will keep on prioritize reads pending in the queue before resuming P/E. This implies under F2, once a read preempts an on-going P/E, the following reads that come in batches with it would be serviced before P/E gets resumed. Thus, the

write overhead in this scenario is larger (in addition to overhead caused by RPS), and this complies with our observation. We further justify this point by comparing the original P/E latency reported by the device with latency after suspension in Fig. 3.7, where the results on the y axis represents the increased percentage of write latency. As we can see, F2 obtains larger overhead on suspended writes than the other three.

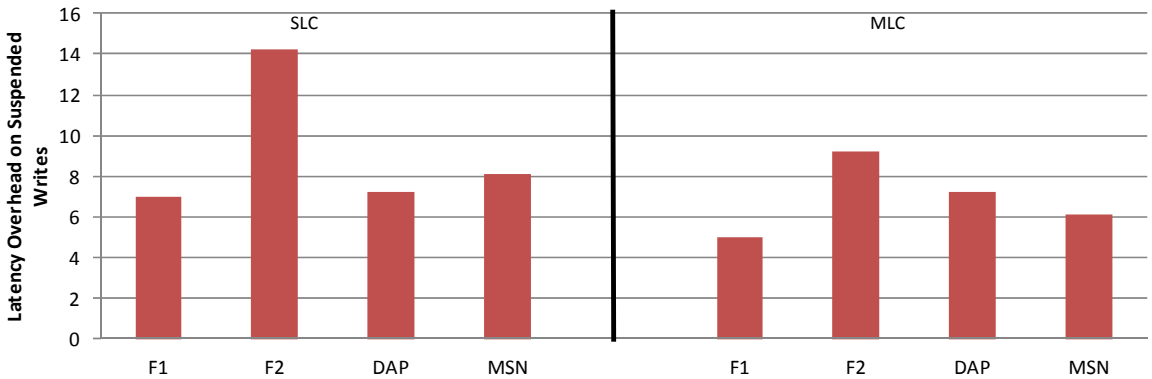


Figure 3.7: Compare the original write latency with the effective write latency resulted from P/E Suspension. Y axis represents the percentage of increased latency caused by P/E suspension.

In addition to the read service time resulted from the boosted RPS, the other factor that affects the write overhead is the percentage of P/E that had been suspended, which is presented in Fig. 3.8. There is 4.87% and 7.39% on average and at-most 8.92% (SLC, F2) and 12.65% (MLC, MSN) of the P/E that had been suspended. Referring to Table 3.2, we can observe that the percentage numbers here are roughly proportional to the percentage of read requests in each trace, e.g., F2 and MSN have the largest percentage of read requests as well as that of the suspended P/E.

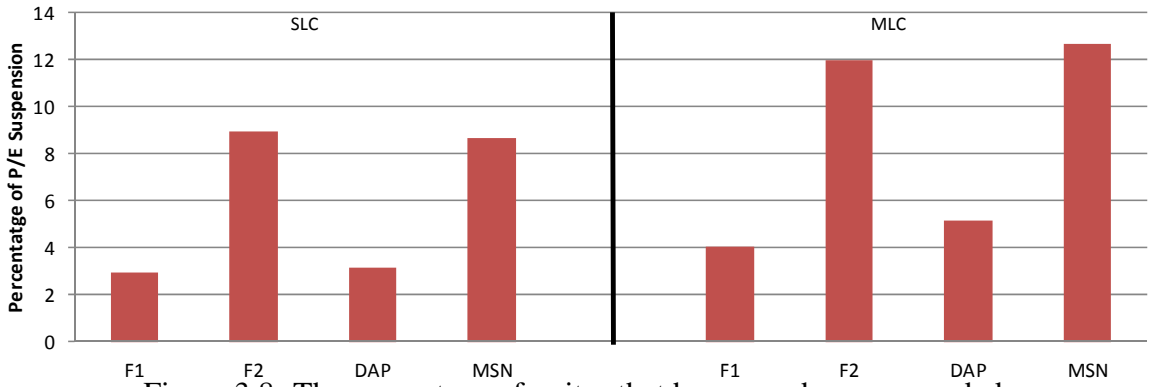


Figure 3.8: The percentage of writes that have ever been suspended.

Without Write-Suspend-Erase: the Sensitivity of Write Latency to the Write Queue Size

In this subsection, we learn the sensitivity of write overhead to the maximum write queue size. In order to obtain an amplified write overhead, we select F2, which has the highest percentage of read requests, and compress the simulation time of F2 by 7 times to intensify the workload. In Figure 3.9 we present the write latency results of RPS and PES_IPC (normalized to that of FIFO) varying the maximum write queue size from 16 to 512. Clearly, the write overhead of both RPS and PES_IPC is sensitive to the maximum write queue size, which suggests that the flash controller should limit the write queue size to control the write overhead. Noting that, relative to RPS, the PES_IPC has a near-constant increase on the write latency, which implies that the major contributor of overhead is RPS when the queue size is varied.

Enable Write-Suspend-Erase: The Write Performance Gain

In order to eliminate the overhead on write latency, we enable the feature of Write-Suspend-Erase. The effectiveness of this feature is illustrated in Figure 3.10, where the write latency of *PES_IPC with Write-Suspend-Erase* is normalized to that of FIFO. Compared to FIFO,

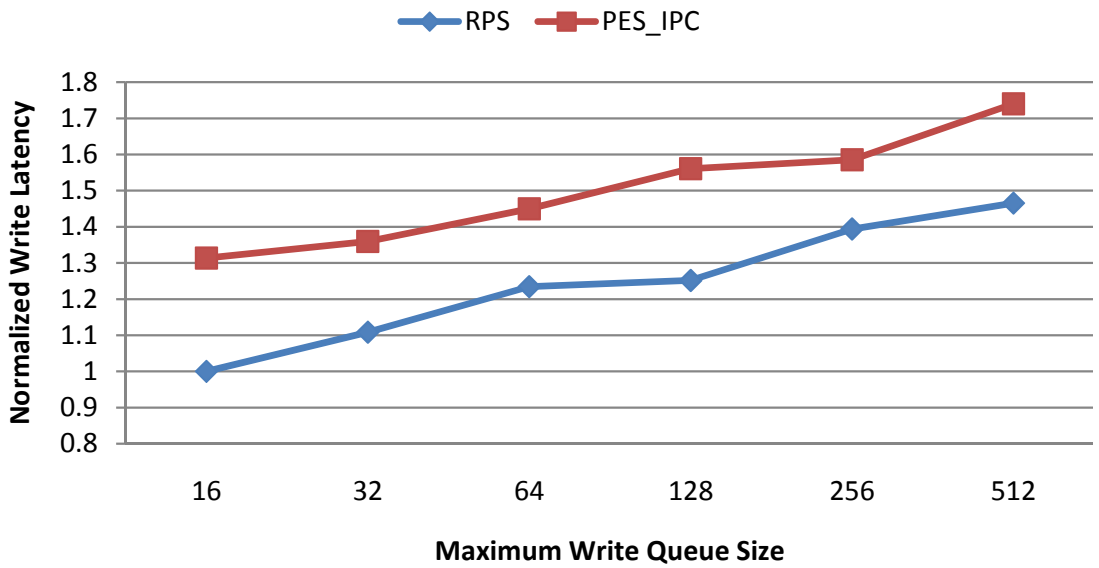


Figure 3.9: The write latency performance of RPS and PES_IPC while the maximum write queue size varies. Normalized to FIFO.

our approach reduces the write latency by 13.6% and 11.2% on average, i.e., the write overhead caused by reads is balanced and we may even outperform FIFO.

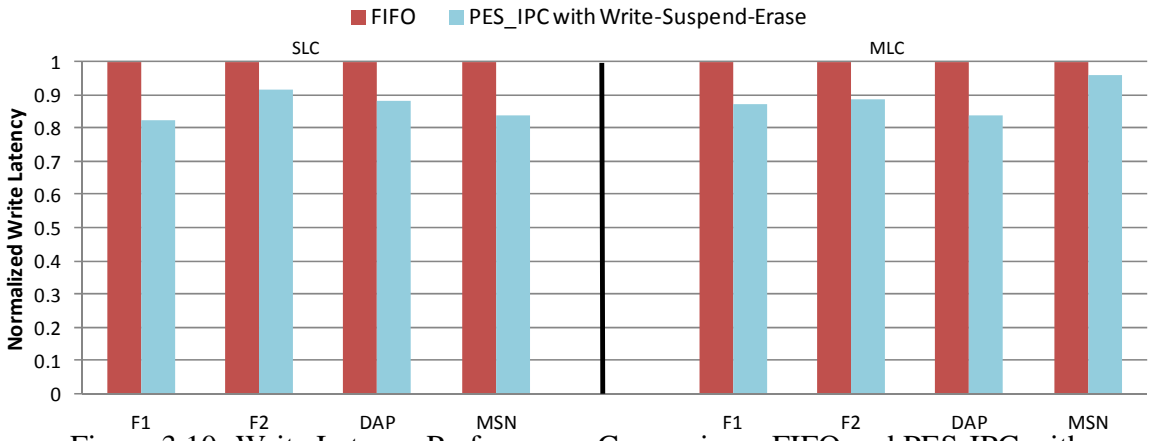


Figure 3.10: Write Latency Performance Comparison: FIFO and PES_IPC with Write-Suspend-Erase enabled. Normalized to FIFO.

3.6 Summary

One performance problem of NAND flash is that its program and erase latency is much higher than the read latency. This problem causes the chip contention between reads and P/Es due to the fact that with current NAND flash interface, the on-going P/E cannot be suspended and resumed. To alleviate the impact of the chip contention on the read performance, in this chapter we propose a light-overhead *P/E suspension* scheme by exploiting the internal mechanism of P/E algorithm in NAND flash. We further apply this idea to enable writes to preempt erase operations in order to reduce the write latency. The design is simulated/evaluated with precise timing and realistic SSD modeling of multi-chip/channel. Experimental results show that the proposed P/E suspension significantly reduces the read and write latency.

Chapter 4

Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality

4.1 Introduction

The limited lifetime of SSD is a major drawback that hinders its deployment in reliability sensitive environments [4, 7]. As pointed out in [7], "endurance and retention (of SSDs) is not yet proven in the field" and integrating SSDs into commercial systems is "painfully slow". The reliability problem of SSDs mainly comes from the following facts. Flash memory must be erased before it can be written and it may only be programmed/erased for a limited times (5K to 100K) [21]. In addition, the out-of-place writes result in invalid pages to be discarded by garbage collection (GC). Extra writes are introduced in GC operations to move valid pages to a clean block [3] which further aggravates the lifetime problem of SSDs.

Existing approaches for this problem mainly focus on two perspectives: 1) to prevent early defects of flash blocks by wear-leaving [65, 14]; 2) to reduce the number of write

operations on the flash. For the later, various techniques are proposed including in-drive buffer management schemes [37, 29, 33, 76] to exploit the temporal or spatial locality; FTLs (Flash Translation Layer) [38, 41, 32, 23] to optimize the mapping policies or garbage collection schemes to reduce the write-amplification factor; or data deduplication [15, 22] to eliminate writes of existing content in the drive.

This work aims to efficiently solve this lifetime issue from a different aspect. We propose a new FTL scheme, ΔFTL, to reduce the write count via exploiting the content locality. The content locality has been observed and exploited in memory systems [24], file systems [16], and block devices [54, 79, 80]. Content locality means data blocks, either blocks at distinct locations or created at different time, share *similar contents*. We exploit the content locality that exists between the new (the content of update write) and the old version of page data mapped to the same logical address. This content locality implies the new version resembles the old to some extend, so that the difference (*delta*) between them can be compressed compactly. Instead of storing new data in its original form in the flash, ΔFTL stores the compressed deltas to reduce the number of writes.

4.2 Related Work Exploiting the Content Locality

The content locality implies that the data in the system share similarity with each other. Such similarity can be exploited to reduce the memory or storage usage by delta-encoding the difference between the selected data and its reference. Content locality has been leveraged in various level of the system. In virtual machine environments, VMs share a significant number of *identical* pages in the memory, which can be deduplicated to reduce the memory system pressure. Difference engine [24] improves the performance

over deduplication by detecting the *nearly* identical pages and coalesce them via in-core compression [50] into much smaller memory footprint. Difference engine detects similar pages based on hashes of several chucks of each page: hash collisions are considered as a sign of similarity. Different from difference engine, GLIMPSE [51] and DERD system [16] work on the file system to leverage similarity across files; the similarity detection method adopted in these techniques is based on Rabin fingerprints over chunks at multiple offsets in a file. In the block device level, Peabody [54] and TRAP-Array [80] are proposed to reduce the space overhead of storage system backup, recovery, and rollback via exploiting the content locality between the previous (old) version of data and the current (new) version. Peabody mainly focuses on eliminating duplicated writes, i.e., the update write contains the same data as the corresponding old version (silent write) or sectors at different location (coalesced sectors). On the other hand, TRAP-Array reduces the storage usage of data backup by logging the compressed XORs (delta) of successive writes to each data block. The intensive content locality in the block I/O workloads produces a small compression ratio on such deltas and TRAP-Array is significantly space-efficient compared to traditional approaches. I-CASH [79] takes the advantage of content locality existing across the entire drive to reduce the number of writes in the SSDs. I-CASH stores only the reference blocks on the SSDs while logs the delta in the HDDs.

Our approach ΔFTL is mostly similar to the idea of TRAP-Array [80] , which exploits the content locality between new and old version. The major differences are: 1) ΔFTL aims at reducing the number of program/erase (**P/E**) operations committed to the flash memory so as to extend SSD's lifetime, instead of reducing storage space usage involved in data backup or recovery. Technically, the history data are backed up in TRAP-Array while they are considered "invalid" and discarded in ΔFTL; 2) ΔFTL is an embedded

software in the SSD to manage the allocation and de-allocation of flash space, which requires relative complex data structures and algorithms that are "flash-aware". It also requires that the computation complexity should be kept minimum due to limited micro-processor capability.

4.3 Delta-FTL Design

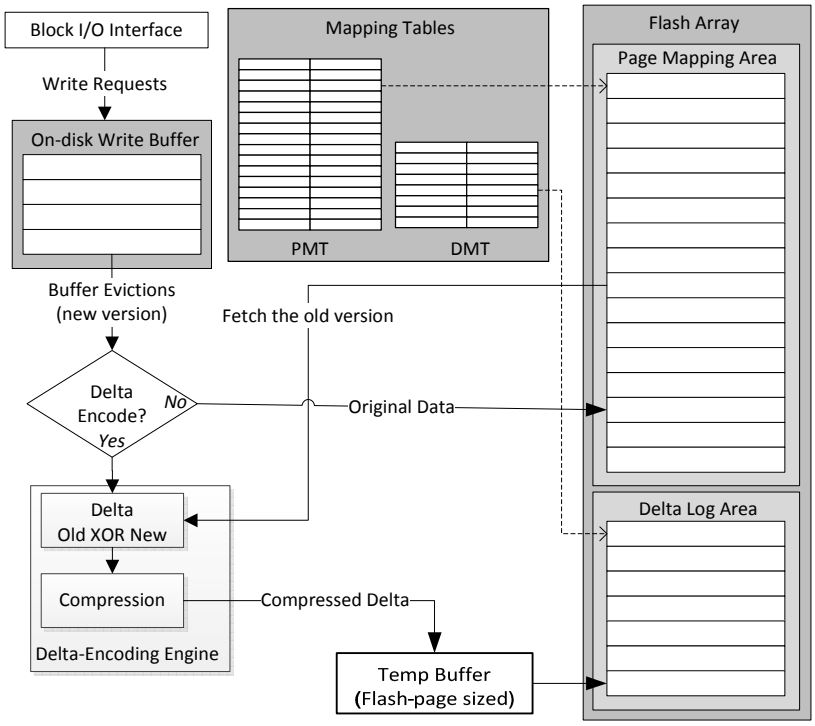


Figure 4.1: ΔFTL Overview

ΔFTL is designed as a flash management scheme that can store the write data in form of compressed deltas on the flash. Instead of devising from scratch, ΔFTL is rather an enhancement to the framework of the popular page-mapping FTL like DFTL [23]. Figure 4.1 gives an overview of ΔFTL and unveils its major differences from a regular page-mapping FTL:

- First, ΔFTL has a dedicated area, *Delta Log Area* (DLA), for logging the compressed deltas.

- Second, the compressed deltas must be associated with their corresponding old versions to retrieve the data. An extra mapping table, *Delta Mapping Table* (DMT), collaborates with *Page Mapping Table* (PMT) to achieve this functionality.

- Third, ΔFTL has a *Delta-encoding Engine* to derive and then compress the delta between the write buffer evictions and their old version on the flash. We have a set of dispatching rules determining whether a write request is stored in its original form or in its "delta-XOR-old" form. For the first case, the data is written to a flash page in page mapping area in its original form. For the later case, the delta-encoding engine derives and then compresses the delta between old and new. The compressed deltas are buffered in a flash-page-sized *Temp Buffer* until the buffer is full. Then, the content of the temp buffer is committed to a flash page in delta log area.

4.3.1 Dispatching Policy: Delta Encode?

The content locality between the new and old data allows us to compress the delta, which has rich information redundancy, to a compact form. Writing the compressed deltas rather than the original data, would indeed reduce the number of flash writes. However, delta-encoding all data indiscriminately would cause overheads.

First, if a page is stored in "delta-XOR-old" form, this page actually requires storage space for both delta and the old page, compared to only one flash page if in the original form. The extra space is provided by the overprovisioning area of the drive [3]. To make a trade-off between the overprovisioning resource and the number of writes, ΔFTL favors the

data that are overwritten frequently. This policy can be interpreted intuitively with a simple example: in a workload, page data A is only overwritten once while B is overwritten 4 times. Assuming the compression ratio is 0.25, delta-encoding A would reduce the number of write by $3/4$ page (compared to the baseline which would take one page write) at a cost of $1/4$ page in the overprovision space. Delta-encoding B, on the other hand, reduces the number of write by $4 * (3/4) = 3$ pages at the same cost of space. Clearly, we would achieve better performance/cost ratio with such write "hot" data rather than the cold ones. The approach taken by ΔFTL to differentiate hot data from cold ones is discussed in Section 4.3.4.

Second, fulfilling a read request targeting a page in "delta-XOR-old" form requires two flash page reads. This may have reverse impact on the read latency. To alleviate this overhead, ΔFTL avoids delta-encoding pages that are read intensive. If a page in "delta-XOR-old" form is found read intensive, ΔFTL will merge it to the original form to avoid the reading overhead.

Third, the delta-encoding process involves operations to fetch the old, derive delta, and compress delta. This extra time may potentially add overhead to the write performance (discussed in Section 4.3.2). ΔFTL must cease delta-encoding if it would degrade the write performance.

To summarize, ΔFTL delta-encodes data that are *write-hot* but *read-cold* while ensuring the write performance is not degraded.

4.3.2 Write Buffer and Delta-encoding

The in-drive write buffer resides in the volatile memory (SRAM or DRAM) managed by an SSD's internal controller and shares a significant portion of it [37, 29, 33]. The write

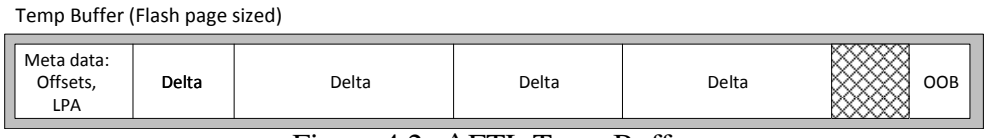


Figure 4.2: ΔFTL Temp Buffer

buffer absorbs repeated writes and improves the spatial locality of the output workload from it. We concentrate our effort on FTL design, which services write buffer's outputs, and adopt simple buffer management schemes like FIFO or SLRU [34] that are usual in disk drives. When buffer eviction occurs, the evicted write pages are dispatched according to our dispatching policy discussed above to either ΔFTL's *Delta-encoding Engine* or directly to the page mapping area.

Delta-encoding engine takes the new version of the page data (i.e., the evicted page) and the corresponding old version in page mapping area, as its inputs. It derives the delta by XOR the new and old version and then compress the delta. The compressed delta are buffered in *Temp Buffer*. *Temp Buffer* is of the same size as a flash page. Its content will be committed to delta log area once it is full or there is no space for the next compressed delta. Splitting a compressed delta on two flash pages would involve in unnecessary complications for our design. Storing multiple deltas in one flash page requires meta-data, like LPA (logical page address) and the offset of each delta (as shown in Figure 4.2) in the page, to associate them with their old versions and locate the exact positions. The meta-data is stored at the MSB part of a page instead of attached after the deltas, for the purpose of fast retrieval. This is because the flash read operation always buses out the content of a page from its beginning [56]. The content of temp buffer described here is essentially what we have in flash pages of delta log area.

Delta-encoding engine demands the computation power of SSD's internal micro-processor and would introduce overhead for write requests. We discuss the delta-encoding

latency in Section 4.3.2 and the approach adopted by ΔFTL to control the overhead in Section 4.3.2.

Delta-encoding Latency: Delta-encoding involves two steps: to derive delta (XOR the new and old versions) and to compress it. Among many data compression algorithms, the lightweight ones are favorable for ΔFTL due to the limited computation power of the SSD's internal micro-processor. We investigate the latency of a few candidates, including Bzip2 [64], LZO [55], LZF [45], Snappy [19], and Xdelta [50], by emulating the execution of them on the ARM platform: the source codes are cross-compiled and run on the *SimpleScalar-ARM* simulator [49]. The simulator is an extension to SimpleScalar supporting ARM7 [5] architecture and we configured a processor similar to ARM®Cortex R4 [1], which inherits ARM7 architecture. For each algorithm, the number of CPU cycles is reported and the latency is then estimated by dividing the cycle number by the CPU frequency. We select LZF (LZF1X-1) from the candidates because it makes a good trade-off between speed and compression performance, plus a compact executable size. The average number of CPU cycles for LZF to compress and decompress a 4KB page is about 27212 and 6737, respectively. According to Cortex R4's write paper, it can run at a frequency from 304MHz to 934MHz. The latency values in μs are listed in Table 4.1. An intermediate frequency value (619MHz) is included along with the other two to represent three classes of micro-processors in SSDs.

Table 4.1: Delta-encoding Latency

Frequency(MHz)	304	619	934
Compression(μs)	89.5	44.0	29.1
Decompression(μs)	22.2	10.9	7.2

Discussion: Write Performance Overhead: ΔFTL's delta-encoding is a two-step procedure. First, delta-encoding engine fetches the old version from the page mapping

area. Second, the delta between the old and new data are derived and compressed. The first step consists of raw flash access and bus transmission, which exclusively occupy the flash chip and the bus to the micro-processor, respectively. The second step occupies exclusively the micro-processor to perform the computations. Naturally, these three elements, the flash chip, the bus, and micro-processor, forms a simple pipeline, where the delta-encoding procedures of a serial of write requests could be overlapped. An example of four writes is demonstrated in Figure 4.3, where T_{delta_encode} is the longest phase. This is true for a micro-processor of 304MHz or 619MHz assuming T_{read_raw} and T_{bus} take 25μs and 40μs (Table 4.3), respectively. A list of symbols used in this section is summarized in Table 4.2.

Table 4.2: List of Symbols

Symbols	**Description**
n	Number of pending write pages
P_c	Probability of compressible writes
R_c	Average compression ratio
T_{write}	Time for page write
T_{read_raw}	Time for raw flash read access
T_{bus}	Time for transferring a page via bus
T_{erase}	Time to erase a block
T_{delta_encode}	Time for delta-encoding a page
B_s	Block size (pages/block)
N	Total Number of page writes in the workload
T	Data blocks containing invalid pages (baseline)
t	Data blocks containing invalid pages (ΔFTL's PMA)
PE_{gc}	The number of P/E operations done in GC
F_{gc}	GC frequency
OH_{gc}	Average GC overhead
G_{gc}	Average GC gain (number of invalid pages reclaimed)
S_{cons}	Consumption speed of available clean blocks

Table 4.3: Flash Access Latency

Parameter	**Value**
Flash Read/Write/Erase	$25\mu s/200\mu s/1.5ms$
Bus Transfer Time	$40\mu s$

For an analytical view of the write overhead, we assume there is a total number of n write requests pending for a chip. Among these requests, the percentage that is considered

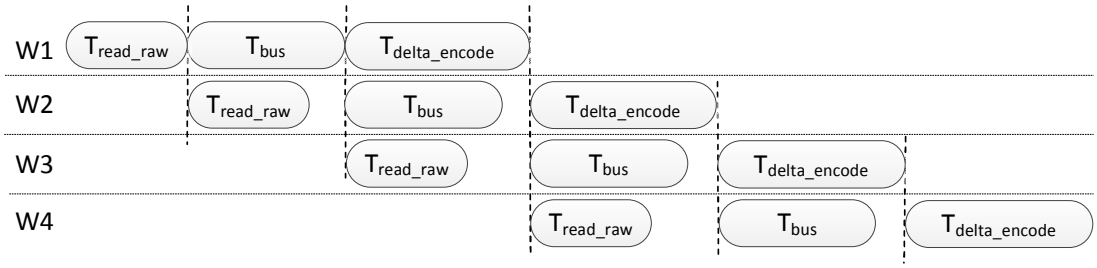


Figure 4.3: ΔFTL Delta-encoding Timeline

compressible according to our dispatching policy is P_c and the average compression ratio is R_c. The delta-encoding procedure for these n requests takes a total time of:

$MAX(T_{read\_raw}, T_{bus}, T_{delta\_encode}) * n * P_c$

The number of page writes committed to the flash is the sum of original data writes and compressed delta writes: $(1 - P_c) * n + P_c * n * R_c$. For the baseline, which always outputs the data in their original form, the page write total is n. We define that the write overhead exists if ΔFTL's write routine takes more time than the baseline. Thus, there is *no* overhead if the following expression is true:

$$\begin{gathered} MAX(T_{read\_raw}, T_{bus}, T_{delta\_encode}) * n * P_c + \\ ((1 - P_c) * n + P_c * n * R_c) * T_{write} < n * T_{write} \end{gathered} \tag{4.1}$$

Expression 4.1 can be simplified to:

$$1 - Rc > \frac{MAX(T_{read\_raw}, T_{bus}, T_{delta\_encode})}{T_{write}} \tag{4.2}$$

Substituting the numerical values in Table 4.1 and Table 4.3, the right side of Expression 4.2 is 0.45, 0.22, and 0.20, for micro-processor running at 304, 619, and 934MHz, respectively. Therefore, the viable range of R_c should be smaller than 0.55, 0.78, and 0.80. Clearly, high performance micro-processor would impose a less restricted constraint on R_c. If R_c is

out of the viable range due to weak content locality in the workload, in order to eliminate the write overhead, ΔFTL must switch to the baseline mode where the delta-encoding procedure is bypassed.

4.3.3 Flash Allocation

ΔFTL's flash allocation scheme is an enhancement to the regular page mapping FTL scheme with a number of flash blocks dedicated to store the compressed deltas. These blocks are referred to as *Delta Log Area* (DLA). Similar to page mapping area (PMA), we allocate a clean block for DLA so long as the previous active block is full [3]. The garbage collection policy will be discussed in Section 4.3.5.

DLA cooperates with PMA to render the latest version of one data page if it is stored as *delta-XOR-old* form. Obviously, read requests for such data page would suffer from the overhead of fetching two flash pages. To alleviate this problem, we keep the track of the read access popularity of each delta. If one delta is found read-popular, it is merged with the corresponding old version and the result (data in its original form) is stored in PMA.

Furthermore, as discussed in Section 4.3.1, write-cold data should not be delta-encoded in order to save the overprovisioning space. Considering the temporal locality of a page may last for only a period in the workload, if a page previously considered write-hot is no longer demonstrating its temporal locality, this page should be transformed to its original form from its delta-XOR-old form. ΔFTL periodically scans the write-cold pages and merges them to PMA from DLA if needed.

4.3.4 Mapping Table

The flash management scheme discussed above requires ΔFTL to associate each valid delta in DLA with its old version in PMA. ΔFTL adopts two mapping tables for this purpose: *Page Mapping Table* (PMT) and *Delta Mapping Table* (DMT).

Page mapping table is the primary table indexed by logical page address (LPA) of 32bits. For each LPA, PMT maps it to a physical page address (PPA) in page mapping area, either the corresponding data page is stored as its original form or in delta-XOR-old form. For the later case, the PPA points to the old version. PMT differentiates this two cases by prefixing a flag bit to the 31bits PPA (which can address 8TB storage space assuming a 4KB page size). As demonstrated in Figure 4.4: if the flag bit is "1", which means this page is stored in delta-XOR-old form, we use the PPA (of the old version) to consult the delta mapping table and find out on which physical page the corresponding delta resides. Otherwise, the PPA in this page mapping table entry points to the original form of the page. DMT does not maintain the offset information of each delta in the flash page; we locate the exact position with the metadata prefixed in the page (Figure 4.2).

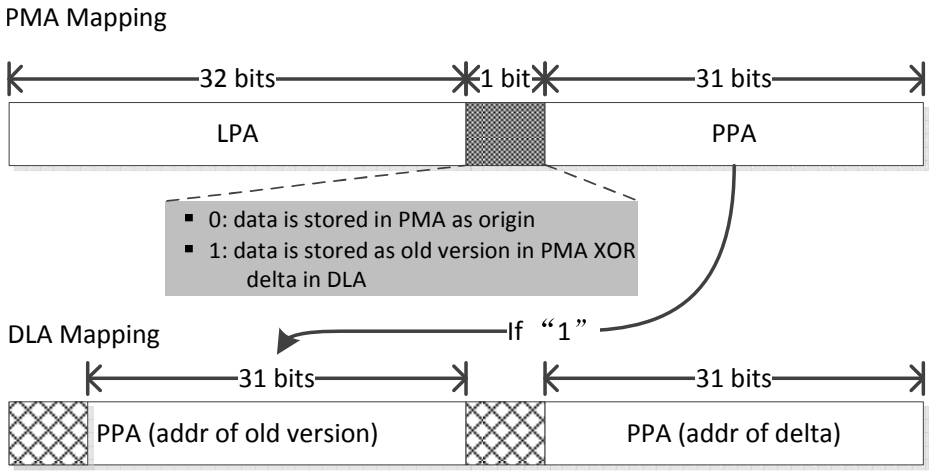


Figure 4.4: ΔFTL Mapping Entry

Store Mapping Tables On the Flash: ΔFTL stores both mapping tables on the flash and keeps an *journal* of update records for each table. The updates are first buffered in the in-drive RAM and when they grow up to a full page, these records are flushed to the journal on the flash. In case of power failure, a built-in capacitor or battery in the SSD (e.g., a SuperCap [72]) may provide the power to flush the un-synchronized records to the flash. The journals are merged with the tables periodically.

Cache Mapping Table In the RAM: ΔFTL adopts the same idea of caching popular table entries in the RAM as DFTL [23], as shown in Figure 4.5(a). The cache is managed using *segment LRU* scheme (SLRU) [34]. Different from two separate tables on the flash, the mapping entries for data either in the original form or delta-XOR-old form are included in one SLRU list. For look-up efficiency, we have all entries indexed by the LPA. Particularly, entries for data in delta-XOR-old form associate the LPA with PPA of old version and PPA of delta, as demonstrated in Figure 4.5(b). When we have an address look-up miss in the mapping table cache and the target page is in delta-XOR-old form, both on-flash tables are consulted and we merge the information together to an entry as shown in Figure 4.5(b).

As discussed in Section 4.3.3, the capability of differentiating write-hot and read-hot data is critical to ΔFTL. We have to avoid delta-encoding the write-cold or read-hot data and merge the delta and old version of one page if it is found read-hot or found no longer write-hot. To keep the track of read/write access frequency, we associate each mapping entry in the cache with an *access count*. If the mapping entry of a page is found having a read-access (or write-access) count larger or equal to a predefined threshold, we consider this page read-hot (or write-hot) and vice versa. In our prototyping implementation, we set this threshold as 2 and it captures the temporal locality for both read and writes successfully

in our experiments. This information is forwarded to the dispatching policy module to guide the destination of a write request. In addition, merge operations take place if needed.

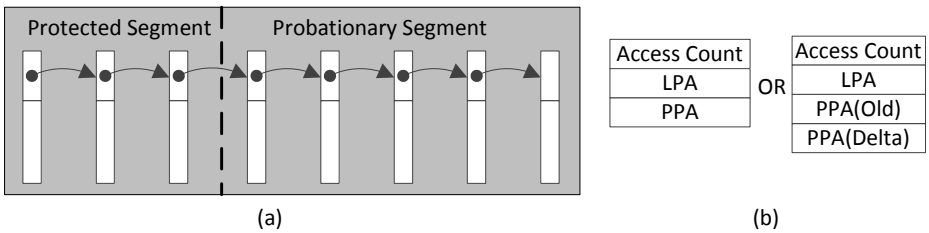


Figure 4.5: ΔFTL Buffered Mapping Entry

4.3.5 Garbage Collection

Overwrite operations causes invalidation of old data, which the garbage collection engine is required to discard when clean flash blocks are short. GC engine copies the valid data on the victim block to a clean one and erase the victim thereafter. ΔFTL selects victim blocks based on a simple "greedy" policy, i.e., blocks having the most number of invalid data result in the least number of valid data copy operations and the most clean space reclaimed [36]. ΔFTL's GC victim selection policy does not differentiate blocks from page mapping area or delta log area. In delta log area, the deltas becomes *invalid* in the following scenarios:

- If there is a new write considered not compressible (the latest version will be dispatched to PMA), according to the dispatching policy, the corresponding delta of this request and the old version in PMA become invalid.

- If the new write is compressible and thus a new delta for the same LPA is to be logged in DLA, the old delta becomes invalid.

- If this delta is merged with the old version in PMA, either due to read-hot or write-cold, it is invalidated.
- If there is a TRIM command indicating that a page is no longer in use, the corresponding delta and the old version in PMA are invalidated.

For any case, ΔFTL maintains the information about the invalidation of the deltas for GC engine to select the victims. In order to facilitate the merging operations, when a block is selected as GC victim, the GC engine will consult the mapping table for information about the access frequency of the valid pages in the block. The GC engine will conduct necessary merging operations while it is moving the valid pages to the new position. For example, for a victim block in PMA, GC engine finds out a valid page is associated with a delta which is read-hot, then this page will be merged with the delta and mark the delta as invalidated.

4.4 Discussion: SSD Lifetime Extension of ΔFTL

Analytical discussion about ΔFTL's performance on SSD lifetime extension is given in this section. In this chapter, we use the number of program and erase operations executed to service the write requests as the metric to evaluate the lifetime of SSDs. This is a common practice in most existing related work targeting SSD lifetime improvement [15, 23, 66, 79]. This is because the estimation of SSDs' lifetime is very challenging due to many complicated factors that would affect the actual number of write requests an SSD could handle before failure, including implementation details the device manufacturers would not unveil. On the other hand, comparing the P/E counts resulted from our approach to the baseline is relatively a more practical metric for the purpose of performance evaluation. Write amplification is a well-known problem for SSDs: due to the out-of-place-update

feature of NAND flash, the SSDs have to take multiple flash write operations (and even erase operations) in order to fulfill one write request. There are a few factors that would affect the write amplification, e.g., the write buffer, garbage collection, wear leveling, etc [28]. We focus on garbage collection for our discussion, providing that the other factors are the same for ΔFTL and the regular page mapping FTLs. We breakdown the total number of P/E operations into two parts: the **foreground** writes issued from the write buffer (for the baseline) or ΔFTL's dispatcher and delta-encoding engine; the **background** page writes and block erase operations involved in GC processes. Symbols introduced in this section are listed in Table 4.2.

4.4.1 Foreground Page Writes

Assuming for one workload, there is a total number of N page writes issued from the write buffer. The baseline has N foreground page writes while ΔFTL has $(1-P_c)*N+P_c*N*R_c$ (as discussed in Section 4.3.2). ΔFTL would resemble the baseline if P_c (percentage of compressible writes) approaches 0 or R_c (average compression ratio of compressible writes) approaches 1, which means the temporal locality or content locality is weak in the workload.

4.4.2 GC Caused P/E Operations

The P/E operations caused by GC processes is essentially determined by the frequency of GC and the average overhead of each GC, which can be expressed as:

$$PE_{gc} \propto F_{gc} * OH_{gc} \tag{4.3}$$

GC process is triggered when clean flash blocks are short in the drive. Thus, the GC frequency is proportional to *the consumption speed of clean space* and inversely proportional to *the average number of clean space reclaimed of each GC* (GC gain):

$$F_{gc} \propto \frac{S_{cons}}{G_{gc}} \tag{4.4}$$

Consumption Speed is actually determined by the number of foreground page writes (N for the baseline). *GC Gain* is determined by the average number of invalid pages on each GC victim block.

GC P/E of The Baseline

First, let's consider the baseline. Assuming for the given workload, all write requests are overwrites to existing data in the drive, then N page writes invalidate a total number of N existing pages. If these N invalid pages spread over T data blocks, the average number of invalid pages (thus GC Gain) on GC victim blocks is N/T. Substituting into Expression 4.4, we have the following expression for the baseline:

$$F_{gc} \propto \frac{N}{N/T} = T \tag{4.5}$$

For each GC, we have to copy the valid pages (assuming there are B_s pages/block, we have $B_s - N/T$ valid pages on each victim block on average) and erase the victim block. Substituting into Expression 4.3, we have:

$$PE_{gc} \propto T * (Erase + Program * (B_s - N/T)) \tag{4.6}$$

GC P/E of ΔFTL

Now let's consider ΔFTL's performance. Among N page writes issued from the write buffer, $(1 - P_c) * N$ pages are committed in PMA causing the same number of flash pages in PMA to be invalidated. Assuming there are t blocks containing invalid pages caused by those writes in PMA, we apparently have $t \leq T$. The average number of invalid pages in PMA is then $(1 - P_c) * N/t$. On the other hand, $P_c * N * R_c$ pages containing compressed deltas are committed to DLA. Recall that there are three scenarios where the deltas in DLA get invalidated (Section 4.3.5). Omitting the last scenario which is rare compared to the first two, the number of deltas invalidated is determined by the *overwrite rate* (P_{ow}) of deltas committed to DLA: while we assume in the workload all writes are overwrites to existing data in the drive, this *overwrite rate* here defines the percentage of deltas that are overwritten by the subsequent writes in the workload. For example, no matter the subsequent writes are incompressible and committed to PMA or otherwise, the corresponding delta gets invalidated. The average invalid space (in the term of pages) of victim block in DLA is thus $P_{ow} * B_s$. Substituting these numbers to Expression 4.4: If the average GC gain in PMA outnumbers that in DLA, we have:

$$F_{gc} \propto \frac{(1 - P_c + P_c R_c)N}{(1 - P_c)N/t} = t(1 + \frac{P_c R_c}{1 - P_c}) \tag{4.7}$$

Otherwise, we have:

$$F_{gc} \propto \frac{(1 - P_c + P_c R_c)N}{P_{ow} B_s} \tag{4.8}$$

Substituting Expression 4.7 and 4.8 to Expression 4.3, we have for GC introduced P/E:

$$PE_{gc} \propto t(1 + \frac{P_c R_c}{1-P_c}) * \\ (Erase + Program * (B_s - (1 - P_c)N/t)) \tag{4.9}$$

or:

$$PE_{gc} \propto \frac{(1-P_c+P_c R_c)N}{P_{ow} B_s} * \\ (T_{erase} + T_{write} * B_s(1 - P_{ow})) \tag{4.10}$$

4.4.3 Summary

From above discussions, we observe that ΔFTL favors the disk I/O workloads that demonstrate: (i) High content locality that results in small R_c; (ii) High temporal locality for writes that results in large P_c and P_{ow}. Such workload characteristics are widely present in various OLTP applications such as TPC-C, TPC-W, etc [80, 79, 67, 35].

4.5 Performance Evaluation

We have implemented and evaluated our design of ΔFTL based on a series of comprehensive trace-driven simulation experiments. In this section, we present the experimental results comparing ΔFTL with the page mapping FTL as the baseline. Essentially, the number of foreground writes and the efficiency of GC are reflected by the number of GC operations. Thus, in this section we use the number of GC operations as the major metric to evaluate ΔFTL's performance on extending SSD's lifetime. In addition, we evaluate the overheads ΔFTL may potentially introduce, including read and write performance. Particularly, read/write performance is measured in terms of response time.

4.5.1 Simulation Tool and SSD Configurations

ΔFTL is a device-level software in the SSD controller. We have implemented it (as well as the baseline page mapping FTL) in an SSD simulator based on the Microsoft Research SSD extension [3] for DiskSim 4.0. The simulated SSD is configured as follows: there are 16 flash chips, each of which owns a dedicated channel to the flash controller. Each chip has four planes that are organized in a RAID-0 fashion; the size of one plane is 1GB assuming the flash is used as 2-bit MLC (page size is 4KB). To maximize the concurrency, each individual plane has its own allocation pool [3]. The garbage collection processes are executed in the background so as to minimizing the interference upon the foreground requests. In addition, the percentage of flash space over-provisioning is set as 30%, which doubles the value suggested in [3]. Considering the limited working-set size of the workloads used in this work, 30% over-provisioning is believed to be sufficient to avoid garbage collection processes to be executed too frequently. The garbage collection threshold is set as 10%, which means if the clean space goes below 10% of the exported space, the garbage collection processes are triggered. Due to negligible impact that the write buffer size has on ΔFTL's performance compared to the baseline, we only report the results with buffer size of 64MB. The SSD is connected to the host via a PCI-E bus of 2.0 GB/s. In addition, the physical operating parameters of the flash memory are summarized in Table 4.3.

4.5.2 Workloads

We choose 6 popular disk I/O traces for the simulation experiments. *Financial 1* and *Financial 2* (F1, F2) [67] were obtained from OLTP applications running at two large

financial institutions; the *Display Ads Platform and payload servers* (DAP-PS) and *MSN storage metadata* (MSN-CFS) traces were from the Production Windows Servers and described in [35] (MSN-CFS trace contains I/O requests on multiple disks and we only use one of them); the *Cello99* [27] trace pool is collected from the "Cello" server that runs HP-UX 10.20. Because the entire *Cello99* is huge, we randomly use one day traces (07/17/99) of two disks (Disk 3 and Disk 8). Table 4.4 summarizes the traces we use in our simulation.

Table 4.4: Disk Traces Information

	Reads(10^6)	Read %	Writes	Write %	Duration(h)
F1	1.23	23.2	4.07	76.8	12
F2	3.04	82.3	0.65	17.7	12
C3	0.75	35.3	1.37	64.7	24
C8	0.56	27.4	1.48	72.6	24
DAP	0.61	56.2	0.47	43.8	24
MSN	0.40	75.0	0.13	25.0	6

4.5.3 Emulating the Content Locality

As pointed out in [54, 16, 80, 15], the content locality of a workload is application specific and different applications may result in distinctive extent of content locality. In this chapter, instead of focusing on only the workloads possessing intensive content locality, we aim at exploring the performance of ΔFTL under diverse situations. The content locality as well as temporal locality are leading factors that have significant impact on ΔFTL's performance. In our trace-driven simulation, we explore various temporal locality characteristics via 6 disk I/O traces; on the other hand, we emulate the content locality by assigning randomized compression ratio values to the write requests in the traces. The compression ratio values follows Gaussian distribution, of which the average equals R_c. Referring to the values of R_c reported in [80] (0.05 to 0.25) and in [54] (0.17 to 0.6),

we evaluate three levels of content locality in our experiments, having $R_c = 0.50, 0.35,$ and 0.20 to represent low, medium, and high content locality, respectively. In the rest of this section, we present the experimental results under 6 traces and three levels of content locality, comparing ΔFTL with the baseline.

4.5.4 Experimental Results

To verify the performance of ΔFTL, we measure the number of garbage collection operations and foreground writes, the write latency, and overhead on read latency.

Number of Garbage Collection Operations and Foreground Writes

First, we evaluate the number of garbage collection operations as the metric for ΔFTL's performance on extending SSD lifetime. Due to the large range of the numerical values of the experimental results, we normalize them to the corresponding results of the baseline as shown in Figure 4.6. Clearly, ΔFTL significantly reduces the GC count compared to the baseline: ΔFTL results in only 58%, 46%, and 33% of the baseline GC count on average, for $R_c = 0.50, 0.35, 0.20$ respectively. ΔFTL's maximum performance gain (22% of baseline) is achieved with C3 trace when $R_c = 0.20$; the minimum (82%) is with F1, $R_c = 0.50$. We may observe from the results that the performance gain is proportional to the content locality, which is represented by the average compression ratio R_c; in addition, ΔFTL performs relatively poorer with two traces F1 and F2, compared to the rests. In order to interpret our findings, we examine two factors that determine the GC count: the consumption speed of clean space (S_{cons}, Expression 4.4) and the speed of clean space reclaiming, i.e., the average GC gain (G_{gc}). **Consumption Speed:** As discussed in Section 4.4, the consumption speed is determined by the number of foreground flash page

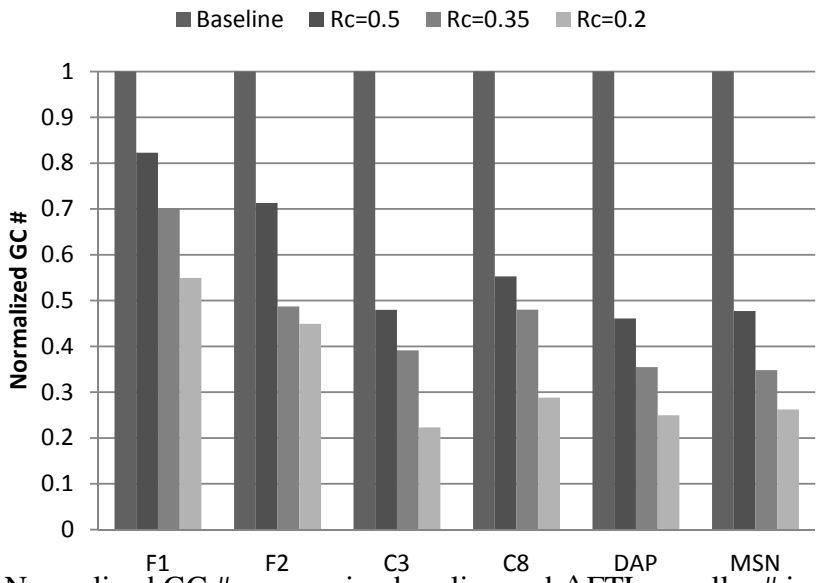


Figure 4.6: Normalized GC #: comparing baseline and ΔFTL; smaller # implies longer SSD lifetime.

writes. We plot the normalized number of foreground writes in Figure 4.7. As seen in the figure, the results are proportional to R_c as well; F1 and F2 produce more foreground writes than the others, which result in larger GC counts as shown in Figure 4.6. If there are N writes in the baseline, ΔFTL would have $(1 - P_c + P_c * R_c) * N$. The foreground write counts are reversely proportional to R_c (self-explained in Figure 4.7) as well as P_c. So, what does P_c look like? Recall in Section 4.3.1 that P_c is determined by the dispatching rules, which favor write-hot and read-cold data. The access frequency characteristics, i.e., the temporal locality, is workload-specific, which means the P_c values should be different among traces but not affected by R_c. This point is justified clearly in Figure 4.8, which plots the ratio of DLA writes (P_c) out of the total foreground writes. We may also verify that the foreground write counts (Figure 4.7) are reversely proportional to P_c: F1 and F2 have the least P_c values among all traces and they produce the most number of foreground writes; this trend can be also observed with other traces. **Garbage collection gain** is another factor that determines GC count. Figure 4.9 plots the average GC gain in terms

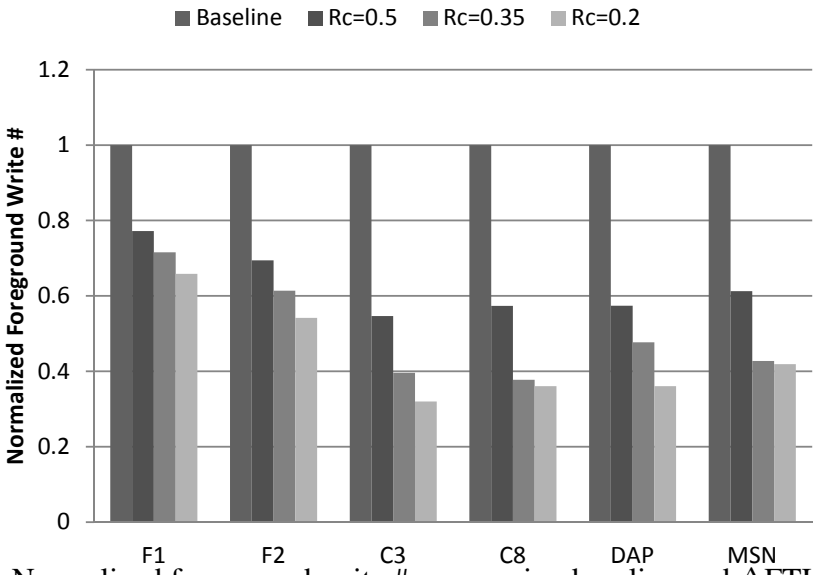


Figure 4.7: Normalized foreground write #: comparing baseline and ΔFTL; smaller # implies: a) larger P_c and b) lower consumption speed of clean flash space.

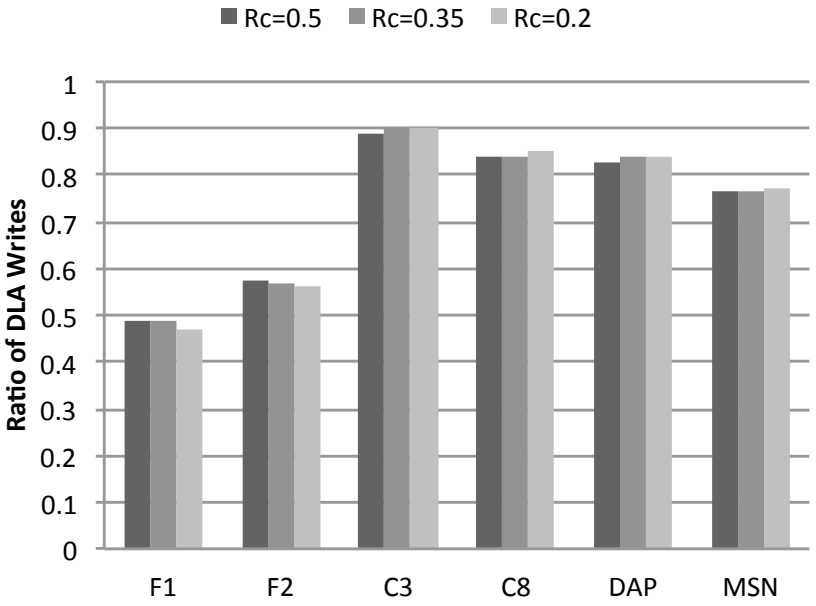


Figure 4.8: Ratio of DLA writes (P_c).

of the number of invalid pages reclaimed. GC gain ranges from 14 (C8, baseline) to 54 (F2, $R_c = 0.20$). F1 and F2 outperform the other traces on the average GC gain. However, comparing to the baseline performance, ΔFTL actually does not improve much with F1 and F2: we normalize each trace's results with its individual baseline in Figure 4.10. ΔFTL even degrades average GC gain with F1 and F2 when $R_c = 0.50$. This also complies with the GC count results shown in Figure 4.6, where ΔFTL achieves poorer performance gain with F1 and F2 compared to the others. The reason why ΔFTL does not improve GC

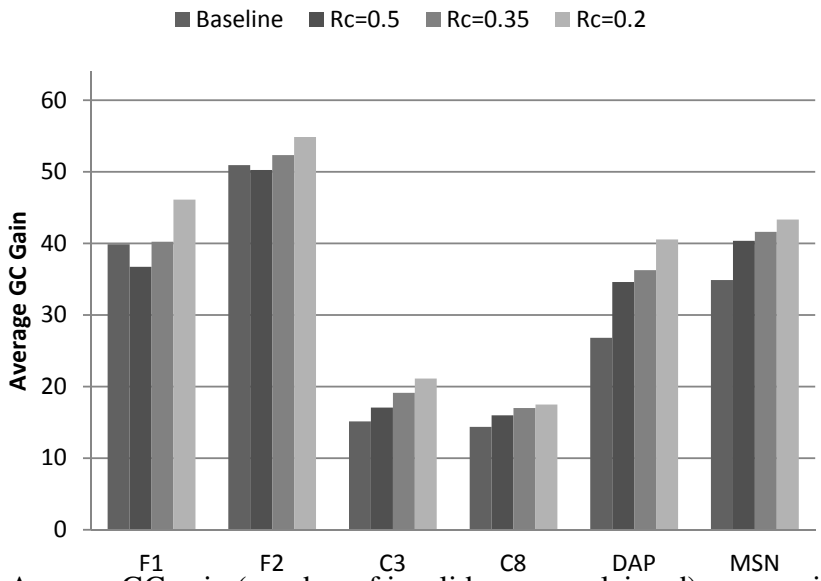


Figure 4.9: Average GC gain (number of invalid pages reclaimed): comparing baseline and ΔFTL; smaller # implies lower GC efficiency on reclaiming flash space.

gain significantly over the baseline with F1 and F2 is: compared to the other traces, F1 and F2 result in larger invalid page counts in blocks of PMA, which makes ΔFTL's GC engine to choose more PMA blocks as GC victims than DLA blocks. Thus, the average GC gain performance of ΔFTL resembles the baseline. To the contrary, ΔFTL benefits from the relative higher temporal locality of write requests in the DLA than in the PMA, under the other 4 traces. This is the reason why ΔFTL outperforms the baseline with these traces. In order to verify this point, we collect the number of GC executed in DLA and plot the ratio

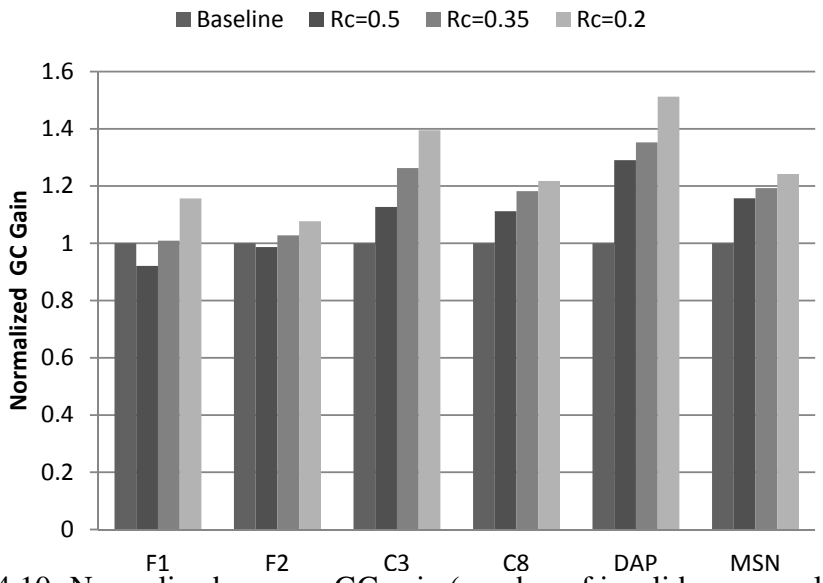


Figure 4.10: Normalized average GC gain (number of invalid pages reclaimed): comparing baseline and ΔFTL.

over the total in Figure 4.11: the majority of the total GC operations lies in PMA for F1 and F2 and in DLA for the rest.

Write Performance

In ΔFTL, the delta-encoding procedure in servicing a write request may cause overhead on write latency if R_c is out of the viable range (Section 4.3.2). R_c values adopted in our simulation experiments ensures there is no write overhead. ΔFTL significantly reduces the foreground write counts, and the write latency performance also benefits from this. As shown in Figure 4.12, ΔFTL reduces the average write latency by 36%, 47%, and 51% when $R_c = 0.50, 0.35, 0.20$, respectively.

Garbage Collection Overhead

The GC operation involves copying the valid data from the victim block to a clean block and erasing the victim block. The GC overhead, i.e., the time for a GC operation, may

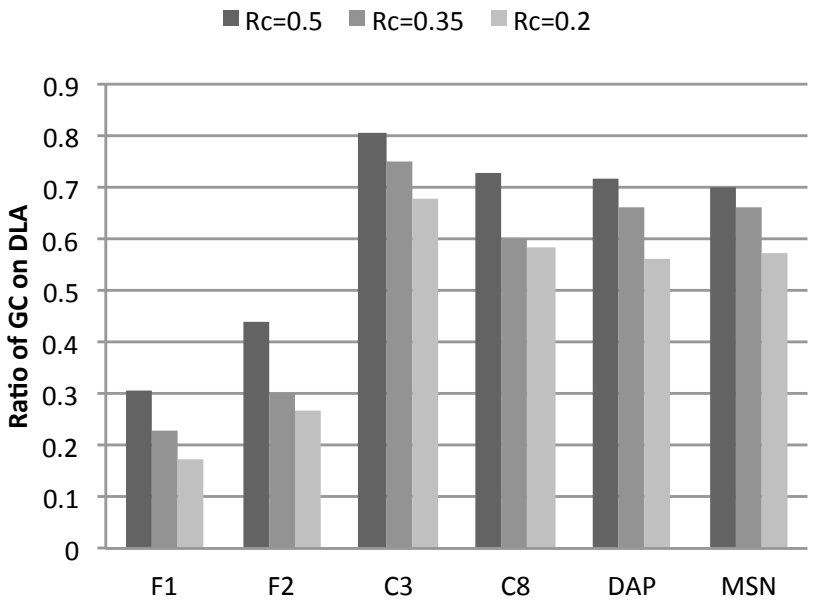


Figure 4.11: Ratio of GC executed in DLA.

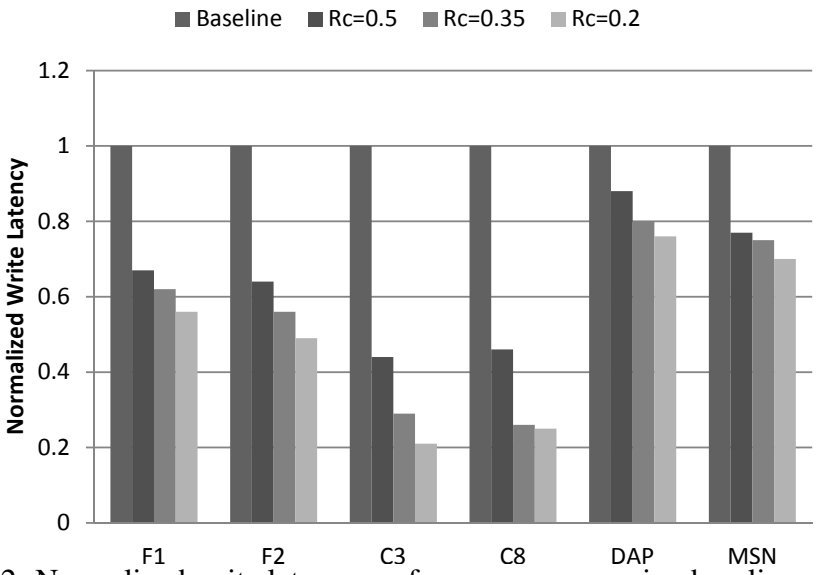


Figure 4.12: Normalized write latency performance: comparing baseline and ΔFTL.

potentially hinder the foreground requests to be serviced. We evaluate the average GC overhead of ΔFTL and compare the results to the baseline in Figure 4.13. We observe that ΔFTL does not significantly increase the GC overhead under most cases.

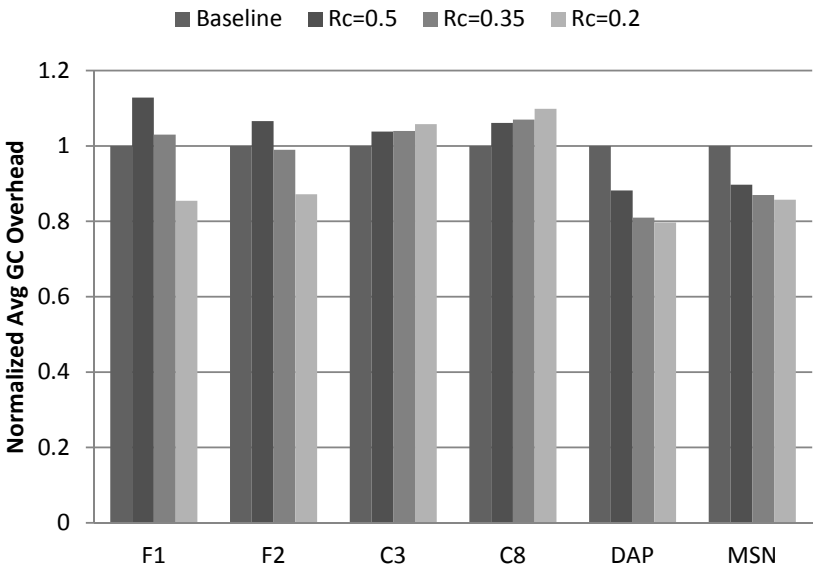


Figure 4.13: Normalized average GC overhead.

Overhead on Read Performance

ΔFTL reduces the write latency significantly and therefore alleviates the chip contention between the read and write requests, resulting less queuing delay for the reads. Under intensive workloads, the effective read latency (considering queuing delay on the device side) is reduced in Delta-FTL. However, ΔFTL inevitably introduces overhead on the raw read latency (despite queuing delay) when the target page is delta-encoded. Fulfilling such a read request requires two flash read operations. To overcome this potential overhead, ΔFTL delta-encodes only the write-hot and read-cold data and merges DLA pages to their original form if they are found read-hot. To evaluate the effectiveness of our approach, we collect the raw read latency values reported by the simulator and demonstrate the results in Figure 4.14. Compared to the baseline (normalized to 1), ΔFTL's impact on the read

performance is trivial: the read latency is increased by 5.3%, 5.4%, and 5.6% on average* when $R_c = 0.50, 0.35, 0.20$, respectively. The maximum (F2, $R_c = 0.50$) is 10.7%. To

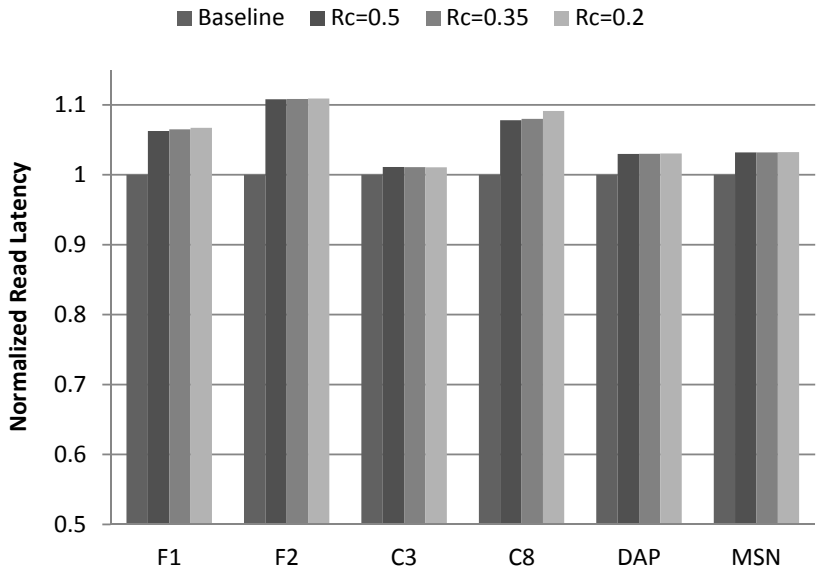


Figure 4.14: Normalized read latency performance: comparing baseline and ΔFTL.

summarize, our experimental results verify that ΔFTL significantly reduces the GC count and thus extends SSDs' lifetime at a cost of trivial overhead on read performance.

4.6 Summary

The limited lifetime impedes NAND flash-based SSDs from wide deployment in reliability-sensitive environments. In this chapter, we have proposed a solution, ΔFTL, to alleviate this problem. ΔFTL extends SSDs' lifetime by reducing the number of program/erase operations for servicing the disk I/O requests. By leveraging the content locality existing between the new data and its old version, ΔFTL stores the new data in the flash in the form of compressed delta. We have presented the design of ΔFTL in detail including the data structures, algorithms, and overhead control approaches in this chapter. ΔFTL is

*x% read latency overhead implies that x% of the requested pages are delta-encoded, which would double the raw latency compared to non-delta-encoded pages.

prototyped and evaluated via simulation. Our trace-driven experiments demonstrate that ΔFTL significantly extends SSD's lifetime by reducing the number of garbage collection operations at a cost of trivial overhead on read latency performance. Specifically, ΔFTL results in 33% to 58% of the baseline garbage collection operations, while the read latency is only increased by approximately 5%.

Chapter 5

Conclusions

In this dissertation, we make the following contributions to improve the performance and reliability of NAND flash-based SSDs:

- In the research work presented in Chapter 2 (**DiffECC**): 1) At the level of raw flash page organization, we propose to use finer-grained page segmentation along with shorter and weaker ECCs when NAND flash pages are programmed in a slower-than-normal speed. For the baseline mode (no segmentation, longer and stronger ECC, and normal program speed), we have to fetch and decode the entire page even if only a few sectors are requested. Compared to the baseline, our approach can reduce the read latency by fetching/decoding only the requested sectors due to segmentation, which can result in less bus transfer time and ECC decoding time. In addition, if the entire page is requested, our approach may also parallelize the fetching and decoding of each segment in the page. 2) At the level of disk access scheduling, we propose to buffer the writes and utilize the bandwidth of the idle time to opportunistically slow down the writes. 3) Based on simulations of real-world disk traces and using

2 bits/cell NAND flash memory, we demonstrate that this proposed cross-layer co-design can reduce the SSD read latency by up to 59.4% without compromising the write throughput.

- In the research work presented in Chapter 3 (**NAND flash Program/Erase Suspension**): 1) We analyze the impact of the long P/E latency on read performance, showing that even with the read prioritization scheduling, the read latency is still severely compromised. 2) By exploiting the internal mechanism of the P/E algorithms in NAND flash memory, we propose a low-overhead P/E suspension scheme which suspends the on-going P/E operations for servicing the pending read requests. In particular, two strategies for suspending the program operation, *Inter Phase Suspension (IPS)* and *Intra Phase Cancelation(IPC)* are proposed. In addition, we render the second priority to writes, which may preempt the erase operations. 3) Based on simulation experiments under various workloads, we demonstrate that the proposed design can significantly reduce the SSD read and write latency for both SLC and MLC NAND flash.

- In the research work presented in Chapter 4 (**Delta-FTL**): 1) We propose a novel FTL scheme, ΔFTL to extend SSD lifetime via exploiting the content locality. We describe how ΔFTL functionality can be achieved from the data structures and algorithms that enhance the regular page-mapping FTL. 2) We propose techniques to alleviate the potential performance overheads of ΔFTL. 3) We model ΔFTL's performance on extending SSD's lifetime via analytical discussions and outline the workload characteristics favored by ΔFTL. 4) We evaluate the performance of ΔFTL under real-world workloads via simulation experiments. Results show that ΔFTL

significantly extends SSD's lifetime by reducing the number of garbage collection operations at a cost of trivial overhead on read latency performance. Specifically, ΔFTL results in 33% to 58% of the baseline garbage collection operations; and the read latency is only increased by approximately 5%.

A List Of Publications

This dissertation is mainly based on the following papers.

I **Guanying Wu**, Xubin He, Ningde Xie, and Tong Zhang. Exploiting Workload Dynamics to Improve SSD Read Latency via Differentiated Error Correction Codes. *ACM Transactions on Design Automation of Electronic Systems*. 2013. Accepted.

II **Guanying Wu**, Xubin He, and Ben Eckart. An Adaptive Write Buffer Management Scheme for Flash-Based SSDs. *ACM Transactions on Storage, Vol. 8, No. 1, February, 2012*.

III **Guanying Wu** and Xubin He. Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality. In: *Proceedings of the European Conference on Computer Systems (Eurosys'2012, acceptance rate: 27/178=15%)*, April 10-13, 2012 Bern, Switzerland.

IV **Guanying Wu** and Xubin He. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'2012, acceptance rate: 26/137=19%)*, February 14-17, 2012, San Jose, USA.

V **Guanying Wu**, Xubin He, Ningde Xie, and Tong Zhang. DiffECC: Improving SSD Read Performance Using Differentiated ECC Schemes. In: *Proceedings of The 18th*

Annual Meeting of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'2010), **Best Paper Award Candidate**. August 17-19, 2010, Miami, USA.

VI **Guanying Wu**, Ben Eckart, and Xubin He. BPAC: An adaptive write buffer management scheme for flash-based Solid State Drives. In: *Proceedings of The IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'2010)*, May 6-7, 2010, Reno, USA.

Bibliography

[1] ARM Cortex R4. www.arm.com/files/pdf/Cortex-R4-white-paper.pdf. 68

[2] R. Agarwal and M. Marrow. A closed-form expression for write amplification in nand flash. In *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pages 1846–1850, dec. 2010.

[3] Nitin Agrawal, Vijayan Prabhakaran, and et al. Design Tradeoffs for SSD Performance. In *USENIX ATC*, Boston, Massachusetts, USA, 2008. viii, 3, 10, 21, 31, 39, 40, 61, 65, 71, 80

[4] D.G. Andersen and S. Swanson. Rethinking flash in the data center. *IEEE Micro*, 30(4):52–54, 2010. 61

[5] ARM®. Arm7. http://www.arm.com/products/processors/classic/arm7/index.php. 68

[6] JEDEC Solid State Technology Association. Stress-test-driven qualification of integrated circuits, jesd47g.01., 2010. http://www.jedec.org/.

[7] L. A. Barroso. Warehouse-scale Computing. In *Keynote in the SIGMOD?0 conference*, 2010. 61

[8] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to Flash memory. *Proceedings of the IEEE*, 91:489–502, April 2003. 3, 16

[9] R.E. Blahut. *Algebraic codes for data transmission*. Cambridge University Press, 2003. 17

[10] J.E. Brewer and M. Gill. Nonvolatile Memory Technologies with Emphasis on Flash. *IEEE Whiley-Interscience, Berlin*, 2007. ix, 3, 4, 43, 44

[11] Y. Cai, E.F. Haratsch, O. Mutlu, and K. Mai. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 521–526. IEEE, 2012.

[12] Y. Cai, G. Yalcin, O. Mutlu, E.F. Haratsch, A. Cristal, O.S. Unsal, and K. Mai. Flash Correct-and-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime. In *Proceedings of ICCD*, 2012.

[13] L.P. Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1126–1130. ACM, 2007.

[14] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. Endurance enhancement of flash-memory storage systems: An efficient static wear leveling design. In *DAC*, San Diego, CA, USA, June 2007. 9, 61

[15] F. Chen, T. Luo, and X. Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *FAST'2011*, 2011. 8, 62, 75, 81

[16] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX ATC*, pages 1–23, 2003. 62, 63, 81

[17] N. Duann. Error Correcting Techniques for Future NAND Flash Memory in SSD Applications. In *Flash Memory Summit*, 2009. 16

[18] K.-T. Park et al. A Zeroing Cell-to-Cell Interference Page Architecture With Temporary LSB Storing and Parallel MSB Program Scheme for MLC NAND Flash Memories. *IEEE Journal of Solid-State Circuits*, 43:919–928, April 2008. 13

[19] Google. Snappy. http://code.google.com/p/snappy/. 68

[20] S. Gregori, A. Cabrini, O. Khouri, and G. Torelli. On-chip error correcting techniques for new-generation flash memories. *Proceedings of the IEEE*, 91(4):602–616, 2003. 16

[21] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–33. ACM, 2009. 10, 61

[22] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *FAST'2011*, 2011. 8, 62

[23] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09*, pages 229–240, 2009. 8, 62, 64, 73, 75

[24] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010. 62

[25] Red Hat. The journalling flash file system, version 2, 2010. 8

[26] Steven R. Hetzler. System Impacts of Storage Trends Hard Errors and Testability. *;login:*, 36(3), 2011.

[27] HP Lab. Cello99 Traces, 2008. http://tesla.hpl.hp.com/opensource/. 22, 81

[28] X.Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR09*, page 10. ACM, 2009. 76

[29] Heeseung Jo, JeongUk Kang, SeonYeong Park, JinSoo Kim, and Joonwon Lee. FAB: flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006. 8, 62, 66

[30] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. DFS: a file system for virtualized flash storage. In *FAST'10*. USENIX, Feb 2010. 8

[31] D. Jung, Y.H. Chae, H. Jo, J.S. Kim, and J. Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 160–164. ACM, 2007.

[32] J. U. Kang, H. Jo, J. S. Kim, and J. Lee. A superblock-based flash translation layer for nand flash memory. In *International Conference on Embedded Software*, 2006. 6, 8, 62

[33] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices. *IEEE Transactions on Computers*, 58(6):744–758, 2009. 62, 66

[34] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 27(3):38–46, March 1994. 67, 73

[35] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of Storage Workload Traces from Production Windows Servers. In *IISWC'08*, 2008. 20, 22, 40, 79, 81

[36] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the USENIX 1995 Technical Conference*, pages 13–13. USENIX Association, 1995. 74

[37] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage Abstract. In *Proceedings of FAST*, 2008. 8, 62, 66

[38] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for Compact Flash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002. 6, 8, 62

[39] M.H. Kryder and C.S. Kim. After Hard DrivesłWhat Comes Next? *IEEE Transactions on Magnetics*, 45(10), 2009. 11

[40] S. Lee, T. Kim, K. Kim, and J. Kim. Lifetime Management of Flash-Based SSDs Using Recovery-Aware Dynamic Throttling. In *FAST'2012*. USENIX, 2012. 9

[41] Sang-Won Lee, Won-Kyoung Choi, and Dong-Joo Park. FAST: An FTL Scheme with Fully Associative Sector Translations. In *UKC 2005*, August 2005. 6, 8, 62

[42] Sungjin Lee, Keonsoo Ha, Kangwon Zhang, Jihong Kim, and Junghwan Kim. FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. In *USENIX ATC*. USENIX, June 2009. 9

[43] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS*, 42(6):36–42, 2008. 8

[44] Yong-Goo Lee, Dawoon Jung, Dongwon Kang, and Jin-Soo Kim. uftl: a memory-efficient flash translation layer supporting multiple mapping granularities. In *EMSOFT '08*, pages 21–30, New York, NY, USA, 2008. ACM. 8

[45] Marc Lehmann. Lzf. http://oldhome.schmorp.de/marc/liblzf.html. 68

[46] S. Lin and D. J. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice Hall, 1983. 16, 17

[47] R.S. Liu, C.L. Yang, and W. Wu. Optimizing NAND Flash-Based SSDs via Retention Relaxation. In *FAST'2012*. USENIX, 2012. 9

[48] R.S. Liu, C.L. Yang, and W. Wu. Optimizing NAND Flash-Based SSDs via Retention Relaxation. In *FAST'2012*. USENIX, 2012.

[49] SimpleScalar LLC. Simplescalar/arm. http://www.simplescalar.com/v4test.html. 68

[50] JP MacDonald. xdelta. http://xdelta.org. 63, 68

[51] U. Manber and S. Wu. Glimpse: A tool to search through entire file systems. In *Usenix Winter 1994 Technical Conference*, pages 23–32, 1994. 63

[52] Charles Manning. Yet another flash file system, 2010. 8

[53] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L.R. Nevill. Bit error rate in nand flash memories. In *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*, pages 9–19. IEEE, 2008.

[54] C.B. Morrey III and D. Grunwald. Peabody: The time travelling disk. In *Proceedings of MSST 2003*, pages 241–253. IEEE. 62, 63, 81

[55] MF Oberhumer. Lzo. http://www.oberhumer.com/opensource/lzo. 68

[56] ONFI Working Group. The Open NAND Flash Interface, 2010. http://onfi.org/. 5, 67

[57] Alina Oprea and Ari Juels. A clean-slate look at disk scrubbing. In *FAST'10: the 8th USENIX Conference on File and Storage Technologies*. USENIX, Feb 2010. 22

[58] Yangyang Pan, Guiqiang Dong, and Tong Zhang. Exploiting memory device wear-out dynamics to improve NAND flash memory system performance. In *FAST'2011*. USENIX, 2011. 9

[59] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of CASES'2006*, pages 234–241, 2006. 8

[60] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis. Block management in solid-state devices. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 21–21, Berkeley, CA, USA, 2009. USENIX Association. ix, 5, 6

[61] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992. 6

[62] Samsung, 2010. http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products-OneNAND.html. 5

[63] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *EuroSys'2012*. ACM, 2012. 9

[64] J. Seward. The bzip2 and libbzip2 official home page. 2002. http://sources.redhat.com/bzip2. 68

[65] SiliconSystems. Increasing flash solid state disk reliability. *Technical report*, 2005. 9, 61

[66] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD Lifetimes with Disk-Based Write Caches. In *FAST'10*. USENIX, Feb 2010. 9, 75

[67] Storage Performance Council. SPC trace file format specification, 2010. http://traces.cs.umass.edu/index.php/Storage/Storage. 22, 40, 79, 80

[68] F. Sun, K. Rose, and T. Zhang. On the use of strong bch codes for improving multilevel nand flash memory storage capacity. In *IEEE Workshop on Signal Processing Systems (SiPS)*, 2006. 16, 18, 27

[69] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. A Hybrid Solid-State Storage Architecture for the Performance, Energy Consumption, and Lifetime Improvement. In *HPCA*, pages 141–153. IEEE, Jan 2010. 9

[70] K. Takeuchi, T. Tanaka, and H. Nakamura. A double-level-V_{th} select gate array architecture for multilevel NAND flash memories. *IEEE Journal of Solid-State Circuits*, 31:602–609, April 1996. 14

[71] Toshiba, 2010. http://www.toshiba.com/taec/news/press-releases/2006/memy-06-337.jsp. 5

[72] wikipedia. Battery or super cap, 2010. http://en.wikipedia.org/wiki/Solid-state-drive#Battery_or_SuperCap. 73

[73] Chin-Hsien Wu and Tei-Wei Kuo. An adaptive two-level management for the flash translation layer in embedded systems. In *ICCAD '06*, pages 601–606, New York, NY, USA, 2006. ACM. 8

[74] Guanying Wu and Xubin He. Delta-FTL: improving SSD lifetime via exploiting content locality. In *EuroSys'2012*. ACM, 2012. 8

[75] Guanying Wu and Xubin He. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of FAST'2012*, 2012. 5

[76] Guanying Wu, Xubin He, and Ben Eckart. An Adaptive Write Buffer Management Scheme for Flash-Based SSDs. *ACM Transactions on Storage*, 8(1):1–24, 2012. 8, 62

[77] Guanying Wu, Xubin He, Ningde Xie, and Tong Zhang. DiffECC: Improving SSD Read Performance Using Differentiated Error Correction Coding Schemes. *MASCOTS*, pages 57–66, 2010. 3, 49

[78] Guanying Wu, Xubin He, Ningde Xie, and Tong Zhang. DiffECC: Improving SSD Read Performance Using Differentiated Error Correction Coding Schemes. In *MASCOTS*, pages 57–66, 2010. 5

[79] Q. Yang and J. Ren. I-CASH: Intelligently Coupled Array of SSD and HDD. In *Proceedings of HPCA'2011*, pages 278–289. IEEE, 2011. 9, 62, 63, 75, 79

[80] Q. Yang, W. Xiao, and J. Ren. TRAP-Array: A disk array architecture providing timely recovery to any point-in-time. *ACM SIGARCH Computer Architecture News*, 34(2):289–301, 2006. 62, 63, 79, 81

Vita

Guanying Wu was born on Nov. 22nd, 1985, in Feicheng, Shandong, China, and is a Chinese citizen. He graduated from Jinan Foreign Language School, Jinan, Shandong in 2003. He received his B.S. in Electrical Engineering from Zhejiang University, Hangzhou, China in 2007. He received his M.S. degree in Computer Engineering from Tennessee Technological University, Cookeville TN, USA in 2009.