공학박사학위논문

낸드 플래시 기반 저장장치의 성능 및 신뢰성 향상을 위한 계층 교차 최적화 기법

Cross-Layer Optimization Techniques for Improving Performance and Reliability of NAND Flash-Based Storage Systems

2015년 8월

서울대학교 대학원
전기·컴퓨터 공학부
하건수

낸드 플래시 기반 저장장치의 성능 및 신뢰성 향상을 위한 계층 교차 최적화 기법

Cross-Layer Optimization Techniques for Improving Performance and Reliability of NAND Flash-Based Storage Systems

지도교수 김 지 홍

이 논문을 공학박사 학위논문으로 제출함
2015년 3월

서울대학교 대학원
전기·컴퓨터 공학부
하건수

하건수의 공학박사 학위논문을 인준함
2015년 7월

위 원 장 _____ (인)

부위원장 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

위 원 _____ (인)

Abstract

As the cost-per-bit of NAND flash memory is quickly improved by advanced process technologies and multi-leveling techniques, NAND flash-based storage systems are widely employed from mobile embedded systems to high-end enterprise server systems. Although the advanced process and device techniques have greatly improved the cost-per-bit of NAND flash memory, they have also significantly degraded the performance and reliability of NAND flash memory as key side effects of the advanced techniques. In order for NAND flash-based storage systems to be more broadly used in various computing environments, it is critical to overcome the performance and reliability problems of recent high-density NAND flash memory in a satisfactory fashion. In this dissertation, we argue that cross-layer optimization techniques, which vertically integrate various optimization factors from different design abstraction levels, can play key roles in improving performance and reliability of high-density NAND flash memory.

First, we propose read-disturb management techniques which reduce the expensive read-disturb management overheads while maintaining reliability of NAND flash memory. An FTL using the read-disturb management module, called `redFTL`, alleviates highly skewed read accesses to a small part of NAND flash memory into more balanced read accesses to a large number of blocks, thus reducing data migrations needed for avoiding read-disturb errors. As an extended version of `redFTL`, we propose an integrated read-disturb management technique, called `redFTL+`, which fundamen-

tally solves read-disturb problems by exploiting a tradeoff between the read disturbance and write speed. By modifying NAND chips to support multiple read modes with different read voltages and write speeds, redFTL+ intelligently allocates frequently-read data to read-resistant blocks. Since the read disturbance is also proportional to the read time, redFTL+ takes advantage of the difference in the read time among different NAND pages by reallocating read-intensive data to read-resistant pages.

Second, we propose data separation techniques which reduce garbage collection overhead. We propose a program context-aware data separation technique, called PDS, which can reduce the garbage collection overhead by exploiting program context hints. By using a program context, which serves as a proper granularity of maintaining data update behavior, PDS helps an FTL gather data with similar update times to the same blocks. As an improved version of PDS, we propose an integrated data separation technique, called IDS, which uses both update history of NAND device and program context hints for predicting data update behaviors. By classifying data based on the cross-layer information, an FTL using IDS can make more dead or near-dead blocks over PDS, thus reducing the garbage collection overhead.

In order to evaluate the effectiveness of the proposed techniques, we performed a series of evaluations using both a simulator and an emulator with I/O traces which were collected from various systems. Our experimental results show that cross-layer optimization techniques are more effective over our single-layer optimization techniques. RedFTL+ decreases the read-disturb management overhead on average by 24% over redFTL. The IDS-based FTL decreases the garbage collection overhead on aver-

age by 18% over the PDS-based FTL. The evaluation results demonstrate that our cross-layer optimization techniques improve an overall performance of NAND-based storage systems over previous single-layered optimization techniques by reducing overheads from read-disturb management and garbage collection while maintaining the reliability of the storage systems.

Keywords: NAND Flash Memory, Flash-Based Storage Devices, Storage Performance Optimization, Operating System, Embedded System, Storage Reliability Management

Student Number: 2005-21527

Contents

List of Figures

List of Tables

Chapter 1

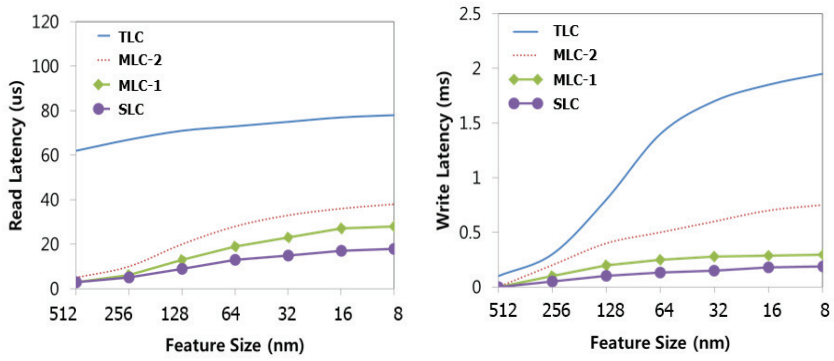
Introduction

1.1 Motivations

Recently, NAND flash memory is widely used as a storage device from embedded systems to high-end enterprise servers. Because of its many attractive characteristics for mobile storage devices such as light weight, low power consumption, durability, and high performance, it has been widely used for mobile embedded systems. In addition to the advantages, as the cost per byte is falling while the storage capacity is increased, large-capacity NAND flash memory devices such as solid state drives (SSDs) are more commonly employed for high-end desktops and enterprise storage servers.

However, as NAND flash memory technology scales down to 20-nm and below, data reliability becomes a major design concern for NAND flash-based storage systems. In particular, as storage systems employ more recent high-density NAND flash chips (such as triple-level cell (TLC) NAND devices), more system resources are dedicated to maintain data integrity because these NAND chips exhibit poor reliability characteristics. For example, 20-nm MLC NAND flash requires 40-bit error correcting codes (ECCs) per 1 KB data, while 30-nm MLC NAND requires 24-bit ECCs [2].

As the density of NAND flash memory is increased, performance of NAND flash memory is also getting worse as well as the data reliability.



(a) A read latency

(b) A write latency

Figure 1: Trends of read and write speeds under various NAND flash chips [1].

Figure 1 shows a tendency of read and write speeds of NAND block under various feature sizes. In Figures 1(a) and 1(b), the x-axis denotes a feature size of NAND chip, and the y-axis represents the latencies of read and write operations, respectively. As shown in Figures 1(a) and 1(b), both read and write operations are slower as the density of NAND flash memory is increased. In the future, as NAND flash memory is scaled down and the number of bits per cell is increased, the data reliability and performance degradation problems will be even more critical to NAND-based storage systems.

1.1.1 Read-Disturb Problem

Among various reliability issues such as limited endurance and short data retention, the read-disturb problem is expected to emerge as a major reliability concern for future high-density NAND flash memory. When a

page P in a block B is read, a read disturb occurs to P's neighboring pages in the block B. Since cells are serially connected in a string structure in NAND flash memory, the cells in the same string can be unintentionally programmed by a certain level of read voltage (e.g., 7V [3]) applied whenever their neighboring cells in the same block are read. If neighboring cells are repeatedly disturbed by many reads to the same block, some of the affected cells may lose their data from frequent unintentional programming. If the number of changed bits by read disturbs exceeds the number of recoverable bits by a recovery method such as an ECC, a read-disturb error occurs.

In order to avoid data corruption by a read-disturb error, an anticipatory prevention procedure, called read reclaim (RR), is required. Since a disturbed block B can return to its initial undisturbed status when it is erased, the block B is erased during RR when RR decides that the current level of read disturbance of the block B is sufficiently high. If valid data exist in the erased block B, they must be moved to other healthier blocks before the block B is erased. One straight forward RR technique maintains the number of performed read operations per block to predict the read-disturbance status of a block [4]. When a block undergoes more read operations than a preset upper bound on the number of read operations allowed per block, called RR threshold, RR is triggered. The RR procedure causes several read, write, and erasure operations while a flash translation layer (FTL) moves valid data pages in the partially read-disturbed blocks and erases the blocks. Frequent RR operations negatively affect the performance of NAND flash memory because of extra data migrations and block erasures during RR invocations.

As the maximum read count is quickly decreased in a high-density NAND flash memory, several device-level approaches for alleviating the side effect of read disturbs have been recently proposed. However, these device-level techniques are not sufficient to tackle a rapid decrease in the maximum read count because device-level techniques must consider several other reliability issues (such as data retention and read disturbs) as well as the device cost at the same time. Moreover, to the best of my knowledge, there is no software-based read-disturb management technique in literatures. In order to prevent reliability of NAND flash memory from being degraded due to frequent read reclaims, an efficient read-disturb management technique is required according to changing properties of high-density NAND flash memory.

1.1.2 Garbage Collection Problem

Because of the performance degradation of high-density NAND flash memory, the overhead of garbage collection (GC) is increased. In NAND flash-based storage systems, garbage collection is required when there are not enough free blocks to write new data because NAND flash memory does not allow an in-place update operation. If data are updated in NAND flash memory, an FTL stores the newly requested data in another page. Since the previously written data remain as invalidated data in the NAND flash memory, a garbage collection process is triggered by the FTL in order to reclaim the blocks with the invalidated data so that new data can be stored into NAND flash memory. A garbage collection procedure involves several read, write, and erase operations. Since each operation of NAND flash memory is

atomic, the FTL cannot process the next request from a file system while an operation is processed during a garbage collection process. As shown in Figures 1(a) and 1(b), because of the performance degradation of read and write operations in high-density NAND flash memory, the response time of each I/O request can be elevated when the request is conflicted with a garbage collection process, thus decreasing file system performance. In other words, the whole system is likely to be delayed for a longer time compared to low-density NAND flash memory due to the slow extra copies and erases during garbage collection processes. Since such problem can be aggravated in the future high-density NAND flash memory, an efficient garbage collection algorithm is becoming more and more important.

In order to minimize the garbage collection overhead, many techniques have been proposed [5]. Regardless of garbage collection algorithms used, moving valid data from selected victim blocks to new blocks during garbage collection takes a significant portion in the total execution time of a garbage collection algorithm. Therefore, reducing the total number of copied data from the victim blocks is a key factor in improving the performance of a garbage collection algorithm. To reduce the amount of copied data from the victim blocks, a common approach is to separate data based on their characteristics so that the number of dead blocks (which have no valid data) or near-dead blocks (which have few valid data) can be increased. The more dead or near-dead blocks are generated, the more likely that they can be selected as victim blocks during garbage collection, thus reducing the garbage collection overhead.

One of the most widely used data separation heuristics is to classify

data based on their update frequency. This data separation technique classifies data based on their write temporal locality, and it treats data with different temporal locality in a different way [15, 10]. The assumption of this technique is that data with high write temporal locality are likely to be updated soon by successive update requests, and hence the number of dead blocks increases if data with high locality are clustered in the same block. The simplest version of this locality-based data separator divides data into two groups, hot data and cold data according to the number of updates in a given time period. By storing hot data in hot blocks, they are more likely to be dead blocks.

From our comparative study using an off-line optimal FTL, we have observed that gathering data with similar future update times to the same blocks, not data with high update frequencies, is a more important factor in minimizing garbage collection overhead. We have found that data with similar update frequencies were not necessarily updated at similar times. For example, there is no clear correlation on their update times among hot data if they were classified as hot data at different times. If several hot data groups with different locality are stored in the same block, the probability that all data in that block are updated together is small because data with different locality have different update times. One of the main reasons of the poor performance of existing garbage collection heuristics can be attributed to the fact that they ignore data update times in devising their data separation techniques. Considering the fact that the existing approach has potential to perform an efficient garbage collection process, a new novel data separation technique which considers update times of data in classifying data is

required.

1.2 Research Goals and Contributions

This dissertation focuses on developing vertically-integrated cross-layer optimization techniques for overcoming the reliability and performance problems of high-density NAND flash memory. In particular, we propose integrated cross-layer optimization techniques which reduce the overheads from read-disturb management and garbage collection while maintaining reliability. The contributions of this dissertation can be summarized as follows:

- We propose a novel read-disturb management technique which reduces the occurrence of read reclaims. This technique detects read-hot pages in a partially disturbed block and *proactively* moves them to other healthier blocks before read reclaim is activated. By distributing read requests, RR occurrences are reduced. Moreover, by avoiding simultaneous data migrations, this technique better balances I/O response times under read reclaim activations. Based on this technique, we have designed a new read disturb-aware flash translation layer (`RedFTL`) for high-density NAND flash memory.

- As an improved version of `redFTL`, we propose a novel integrated approach for managing the read-disturb problem in high-density NAND flash memory. In this dissertation, the advanced `redFTL` is called as `redFTL+`. Based on read voltage scaling and read time scaling (motivated from the NAND device physics), we introduce read-resistant blocks at the device level, and develop a concept of read-resistant

pages. By intelligently exploiting read-resistant blocks and read-resistant pages at the FTL level, frequent read reclaims are dramatically reduced.

- We propose a novel data separation technique which predicts data update times by exploiting program behavior as hints. This technique estimates what data will be updated together based on the data update history of the program context which indirectly identify data update locality. Once data with similar future update times are predicted, the data are allocated into the same block by an FTL using the proposed technique, thus increasing the efficiency of garbage collection processes. (In this dissertation, this program context-based data separation technique is denoted as `PDS`.)

- We devise a further improved version of `PDS`, called integrated data separation technique (`IDS`), which exploits both block access history and program context hints in order to predict more accurately data update behaviors. `IDS` periodically evaluates the update locality of data, and program contexts of the data are clustered into the same update group if their update locality is high. An FTL using the `IDS` can gather data from member program contexts of an update group into the same block, thus reducing the garbage collection overheads. Based on block access history, `IDS` detects more frequently updated data from an update group, and it stores separately the data in different blocks. Since the gathered data are likely to be updated over other cold data, by adding the block access history in classifying data, an FTL

using IDS can further reduce the garbage collection overheads.

1.3 Dissertation Structure

This dissertation consists of eight chapters. The first chapter is the introduction of the dissertation, while the last chapter serves as conclusions with a summary and future work. The six intermediate chapters are organized as follows:

Chapter 2 provides the background for read-disturb problems and existing data separation techniques as background of this dissertation. We explain the read-disturbance problem in high-density NAND flash memory for explaining our proposed read-disturb management techniques. Moreover, we explain how existing data separation techniques classify data for efficient garbage collection.

Chapter 3 describes the overall algorithm of the redFTL. We explain the unnecessary read reclaims by skewed read requests for a small number of blocks, and then describe how redFTL detects frequently read data and reduce the read disturb management overheads by distributing the frequently read data to whole healthier blocks.

Chapter 4 presents a read-disturb aware FTL, called redFTL+. An integrated cross-layer integrated approach including two read-disturb management techniques is explained. After two key tradeoffs related to read disturbance in a NAND flash block are introduced, we explain how redFTL+ exploits the tradeoffs for reducing the read disturb management overheads.

In Chapter 5, we describe a program context-aware data separation

technique, called PDS, which estimates data update time based on program contexts, so as to reduce the garbage collection overhead. We explain the relationship between program contexts and update behavior, and present how PDS classify data based on program contexts.

Chapter 6 introduces an integrated program context-aware data separation technique, called IDS. We explain the limitations of single-layer data separation technique, and describe how IDS further reduces the garbage collection overheads by exploiting cross-layer information.

Chapter 2

Background

In this section, the overall architecture of NAND flash memory is described. In particular, its physical limitations including the read-disturb problem and the erase-before-write restriction are also described. I then explain an integrated cross-layer architecture which is designed to overcome these physical limitations, thus enhancing NAND flash memory to be widely used as storage systems.

2.1 NAND Flash Memory

Figure 2 illustrates overall organization of NAND flash memory. NAND flash memory has multiple blocks, and each block is consists of several pages. In many NAND flash memories, the size of a page is between 2 KB and 8 KB, and a block includes between 128 and 192 pages. Each page also has a spare area of which size is between 16 and 256 bytes. This area is used for storing software metadata and ECC data. A page is a unit of read and write operations, while a block is erased by an erasure operation. NAND flash memory does not support an overwrite operation. Therefore, if data in a page P are updated, the newly requested data must be written in another page, and the previously written data remain as invalid data in the page P. In order to reclaim the page P, an extra reclaim process, called garbage col-

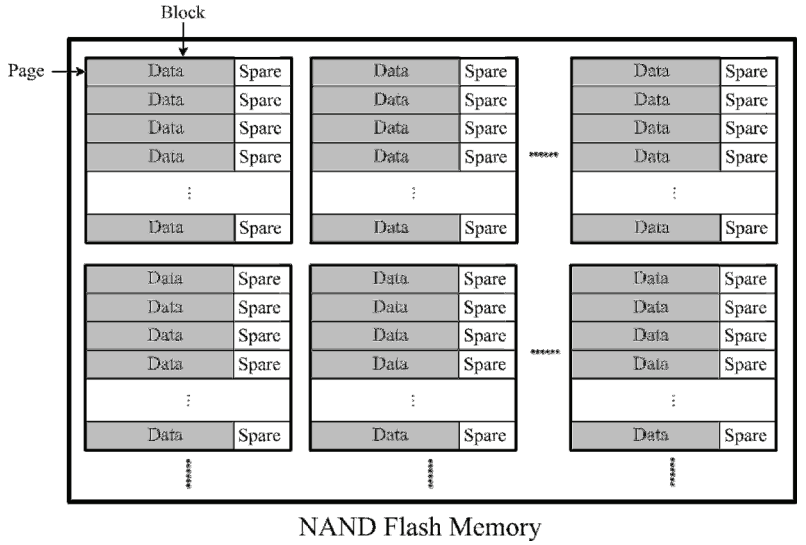
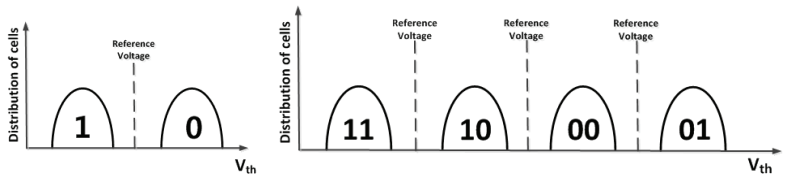


Figure 2: An overall organization of NAND flash memory.

lection, is required. Furthermore, the total number of program/erase (P/E) cycles allowed for each block is limited to between 3,000 and 100,000 cycles.

NAND flash memory stores data in a memory cell. Each memory cell consists of a transistor with a floating gate. By storing electrons in a floating gate, NAND flash memory can store data. The number of electrons stored in the floating gate determines the threshold voltage, denoted as V_{th}, and this threshold voltage represents the state of each cell. If each cell has only two states, the cell is called *single-level cell* (SLC) flash memory. Each cell in SLC can store one bit. Figure 3(a) shows how the value of a bit is decided. If the threshold voltage is greater than a reference voltage, it is regarded as 1. Otherwise, it is determined as 0. In general, the write operation moves the state of a cell from '1' to '0', while the erase operation changes '0' to '1'.



(a) SLC (b) MLC

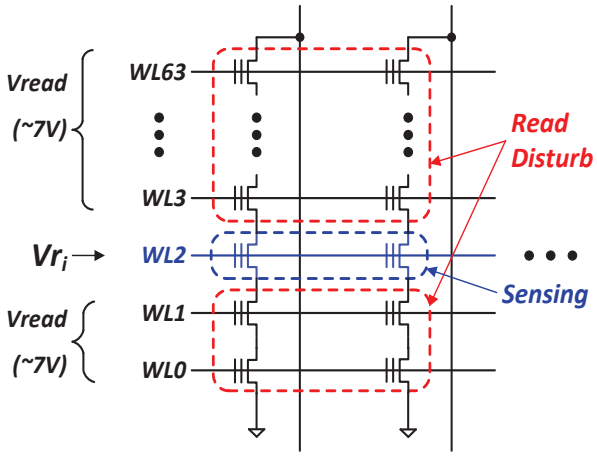
Figure 3: Threshold voltage distributions for SLC (1 bit/cell) and MLC (2 bits/cell).

If flash memory is composed of memory cells which have more than two states, it is called *multi-level cell* (MLC) NAND flash memory. Each cell in MLC stores more than two bits of data, as shown in Figure 3(b). Although the density of MLC NAND flash memory is increased over SLC NAND flash memory, it requires more precise charge placement and sensing due to the narrower voltage ranges between cell states. Therefore, MLC is slower than SLC for all operations, and endurance also worsens.

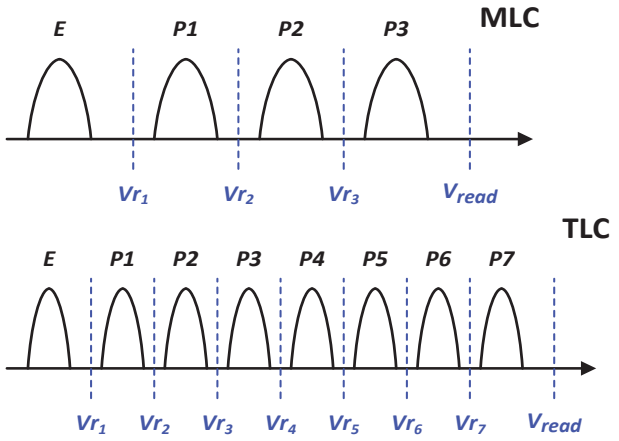
In recent high-density NAND flash memory, as the density of NAND flash memory increases using advanced process techniques such as shrinking processes (e.g., 20 nm and below process technology) and multi leveling (e.g., triple-level cell (TLC)), the read-disturb problem is expected to emerge as a major reliability concern for future high-density NAND flash memory [3]. When a page P in a block B is read in NAND flash memory, a read-disturb error may occur to P's neighboring pages in the block B. Since NAND cells are serially connected in a string structure, cells in the same string are unintentionally programmed when one of their neighboring cells is read. As the number of unintentional programs increases for the same-cell, the state of the cell may change. If the number of changed bits by read

disturbs exceeds the number of recoverable bits by an ECC, a read-disturb error occurs. In order to avoid data corruption by a read-disturb error, an anticipatory prevention procedure, called *read reclaim (RR)*, is required. Since a disturbed block B can return to its initial undisturbed status when it is erased, the block B is erased during RR when RR decides that the current level of read disturbance of the block B is sufficiently high. If valid data exist in the erased block B, they must be moved to other healthier blocks before the block B is erased.

In order to understand why read-disturb errors occur in NAND flash memory, we briefly explain how a NAND cell is read. As shown in Figure 4(a), NAND memory cells are serially connected to form a string unit which is a basic operational unit. During a read operation, a preset reference voltage V_{r_i} is applied to a selected word line (e.g., WL2 in Figure 4(a)) for determining the cell state of the selected word line while a read voltage V_{read} is applied to unselected word lines (e.g., WL0$\sim$WL1 and WL3$\sim$WL63 in Figure 4(a)) to fully turn on the unselected cells in the string (so that the selected cell state can be transferred to a sensing circuit without any interference). A read voltage should be high enough to make the unselected cells transparent during a read operation. Otherwise, the selected cell state cannot be precisely determined because the unselected cells will distort the current read from the selected cell. Figure 4(b) illustrates how reference voltages and read voltages are positioned at MLC and TLC NANDs, respectively.



(a) Voltage settings during a read operation in a NAND flash string.



(b) Reference voltages V_{r_i}'s and the read voltages V_{read}'s of MLC NANDs and TLC NANDs.

Figure 4: A relationship among the read reference voltages and read voltage affecting the read disturbance of NAND cells.

Since a high read voltage is applied to the unselected cells of the same string during a read operation, when a read voltage is high during a read operation, the unselected cells may be unintentionally and softly programmed,

resulting in the read-disturb error. Although the effect of a soft program per read operation is too small to affect cell data, if its effect is continuously accumulated by repetitive read operations, the stored contents in memory cells will be eventually altered [3, 6]. Since such an unwanted soft program is mainly caused by the FN-tunneling effect (which has an exponential dependence on the *read voltage* across a tunnel oxide [7]), the read-disturb problem gets more severe with a higher read voltage. Based on the widely-known FN-tunneling equation [7], the effects of a read voltage and read operation time on the read disturbance can be quantified by the total number N_e of electrons unintentionally injected into a memory cell during repetitive read operations as follows:

$$N_e \propto J_{FN} \times T_{stress} \propto \{{V_{read}}^2 \cdot exp[\frac{-1}{V_{read}}]\} \times \{T_{read} \cdot N_{read}\}, \qquad (2.1)$$

where V_{read} is an applied read voltage, T_{read} is a read operation time, and N_{read} is the total number of read operations. Since J_{FN} represents an FN-tunneling current for a unit time, in order to estimate the total number of N_e accumulated over the entire stress-time interval T_{stress}, the total read time (i.e., the product of T_{read} and N_{read}) is included in the Equation (2.1) [8].

Furthermore, the FN-tunneling effect is linearly increased with the length of a stress-time interval, the degree of the read disturbance is also a linear function of a read operation time. Since the read disturbance of NAND cells is dependent on the read voltage and read operation time, if the read voltage or the read operation time can be lowered or reduced, respectively, the read disturbance of NAND cells can be mitigated.

2.2 System Software for NAND Flash Memory

In order to overcome the physical limitations of NAND flash memory, such as the *erase-before-write* restriction and the limited P/E cycles, a special software layer, called a *flash translation layer* (FTL), is usually used in NAND flash memory-based storage systems [5]. The FTL emulates a normal block device on top of NAND flash memory, thus enabling users to use NAND flash memory as if they use block device such as hard disk drives. The FTL is charge of address mapping, garbage collection, and wear-leveling. The address mapping function maps a logical block address (LBA) from a host system to a physical block address (PBA) in NAND flash memory. When an update request occurs, the FTL newly allocates new free page to the request in NAND flash memory, allowing us to hide the erase-before-write restriction of NAND flash memory. This update process is called *out-place update*. The location information of the newly allocated page are maintained in the page mapping table which keeps track of mapping information between LBA and PBA. The old versions of newly written data remain invalid in the original location. In order to maintain free space in NAND flash memory, the FTL has to perform a garbage collection process which reclaims the invalid pages in NAND flash memory. Finally, the wear-leveling procedure induces all blocks in NAND flash memory to be evenly erased, thus preventing frequently erased blocks from being rapidly worn out than other blocks.

In addition to such essential functions, read reclaim is also necessary in an FTL for high-density NAND flash memory. One straightforward read

reclaim technique maintains the number of performed read operations per block to predict the read-disturbance status of a block [4]. When a block undergoes more read operations than a preset upper bound on the number of read operations allowed per block, RR is triggered. Since valid data in a partially disturbed-blocks must be moved to a healthier block, several read, write, and erasure operation are performed during an RR. Since the read-resistance of NAND flash memory block is getting worsen in high-density NAND flash memory, RR also can affect negatively on performance if it is frequently triggered.

2.3 NAND Flash-Based Storage Devices

Solid state disks (SSDs) consisting of NAND flash memory are being widely used various systems such as laptops, desktops, and enterprise servers. Although a NAND flash chip with an 8-bit serial bus provides limited bandwidth (e.g., 40 MB/s for reads and 13 MB/s for writes), SSD can service high performance I/O by using parallel I/O processing. Figure 5 illustrates a typical SSD architecture. The SSD consists of a microprocessor, internal memory, flash interface, several flash chips, and a host interface module. The microprocessor in the SSD performs an FTL and receives host commands (e.g., read, write, and erasure) through the host interface module from the host system, and then issues several flash I/O commands to the flash chips. Mapping information and metadata of the FTL are loaded to SRAM. The flash memory controller handles multiple I/O commands simultaneously. Therefore, the SSD can offer high performance than a single

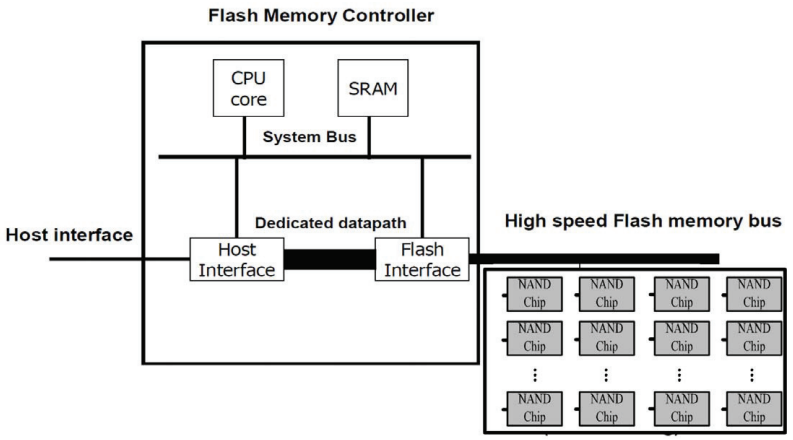


Figure 5: A simplified diagram of a typical SSD.

NAND flash chip.

2.4 Related Work

As explained in Chapter 1, there is no read-disturb management technique, while there have been lots of studies for reducing garbage collection overheads. Although the number of read requests allowed between consecutive block erasures is decreased, read reclaims in recent MLC NAND flash memory have been able to serve sufficient read requests. For instance, MLC flash memory could serve at least 600,000 read requests between consecutive block erasures in 2009 [2]. However, as the maximum read count is quickly decreased in a high-density NAND flash memory, several device-level approaches for alleviating the side effect of read disturbs have been recently proposed. On the other hands, there have been a lot of studies to reduce garbage collection overhead on a variety of system levels includ-

ing firmware, a flash controller, and device logics in a NAND flash chip. (Note that, among the various garbage collection optimization techniques, we mainly focus on data separation technique in this dissertation.) However, considering the rapidly decreasing performance of recent high-density NAND flash memory, those techniques are not enough to provide high performance. Therefore, more advanced techniques that can further improve storage performance are always required.

In this section, we briefly describe existing performance and reliability optimization techniques, the read-disturb management techniques [9, 3] and data separation techniques [10, 11, 12, 13, 14], which are highly related to the proposed techniques in this dissertation. The read-disturb management technique is proposed to enhance read-resistance by modifying NAND chip device. On the other hands, the data separation techniques improve garbage collection efficiency by providing write locality hints to an FTL.

2.4.1 Read-Disturb Techniques

As the maximum read count is quickly decreased in a high-density NAND flash memory, several device-level approaches for alleviating the side effect of read disturbs have been recently proposed. However, these device-level techniques are not sufficient to tackle a rapid decrease in the maximum read count because device-level techniques must consider several other reliability issues (such as data retention and read disturbs) as well as the device cost at the same time. Since there are inherent tradeoff relationships among manufacturing parameters that affect NAND reliability, a device-level optimization technique for improving the NAND read disturbs

is inevitably limited, making it difficult for the device-level technique to provide practical solutions for the read disturb issue. For example, the tunnel oxide thickness (T_{OX}) is one of the important manufacturing parameters which affect many reliability issues such as data retention and read disturbs. For the read-disturb problem, a thicker T_{OX} is desirable, however, in order to compensate for the rapid degradation of data retention caused by a shrink below 20 nm-node, a thinner T_{OX} is inevitably employed [9]. When read disturb is more important than data retention, scaling down T_{OX} is not a proper solution for improving the read-disturb problem.

Kang et al. proposed another device-level approach that mitigates the worst-case condition for the read-disturb problem based on the investigation of a NAND read operation in detail. By adding an additional dummy cell to an existing NAND cell string and controlling its gate voltage properly, the electric field across the tunnel oxide of worst-conditioned NAND cells is effectively reduced, thus mitigating the worst-conditioned read-disturb problem [3]. However, since their approach, which is effective for the worst-conditioned NAND cells only, requires many modifications of the existing NAND device architecture as well as its operations, it has not been widely adopted in commercial NAND devices.

2.4.2 Data Separation Techniques

Since data separation techniques are designed to reduce garbage collection overheads, they are often researched in the context of FTLs. Chang et al. [10] proposed a hot-cold identification technique based on two levels of LRU lists. The first-level keeps track of the logical block addresses

(LBAs) of data which are written more than twice within a short time, while the second-level list stores the LBAs of data which have been recently written once or evicted from the first-level list because the data no longer meet the criterion for inclusion. An LBA is deleted from each list if it does not be accessed for a long time. When an FTL tries to identify the hotness with the LBA of requested data, the requested data is determined as hot if the LBA is included in the first-level list.

Hsieh et al. [11, 12] introduced a hash-based data separation technique which counts write requests in a hash table of moderate size. The LBA corresponding to each request is hashed to an entry, so that each entry contains the sum of the number of requests corresponding to several LBAs. All the entries are periodically divided by two to prevent overflows. LBAs that hash to entries with high values are classified as hot data: this is a set a rather approximate technique. Recently, Park et al. [13] proposed a similar technique in which multiple Bloom filters replace the hash table. The size of each filter is increased in turn, while the entries are successively erased in a round-robin manner. Thus each Bloom filter covers a different time-period, achieving a fine-grained representation of recency.

Data separation techniques based on both frequency and recency are effective in reducing the overhead of garbage collection. However, these techniques have several problems. First, the hotness of data is not clearly be related to the number of accesses and the time that has elapsed the data were modified, because 'hotness' itself is a subjective and relative concept. For example, assume that data items A, B, and C are respectively accessed 10,000, 6,000, and 4,000 times for an hour. Clearly, A is more frequently

accessed than either B or C, and we may classify it as 'hot'. However, it is difficult to decide whether data time B is 'hot' either from the absolute number of accesses, or by comparing this with the data for B and C. The 'hotness' of B is likely to depend on some threshold values, which may be changed. This makes it difficult to evaluate the accuracy of data separation algorithms. Second, even if two data items are classified as having the same hotness, they may well be updated in different time if their hotness was determined at different times.

There are also other approaches to data separation. For example, Chang et al. [15] suggested a size-based prediction technique, in which locality is associated with the amount of data requested. This technique is based on the observation that requests for a small amount of data tend to occur more frequently. Thus the data associated with these small transactions are classified as hot. However, this technique may be compromised if transactions of many sizes are incurred by page cache flushing. Another way of separating data, proposed by Jung et al. [14] is to prevent data generated by different processes from being stored in the same block, on the basis that requests generated by one process are likely to have the same update locality. However, same processes generate frequently and infrequently accessed data consecutively, which is likely to undermine this approach.

Chapter 3

A Single-Layered Read Disturb Management Technique

3.1 Overview

Until 30-nm MLC NAND flash memory, read reclaim is rarely activated because of the strong read-disturb resistance. However, as the techniques for increasing the density of NAND flash memory is evolved, the resistance has been significantly weakened, resulting in quick decreases in the maximum allowable read count between two consecutive block erasures. (In this dissertation, it calls *the maximum read count*.) The read-disturb problem is expected to be more prominent in TLC NAND memory chips. Since a TLC NAND chip has a higher (e.g., $\sim$5%) read voltage and a longer (e.g., $\sim$60%) read operation time compared to an MLC device, TLC NAND cells are likely to be more read disturbed. Figure 6 estimates a future trend on the read-disturb problem by a simple approximation using the FN-tunneling equation (in a similar fashion used for forecasting the trend on MLC devices [2]). As shown in a dashed line with diamond symbols (i.e., the maximum case) of Figure 6, the maximum read count of a TLC NAND chip is estimated to be about 28% on average of that of an MLC device. Moreover, since the read disturbance of a NAND cell is exponentially intensified with

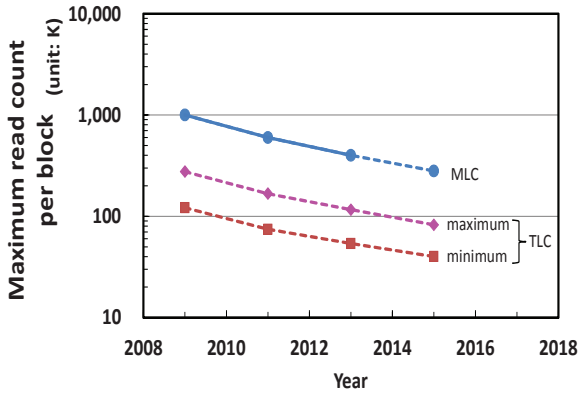


Figure 6: A projected read-disturb trend of future MLC and TLC devices.

the applied read voltage, the maximum read count of a TLC device is expected to be drastically weakened with a higher read voltage. For a block in TLC NAND flash memory, the maximum read count may be just about 40,000. For example, a 5% increase in a read voltage results in a decrease of as much as about 49% in the maximum read count, as shown in a dashed line with square symbols (i.e., the minimum case) of Figure 6. For such a small maximum read count, RR will occur quite frequently in future TLC NAND flash memory.

Frequent read reclaims can negatively affect the performance of NAND flash memory-based storage systems. When a read reclaim procedure is invoked, extra data migrations and a block erasure are performed. Because of these extra operations performed during an RR execution, the overall throughput of NAND flash memory can be significantly degraded. In particular, the I/O response time may fluctuate significantly depending on whether an RR procedure is activated or not. If a normal I/O request conflicts with an

RR procedure, there can be a significant delay in processing the normal I/O request. Such a response time delay can seriously degrade the quality of the service of I/O intensive business applications such as electronic commerce, where stable response times are regarded as one of the most important storage design constraints [16].

Although read disturbs can cause a serious negative impact on the performance of NAND flash memory, the read-disturb problem has not been extensively investigated compared with other NAND flash reliability issues such as limited endurance and data retention. Most existing read-disturb management techniques employ a simple *reactive* solution based on some predictive measures on the severity of read disturbance without considering performance penalty for activating an RR procedure [17]. A straightforward technique uses the number of performed read operations of each block to predict the read-disturbance status of a block. When a block undergoes more read operations than a preset upper bound on the maximum read count of a block, an RR procedure is triggered [4]. (We denote this method as `baseline` because it is used as a baseline read-disturb management technique in this paper.)

In the evaluations of the `baseline` technique with several benchmark traces under varying maximum read count, we have observed that it is very inefficient under *small* maximum read counts. For example, when the maximum read count is 40,000, an FTL based on the `baseline` technique causes a large number of data migrations, increasing the total execution times for extra operations during RR by about 8 times on average over when the maximum read count is 200,000. Moreover, the *reactive* data migration

policy of the `baseline` technique incurs large variations on I/O response times, because once the degree of read disturbance of a disturbed block is high, the `baseline` technique moves all the valid pages in the disturbed block together, thus significantly increasing the I/O response time.

This significant performance penalty is mainly because frequently-read data (which denoted as *read-hot* data) are read from a small number of blocks. Since `baseline` does not modify the read skewness of a given workload, it simply moves the same read-hot data to a different block, thus repeating RR soon. In order to mitigate the skewness of read requests in a given workload, We propose a novel read-disturb management technique which reduces the occurrence of RR. Our technique detects read-hot pages in a partially disturbed block and *proactively* moves them to other healthier blocks before RR is activated. By distributing read requests, the proposed technique reduces RR occurrences. Moreover, by avoiding simultaneous data migrations, the proposed technique better balances I/O response times under RR activations. Based on the proposed technique, we have designed a new read disturb-aware flash translation layer (RedFTL) for high-density NAND flash memory. Experimental results show that RedFTL can reduce the number of RR activations, on average, by 60% over the `baseline` technique. Experimental results show that RedFTL can reduce the time overhead of RR on average by 50% over the `baseline` technique.

3.2 Performance Implications of Read Disturbs

3.2.1 Effect of Frequent Read Reclaims

In order to understand how the existing read-disturb management technique works for a high-density NAND flash-based storage system, we evaluated the `baseline` technique using a trace-driven simulator with four read-intensive benchmark traces. Experimental results show that frequent occurrences of RR introduce considerable extra NAND operations for data migrations and block erasures. Figure 7 shows how extra operations during RRs change under different maximum read counts. The x-axis denotes various maximum read counts, and the y-axis represents the total number of extra operations performed. As shown in Figure 7, the case when the maximum read count is 40,000, frequent RR activations resulted in about 8 times increase in the total extra operations over when the maximum read count is 200,000. In particular, read and write operations for data migrations during RR occupied the majority of the total extra operations. This result indicates that a large number of valid pages exist in disturbed blocks, and the time overhead of migrating these valid pages during RR activations can be considerable if they are simultaneously moved. This long data migration time is the main source of I/O response time fluctuations.

Our evaluation results show that the performance degradation from using the existing simple read-disturb management technique such as `baseline` is so severe that they are not appropriate for high-density NAND flash memory with a small maximum read count. Since the maximum read count will be getting smaller because of the weakened read-disturb resistance of high-

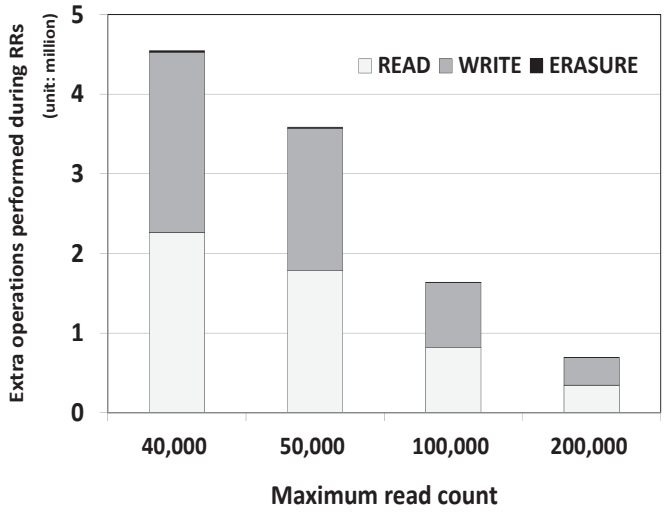


Figure 7: A breakdown of extra operations from read reclaims over varying maximum read counts.

density NAND flash memory, in order for high-density NAND flash memory to be widely adopted in various storage products, it is critical to devise a more efficient read-disturb management technique.

3.2.2 Effect of Read Reclaims on Response Time Fluctuations

A read reclaim incurs a significant fluctuation of I/O response times because a large number of page mitigations are necessary during an RR procedure. For example, Figure 8 shows a snapshot of response time variations for the ads [18] benchmark trace when data in disturbed blocks migrate to healthier blocks. The x-axis and the y-axis represent the logical request time and the read response time, respectively. The logical request time increases by one whenever a read request is performed in NAND flash mem-

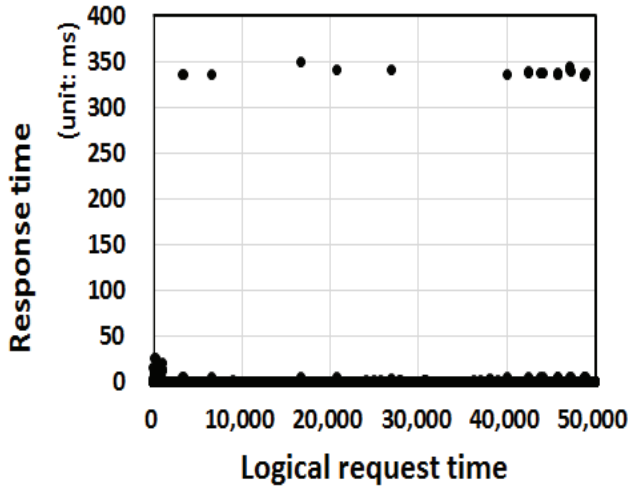


Figure 8: A snapshot of response time variations when the ads benchmark trace is executed.

ory. In Figure 8, there are several high peaks of the response time which correspond to read reclaims. Since all the valid pages should be moved to a healthier block simultaneously in order to avoid data corruption by read-disturb errors when an RR procedure is triggered, the time overhead of an RR procedure is directly proportional to the number of valid pages in the disturbed block. The more valid pages exist in a victim block, the larger required time to migrate them. Unfortunately, most of blocks involved in an RR procedure are filled with many valid data. For example, in the experiments, we observed that, on average, about 99% of pages in a block are moved during RR procedures, thus taking a long time to complete a single RR activation. If an RR activation must move a block with a large number of valid pages, response times can increase up to about 326 *ms*, about 3,260 times increase over the average read operation time of 100 *us*. If page mi-

grations from an RR procedure can be distributed over a longer period, there could be less response time fluctuations.

3.2.3 Effect of SSD Read Buffer on Read Reclaims

When the read-disturb problem is discussed, one straightforward solution seems to be a large SSD read buffer. If the SSD read buffer is large enough to cover most read requests (which were not served by a page cache in an operating system), it is clear that no read-disturb error is likely to occur. In order to understand the effect of SSD read buffer on read reclaims, we performed several experiments using an FTL simulator with four read-dominant I/O traces. (For a detailed description of the experimental setup, refer to Section 3.5.) Since these I/O traces were collected at the block device level, all I/O requests in the traces are actual I/O requests sent to SSDs from a page cache of a kernel. Table 1 summarizes read hit ratios under different read buffer sizes. (These read buffers are all assumed to be managed by the LRU scheme.) As shown in Table 1, except for the case of multi using a 256 MB read buffer, the read hit ratios are low. On average, about 78% of read requests are served by NAND flash chips. (Since the read buffer is likely to be shared by multiple programs at the same time, the read hit ratio

Buffer size	**Benchmark**			
	ads	**websearch**	**tpc-h**	**multi**
64 MB	1%	0%	28%	36%
128 MB	2%	1%	29%	36%
256 MB	3%	2%	30%	96%

Table 1: Read hit ratios of various SSD read buffers.

may be even lower than the experimental results.) This low read hit ratio of a large SSD read buffer comes from large working sets of read-intensive traces. Our evaluation using read-intensive workloads demonstrates that a large SSD read buffer alone cannot solve the read-disturb problem, thus requiring a different solution.

3.3 Read Disturb Management Techniques

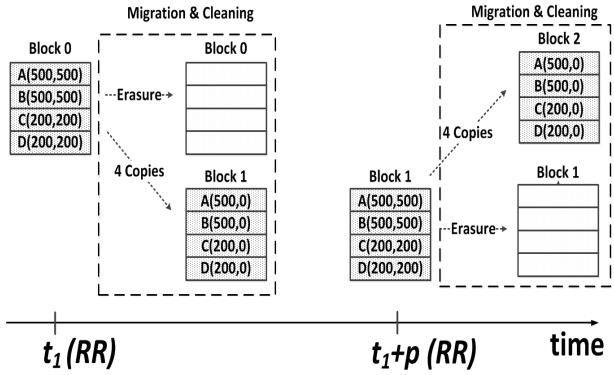
3.3.1 Data Distribution Technique

In order to understand how the `baseline` technique works for a future high-density NAND flash memory, we evaluated `baseline` using read-intensive applications. Figure 7 shows to what extent the total overhead execution time increases during garbage collection (GC) and RR when `baseline` is applied. The x-axis denotes various maximum read counts, and the y-axis represents the normalized total overhead execution times. On the x-axis, the ∞ maximum read count indicates when no RR is activated. Each overhead execution time is normalized to the total execution time spent for GC for the ∞ case. As shown in Figure 7, the smaller the maximum read count is, the more overhead time is spent because of more frequent RR activations. In particular, when the maximum read count was 40,000, data migrations during RR accounted for 88% of the total overhead execution time. This result shows that lots of valid pages exist in disturbed blocks, and the time overhead of moving them during RR can be considerable if they simultaneously migrate.

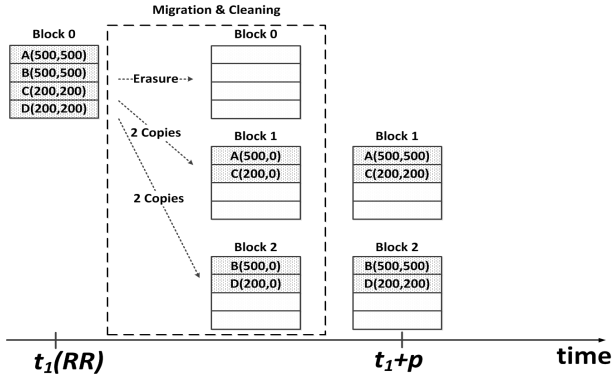
From a detailed analysis, we observed that a small number of heavily

read blocks are responsible for frequent RR activations in `baseline`. Figure 9.(a) illustrates why `baseline` works poorly using an example. Each block has four pages, and each page is represented with a rectangle. A tuple (I, N) in a rectangle indicates that data in a page are read I times per given time period p, and the data have been read N times since the last block erasure. Assume that the RR threshold value is 1,400. As shown in Figure 9.(a), RR is invoked for Block 0 at time t_1 because the read access count to Block 0 reaches the RR threshold. As a result, the pages A, B, C, and D are copied to Block 1, and Block 0 is erased. Since those pages are read 1,400 times for every time period p, RR is invoked for Block 1 once again at time $(t_1 + p)$. If the frequently read Pages A and B had not been migrated together to the same Block 1, the second RR might have been avoided because more blocks could be evenly read.

Based on this observation, we propose a one-to-many data migration technique which splits pages in the same block into multiple groups and moves each group to a different block. Our technique detects read-hot pages in a partially disturbed block and moves them to less disturbed blocks during RR. Those moved read-hot pages are less likely to cause another RR activation because they have been moved to blocks with small read counts. In Figure9(b), the proposed technique moves the pages A and C to Block 1, while Pages B and D are copied to Block 2. In this case, RR does not occur at time $(t_1 + p)$ due to low read access counts in each block. If Blocks 1 and 2 are erased by GC or WL before their read access counts get close to the RR threshold value, their read disturbance can be fully recovered without RR.



(a) The `baseline` technique



(b) The proposed one-to-many migration technique

Figure 9: A snapshot comparison of RR using different data migration techniques.

RR also incurs a significant fluctuation of I/O response times. Figure8 shows a snapshot of response time variations after about 800 million read requests of the websearch benchmark trace were performed. The x-axis and y-axis represent the *logical* read access time and I/O response time for a read request, respectively. The logical read access time increases by one whenever a read operation (to any page) is performed. The maximum read count was set to 40,000. In Figure 8, there are many high peaks of the re-

sponse time because of simultaneous data migrations and block erasures during RR. In the observation, 85% of pages in a block are moved during RR, thus taking a long time to complete a single RR activation. If a block filled with a large number of valid data is erased during RR, as shown in Figure 8, the response time is increased to 332 *ms*, about 66 times increase over a normal block erasure response time. If several pages in those valid pages are moved to other healthier blocks before an RR activation, there would be less response time fluctuations.

3.3.2 Proactive Data Migration

Based on this observation, we also suggest a proactive data migration technique as a part of the main technique, which mitigates the fluctuations of I/O response time. By moving potential read-hot pages in advance before an RR activation, the proposed technique spreads the time overhead of data migrations for a longer time period.

3.4 RedFTL: Read Disturb-Aware FTL

3.4.1 Overview of RedFTL

Based on two ideas explained in Section 3.3, we have designed a new read disturb-aware FTL, called RedFTL, for high-density NAND flash memory. Figure 10 shows an organizational overview of RedFTL. It consists of common modules of a typical FTL as well as several special modules which are specifically designed for read-disturb management support such as a read-hot page separator, a good block pool, a migration manager, and a

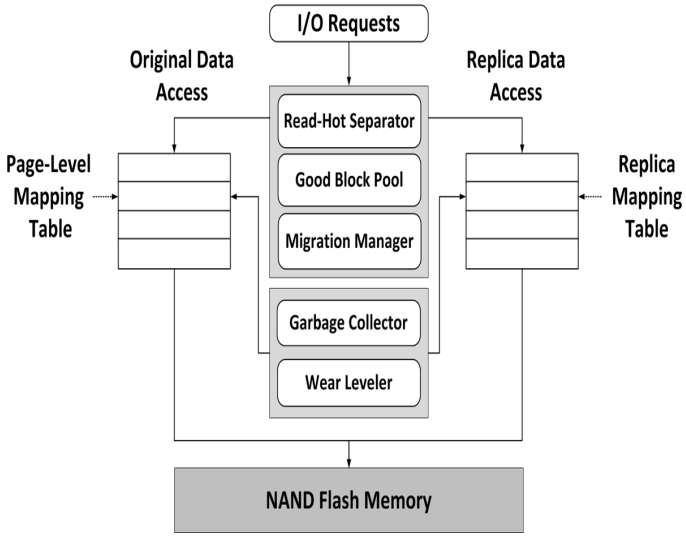


Figure 10: An organization of RedFTL.

replica mapping table.

For a given read request for a block, RedFTL first checks whether RR is likely to be activated soon for the block or not. If the read access count of the block is greater than a preset threshold, called the *replica creation threshold*, RedFTL detects read-hot pages in the block and moves them to other healthier blocks in order to avoid an occurrence of RR. The read-hot page separator classifies read-hot pages in a block based on the read access pattern of each page. The migration manager copies a replica of a read-hot page to a healthier block, and the location information of the replica page is updated in the replica mapping table. RedFTL distributes read requests to more blocks by changing read requests to be read in a replica page if the replica page exists. When a replica page is created, if the original read-hot page of a replica page is immediately invalidated, the block with the origi-

nal page is likely to be selected as a GC victim block because the number of invalid page was increased. In order to avoid such GC behavior, the migration manager makes a copy of the original page instead of moving the original page. A replica pages is invalidated either when the original page of the replica page is updated or when the block with the replica page is erased during garbage collection or wear leveling. Once replica pages in a block have serviced a significant number of read requests, the replica page is permanently moved to the new block by invalidating the original read-hot page of that replica page. This is determined by checking whether the read access count of the block with a replica page exceeds another preset threshold, called the *migration threshold*, or not.

3.4.2 Read-Hot Page Separation

In order to select a read-hot page in a block, the read-hot page separator compares the read-access rate R_P of a page P with the read-access rate R_B of a block B. The read-access rate R_P is defined as $\frac{r_P}{t_P^{last} - t_P^{first}}$ where r_P denotes the read access count of the page P, and t_P^{first} and t_P^{last} represent the first and the last logical read access times of the page P, respectively. The read-access rate R_B can be defined similarly by $\frac{r_B}{t_B^{last} - t_B^{first}}$ where r_B indicates read access count of the block B, while t_B^{first} and t_B^{last} denote the first and the last logical read access times of the block B, respectively. The page P is classified as a read-hot page if $R_P > \alpha \times R_B$. The constant parameter α is used to control the access skewness of read requests in determining a read-hot page. (In the current version, we set α with 2 based on several experiments.) If many pages in the block are continuously read at a similar pace, they are likely to

activate RR frequently if they remain in the same block together. Thus, if read accesses to the block B are almost evenly distributed among its pages, the read-hot page separator selects randomly a half of the valid pages as read-hot pages.

3.4.3 Good Block Pool Management

RedFTL manages a pool of good blocks, which denoted as the *good block pool (GBP)*. GBP maintains less disturbed healthy blocks, which are used in allocating the replicas of read-hot pages. In order to prevent read-hot pages in a block from being stored in the same block, a read-hot page is allocated to a healthier block in the GBP according to FIFO. If a block stores more replica pages than a preset maximum number of replica pages per block, the block is removed from GBP because storing a large number of replicas in a block may waste too much space. Moreover, if a replica page is created in a block, this block is removed from GBP because a block with one or more read-hot pages may have been already partially disturbed.

3.5 Experimental Results

A trace-driven FTL simulator was used in the experiments to evaluate the proposed technique. Tables 2 and 3 summarize various parameters of the simulator and the characteristics of the benchmark traces used for the experiment, respectively. These parameters are based on the recent TLC NAND specification [19]. GC was triggered when the total number of remaining free blocks was less than 4% of the total number of blocks, and it was con-

Table 2: Key parameters of the FTL simulator for experiments

Flash Setting	Value	FTL Setting	Value
Pages per Block	192	Mapping	Page Level
Page Size	8 KB	GC	Greedy Policy
Page Read Latency	100 us	WL	Swapping
Page Write Latency	1,600 us	Buffer Size	256 MB
Block Erasure Latency	5 ms	RR Threshold	38,000

tinued until 6% of the entire blocks became free blocks. In the simulator, wear leveling is activated if the difference of P/E cycles between the oldest block and the youngest block is greater than 40, but it was not triggered in the experiments. The entire blocks was set to 65,536 except for mds. Since mds requires more blocks due to its large working set size, the number of entire blocks was set to 287,995 which is five times of the working set size of mds. Furthermore, migration and replica creation threshold values were set to 90% and 70% of the maximum read count, respectively. We used highly read-dominant public benchmark traces which were collected from actual systems. The trace interval in Table 3 indicates the length of the time interval during which a corresponding trace was collected. We repeated the same benchmark trace multiple times to generate enough read requests in the experiment. The number of iterations for each trace is indicated as the repeat count in Table 6.

Figure 11 shows normalized overhead execution times for data migrations and replica copies during RR when 15 pages in each block were used to store replica pages. The x-axis indicates benchmark traces and applied techniques, and the y-axis denotes the execution time for RR which is normalized to that of `baseline`. Since RedFTL creates the replica pages of

Table 3: Summary of benchmark traces

Benchmark	Description	Read (%)	Trace Interval	Repeat Count
ads [18]	Display ads platform	96	1 day	150
mds [20]	Media server	98	1 week	200
tpc-h [21]	Accesses to a database	92	10 hours	50
websearch [22]	Search engine	100	4 days	100

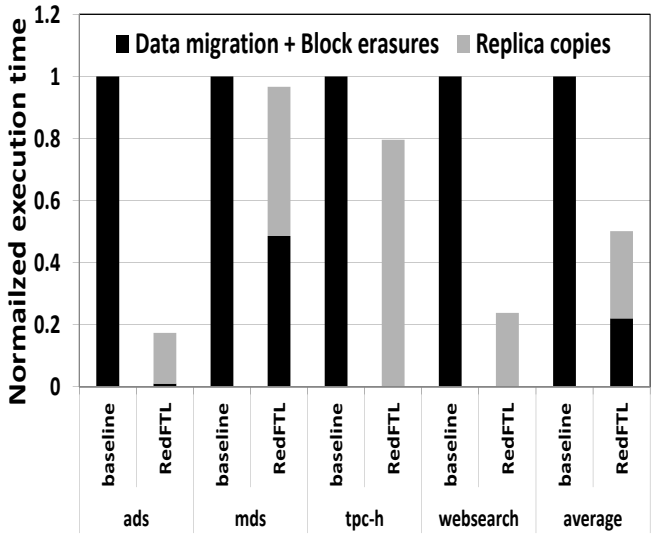


Figure 11: A comparison of the normalized overhead execution times for read reclaim.

read-hot pages and places them to multiple blocks, an extra time overhead occurs. In Figure 11, in the case of tpc-h, many replica pages were created, but they were vanished by frequent GC procedures, thus increasing the extra time overhead. Moreover, in the case of mds, a significant number of pages were classified as read-hot pages by the read-hot page separator because many pages were evenly read at a similar pace. Although this time overhead occupied 56% of the total execution time for RR, RedFTL decreased

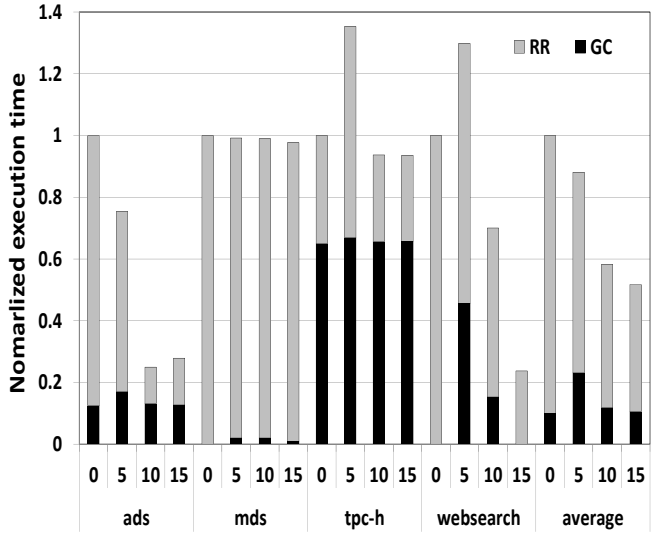


Figure 12: A breakdown of the normalized total overhead execution times.

the overall execution time for RR, on average, by 50% over `baseline` because it reduced the time overhead for data migrations by 78%.

Figure 12 illustrates the total overhead execution time for GC and RR. The x-axis denotes the maximum number of replica pages per block and benchmark traces, and the y-axis represents the overhead execution time which is normalized over `baseline`. The `baseline` is represented by 0 on the x-axis. As shown in Figure 12, RR activations were decreased as the maximum number of replica pages per block gets larger. Since a large number of replica pages can contribute to evenly distribute the read skewness of a given workload, the occurrences of RR was reduced. However, creating replica pages negatively affected performance in the cases of websearch and tpc-h when 5 pages per block are allowed to store replica pages. A small number of replica pages do not service many read requests. Fur-

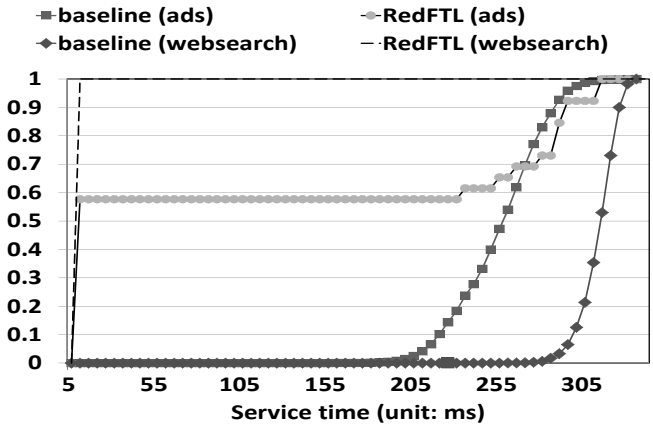


Figure 13: CDFs of service times of RR for `ads` and `websearch` in redFTL and `baseline`.

thermore, GC may be frequently activated because more many blocks are required to store the same number of replica pages as the number of replica pages per block decreases. Although creating replica pages increased total overhead execution time in the two cases, RedFTL reduced the total overhead execution time for GC and RR, on average, by 34% over `baseline` by reducing the number of RR occurrence.

Figure. 13 shows the cumulative distribution functions (CDFs) of the service times of RR procedures for ads and websearch. In both ads and websearch cases, the proactive data migrations of RedFTL limited most service times less than 10 ms. On the other hand, since baseline does not activate RR until the number of reads reaches the RR threshold, most of the valid pages in a disturbed block were simultaneously moved in the experiments, with the peak response time occurring around 332 ms.

Chapter 4

An Integrated Approach for Read Disturb Management

4.1 Overview

`RedFTL` reduces the read-reclaim management overhead with data distributions and proactive read reclaims. It converts highly skewed read accesses to a small number of blocks into more balanced read accesses to a large number of blocks by changing data block locations accessed, thus reducing the occurrences of read reclaims. However, `redFTL` has fundamental limitations to completely resolve the read-disturb management problem. First, `redFTL` is sensitive to the patterns of read requests. If all read blocks in all read requests are evenly read, `redFTL` does not work because there is no read-hot data. Although it is less likely that the whole blocks are evenly read, the effectiveness of `redFTL` is decreased if the skewness of read requests is weakened. Second, `redFTL` cannot reduce occurrences of read reclaims if the number of read requests reaches the same number of the maximum read counts of SSD. Since the total maximum read counts itself does not change regardless of read request access patterns, distributing read-hot data in `redFTL` cannot contribute to reduce occurrence of read reclaims in such situation.

In order to fundamentally and efficiently manage the read-disturb problem, we propose a novel integrated approach for the read-disturb problem. The proposed approach is based on the key observation on read-disturb errors from the NAND device physics that the degree of the read disturbance to neighboring pages is a function of the read voltage and the read operation time. Since the degree of the read disturbance of a block is exponentially dependent on the read voltage applied to unselected word lines when a page is read, lowering the read voltage can significantly increase the maximum read count of a NAND block.

From NAND characterization tests, we built an accurate NAND read-disturbance model under different read voltages and we extended the existing NAND interface to support multiple read modes with different read voltages. In order to lower read voltage, threshold voltage distributions should be placed within narrow windows, thus increasing the program time of a page due to a fine-grained programming method. By exploiting this new tradeoff between the read disturbance and program time, the approach mitigates the read disturbance significantly without a noticeable program time increase.

Another key tradeoff is that the read disturbance has a linear dependence on the read operation time. For the same number of page reads, the maximum read count of a block can be effectively increased by increasing the ratio of fast page reads to slow page reads in the block. In a TLC NAND flash block (where there are three types of pages, LSB pages, CSB pages, and MSB pages), reading an LSB page is faster than reading a CSB page or an MSB page. Therefore, if more read requests can be serviced from LSB

pages (over CSB and MSB pages), the read disturbance of the block is mitigated. In order to make LSB pages to serve more reads, when an RR procedure is activated, frequently-read data in a disturbed block are moved to LSB pages of a healthy block so that more future reads to these data can be serviced from LSB pages (which can be considered as more *read resistant* over the other types of pages).

In order to reduce large variations in I/O response times of the `baseline` technique, we propose a proactive background data migration technique which moves frequently-read pages in a partially read-disturbed block in advance before the block is seriously read-disturbed. The proposed data migration technique triggers a background RR activation when there are enough idle times during runtime. By *proactively* moving valid pages in a disturbed block by a background RR thread, the proposed technique can significantly improve the I/O response time over the `baseline`'s on-demand RR technique.

As an advanced version of `redFTL`, we developed `redFTL+`, which exploits physics of NAND flash memory to resolve read-disturb problems in high-density NAND flash memory. We evaluated the effectiveness of `redFTL+` with an extended FlashBench emulation environment [23] which supports multiple read modes from the proposed read-disturbance model. The experimental results using six read-intensive benchmark traces show that `redFTL+` can reduce the total execution times for extra operations during RR, on average, by 78%. Furthermore, the proposed proactive approach of `redFTL+` mitigates the response time fluctuations during RR activations. `RedFTL+` reduced the response times of read requests, on average, by 237

times over the `baseline` technique.

4.2 Read Disturb Management Techniques

Since both a read voltage and a read operation time have a critical effect on the degree of the read disturbance, reducing them can improve the read-disturb resistance of NAND flash memory. In particular, reducing a read voltage is the most effective way of mitigating the effect of the read disturbance because the FN-tunneling effect is exponentially proportional to the applied voltage. Once a read voltage is fixed to during the device design time, making a read operation time short is another useful option of increasing the read-disturb resistance. Since a read operation time is fixed during the device design time, however, we cannot easily change the read operation time at the S/W level. However, we can exploit different read times among different page types. For example, in an MLC device, read times for LSB pages and MSB pages are different. As shown in Figure 4(b), in order to read an LSB page, only one reference voltage (i.e., V_{r_2}) is used while two reference voltages (i.e., V_{r_1} and V_{r_3}) are necessary for reading an MSB page. Since a read operation time is linearly proportional to the number of voltage sensing operations, the read operation time for an MSB page is two times longer than that for an LSB page. As a result, the degree of the read disturbance from reading an MSB page is about two times higher than that from reading an LSB page. For a TLC chip, there is a similar imbalance of read operation times depending on its page types such as LSB, CSB, and MSB [19]. If we can intelligently exploit the operation time differences

among different page types, the total read operation time can be effectively reduced, thus mitigating the read-disturb problem.

4.2.1 Mitigation of Read Reclaims by Read Voltage Scaling

Considering the relationship between the applied voltage and the current density of the FN-tunneling equation [7], we can mitigate the read-disturb problem by lowering the read voltage during read operations. In order to use a lower read voltage when a NAND page is read, it is necessary to form narrow threshold voltage distributions when a program operation is performed using a fine-grained program control which takes a longer time to write a page [24]. In order to precisely classify the cell state, the read voltage must be set high enough so that it does not touch the tail of a threshold voltage distribution of the highest program state. Furthermore, each program state should be separated by a minimum voltage margin to distinguish different states reliably. Because of these NAND device constraints, the read voltage must be scaled together with other NAND parameters such as preset programming voltages so that the essential voltage margin between the read voltage and the programming voltage can be maintained.

Figure 14 illustrates how read voltage scaling affects threshold voltage distributions for four-state MLC devices. In conventional NAND chips, the voltage gap Mp_i between two adjacent program states is kept large enough to meet the data retention time requirement. On the other hand, the width Wp_i of a program state is mainly affected by a program-time requirement.

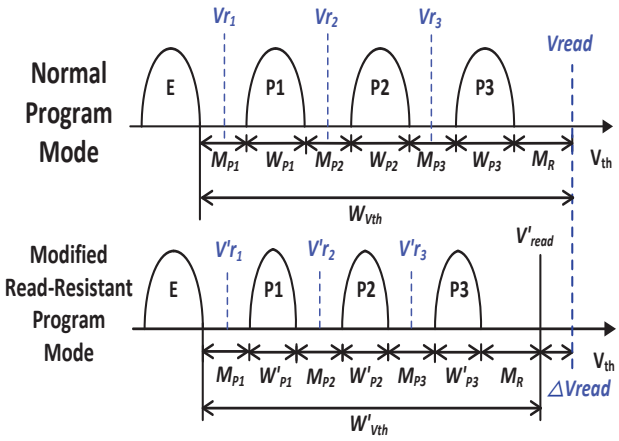
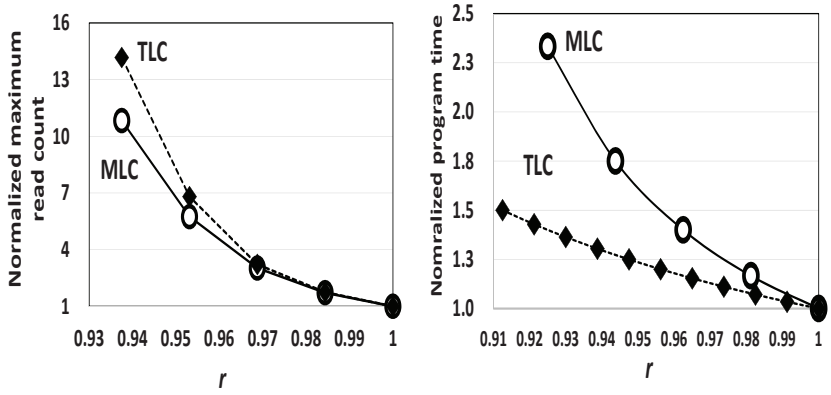


Figure 14: An example of a read voltage shifting by narrowing the width of a threshold voltage distribution.

As a result, the total width Wv_{th} of threshold voltage distributions are carefully designed during the device design time to meet these NAND requirements. For example, the threshold voltage difference M_R between the P3 program state and the read voltage V_{read} is fixed during the device design time by a flash manufacturer. In order to lower the read voltage by ΔV_{read}, as shown in Figure 14, narrow threshold voltage distributions should be formed, reducing Wv_{th} to $W'v_{th}$ by ΔV_{read}. By reducing each Wp_i by $\frac{\Delta V_{read}}{3}$, we achieve a total reduction of ΔV_{read}. Since there is a tradeoff relationship between the program time and the width of a threshold voltage distribution [24], a reduced $W'p_i$ requires a longer program time.

In order to observe the relationship between the read-disturb resistance and the read voltage, we measured the read-disturb resistance of MLC and TLC NAND blocks under different read voltages. In experiments, we measured the maximum read count of NAND flash memory with recent 20 nm-



(a) Maximum read count variations under different read voltages in MLC and TLC blocks.

(b) Program time increases over different read voltages in MLC and TLC blocks.

Figure 15: The effect of lowering the read voltage on the read resistance and program time.

node NAND chips for quantifying the degree of read disturbance. The maximum read count of each block is determined with the number of read cycles of a block whose number of bits errors exceeds the recoverable bits by a 40-bit ECC. Figure 15(a) shows how many read requests can be serviced when a read voltage is decreased. In Figure 15(a), the x-axis denotes the read voltage scaling ratio r, and the y-axis represents the normalized maximum read count which is normalized with those of normal MLC and TLC blocks, respectively. When the read voltage scaling ratio r is set to x, the read voltage is reduced to $x\%$ of the normal read voltage. As shown in Figure 15(a), in both MLC and TLC blocks, the maximum read counts are dramatically increased with lower read voltages. For example, when a read voltage of a TLC chip is decreased by about 4% (denoted as 0.96 in the x-axis), the maximum read count increases by about 5 times. The measurement result

confirms that reducing the read voltage is the most effective way of solving the read-disturb problem.

In addition to the tradeoff between the read voltage and the read resistance, Figure 15(b) shows the relationship between the program speed and the applied read voltage in MLC and TLC blocks. The x-axis denotes the read voltage scaling ratio r, and the y-axis represents the normalized program time (where the program times of normal MLC and TLC blocks are used as baseline cases). As shown in Figure 15(b), the program time is increased as the read voltage is reduced. Since the width of threshold voltage distributions of all program states are narrowed by a fine-grained program method, the amount of read voltage reduction is proportional to the number of states of NAND cells. In other words, in order to reduce the read voltage by the same ΔV_{read}, an MLC chip with three program states needs to shorten Wp_i's more than a TLC chip with seven program states, thus increasing the program time of MLC chips more sharply as r gets smaller. Although the fine-grained program method takes longer time to shorten the widths of threshold voltage distributions, program time increases are acceptable because read-disturb errors are likely concerns for read-dominant workloads where the program speed may not be an important issue.

In order to efficiently exploit read voltage scaling for reducing the read reclaim overhead, we introduce a special NAND block, called a *read-resistant block* (RRB). Unlike a normal block, a proposed RRB can support multiple read modes using different read voltages. Table 4 summarizes key tradeoff relationships among three read modes[1]. R_{mode_0} uses the nominal

[1] As shown in 15(a), we can use a more read-resistant mode (than R_{mode2}) by lowering

read mode	r	increase in mrc	ΔT_{prog}	program mode used
R_{mode_0}	1.00	$\times$ 1	0%	W_{mode_0}
R_{mode_1}	0.98	$\times$ 2	8%	W_{mode_1}
R_{mode_2}	0.96	$\times$ 5	19%	W_{mode_2}

r: read voltage scaling ratio
mrc: maximum read count
$\Delta \mathbf{T_{prog}}$: program time increase

Table 4: Proposed read modes of an RRB with different read voltages.

(i.e., the highest) read voltage while R_{mode_2} uses the lowest read voltage (which is 96% of the nominal read voltage). If an RRB B were to be read by R_{mode_2}, it must be first programmed by W_{mode_2} whose program time is 19% longer than a normal block[2]. However, using R_{mode_2} increases the maximum read count of B by 5 times over using R_{mode_0}. (Since R_{mode_i} can be used only for a block written in W_{mode_i}, where no confusion arises, we use the terms R_{mode_i} and W_{mode_i} interchangeably with $mode_i$.)

Since an RRB can be created by decreasing read voltage of a block, we need to modify an existing NAND flash memory chip architecture. which can change read voltages applied to wordlines of a NAND flash block. Figure 16 shows how the proposed NAND chip architecture can properly change read voltage applied. If a read command occurs, an address is passed

the read voltage scaling ratio r more than 0.96. However, as NAND flash memory technology scales down, it may not be possible to use r less than 0.96. For example, as the density of NAND flash memory is increased, the width of threshold voltage distributions are also getting bigger by side effects of scaling down such as cell-to-cell interference [25] and random telegraph noise [26, 27]. Considering the wider width of threshold voltage distributions, only a small voltage range is available for read voltage reduction, thus limiting the maximum allowed read voltage scaling ratio. In this dissertation, we limit r by 0.96.

[2]If pages in a block are programmed with various write modes, different read voltages must be applied to the pages when they are read. Since an inadequate read voltage applied can corrupt data in a page, the block must be read and written by the same mode.

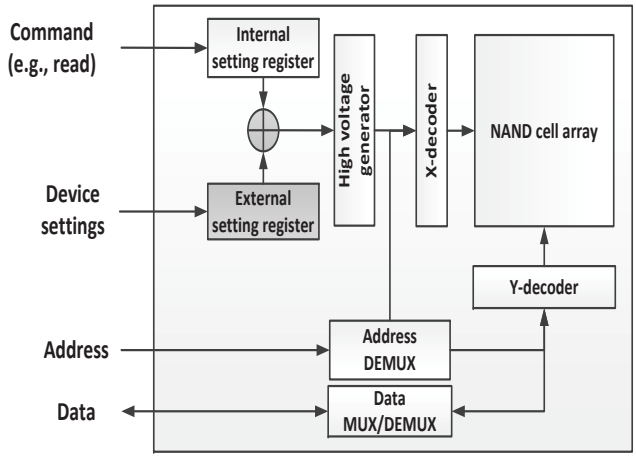


Figure 16: An overall block diagram of the proposed new NAND chip architecture.

to a NAND flash block in order to read the requested data from the proper location where the given address points out in a NAND cell array. Before a read voltage is applied to the wordlines in the NAND chip to perform read operation, the read voltage is determined by an internal setting register and a high voltage generator. The internal setting register stores predetermined digital code values, and the high voltage generator tunes a read voltage according to the digital code values by converting them to analog voltages. As the digital code value is lower, the output read voltage gets smaller.

In order to support our proposed multiple read and write modes, the existing NAND chip architecture should be modified so that different read voltages can be applied to wordlines of a NAND flash block. Figure 16 gives a high-level overview of our proposed new NAND chip architecture. Before a read voltage is applied to the wordlines in the NAND chip to perform read operation, the default read voltage is determined by an internal setting

register and a high voltage generator. The internal setting register stores predetermined digital code values, and the high voltage generator tunes a read voltage according to the digital code values by converting them to analog voltages. In order to change the read voltage, a new read voltage is passed to an external setting register using appropriate digital code values. The high voltage generator changes its output voltage by combining values of both the internal and external setting registers. Since combining two register values can be done quickly with a negligible resource overhead, our proposed NAND chip architecture can provide multiple read and write modes without significant time and area overheads.

4.2.2 Mitigation of Read Reclaims by Read Operation Time Scaling

The read operation time is another important factor affecting read-disturb errors. In order to observe the relationship between the read-disturb resistance and the read operation time, we measured the read-disturb resistance of reading different page types in an MLC block and a TLC block using 20 nm-node NAND chips. Figure 17 shows how different types of page reads can affect the maximum read count. The y-axis denotes the normalized maximum read count under different page access behavior. Since the maximum read count of a block is determined as the worst case where the slowest operations are always performed until the block causes read-disturb errors, the maximum read count is normalized with the maximum read count of the slowest page type. The x-axis represents different read workloads for

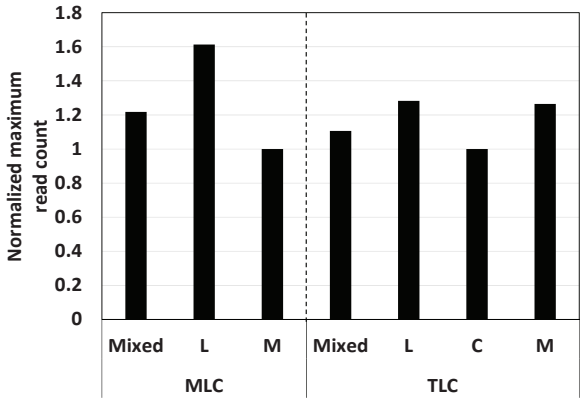


Figure 17: Maximum read counts under different page types of MLC and TLC blocks.

MLC and TLC NAND chips. For example, L/C/M in the x-axis indicates the case where only LSB/CSB/MSB pages in blocks were read, respectively. On the other hand, Mixed represents the case where different page types were evenly read.

For MLC chips, since reading an LSB page takes about 50% shorter over an MSB page, L increases the maximum read count by 61% over M. For TLC chips we tested, reading a CSB page took about 50% longer over reading an LSB page or MSB page. The evaluation result roughly agrees with the difference in the read time, increasing the maximum read count of L and M by 28% and 26%, respectively over C. For different TLC chips where reading an MSB page takes longer than a CSB page or an LSB page [28], we expect a similar difference in the maximum read count of a block depending on read workloads.

In this dissertation, when a page supports a fast read over other pages, we call such pages *read-resistant pages* (RRPs), because a block can serve

more read requests when such pages are more frequently read than slow pages. In order to reduce the read disturbance of a block, we can effectively reduce the performed average read time by reading more data from RRPs.

4.2.3 NAND Read-Disturbance Model

Combining the effect of read voltage scaling and read operation time scaling on the read disturbance of NAND blocks, we developed a novel read-disturbance model that can be used with the proposed RRBs and RRPs. Since the read disturbance of a block from a single read varies depending on the applied read voltage and the type of a page read, we introduce a new read disturbance metric, called *effective read disturbance per read* (in short, *effective read disturbance*), which indicates the effective degree of the read disturbance to the NAND block after a specific read. When a type j page is read by using R_{mode_i}, the effective read disturbance is given as $\frac{\beta_j}{\alpha_i}$ where α_i $(0 \leq i \leq 2)$ denotes a normalized maximum read count (over the maximum read count of a normal block) when R_{mode_i} is used and β_j indicates an effective amount of read disturbance to a block when the type j page is read. For example, in case of the TLC chips we tested in Section 4.2.2, when an LSB page of a TLC block B is read by R_{mode_2}, the block B is effectively read-disturbed by about 0.16, because α_2 and β_{LSB} are 5 and 0.78, respectively. In other words, we can perform about 6 times more reads when the LSB page is read from an RRB with R_{mode_2} over when the CSB page is read from a normal block. When all the read operations (using different combinations of a read voltage and a read page type) to a block B are considered, we can compute the total sum of effective read disturbance

as $\sum_{i \in RMODE} \sum_{j \in PTYPE} \frac{\beta_j}{\alpha_i} \cdot n_{(i,j)}$ where $RMODE = \{0,1,2\}$ represents a set of supported read modes, $PTYPE = \{LSB, MSB, CSB\}$ indicates a set of supported page types, and $n_{(i,j)}$ denotes the total number of the type j page reads with R_{mode_i}.

In this dissertation, we developed two NAND read-disturbance models, TLC_{base} and TLC_{opt}, for two different NAND TLC chips. The TLC_{base} model represents a typical TLC design [29] while the TLC_{opt} model is based on the TLC chips used in the measurement study. TLC chips used in building the TLC_{opt} model are believed to have improved a typical TLC design in several directions including a better handling of various reliability issues [28]. Table 5 summarizes the key parameters of TLC_{base} and TLC_{opt} which are necessary in computing the proposed effective read disturbance. As shown in Table 5, the main difference between TLC_{base} and TLC_{opt} is their β_j's. TLC_{opt} has small differences among its β_i's while TLC_{base} has a larger difference among its β_i's. Using TLC_{base} and TLC_{opt}, we can estimate the effective impact of each read on the NAND read disturbance.

model	β_{MSB}	β_{CSB}	β_{LSB}	α_0	α_1	α_2
TLC_{base}	1	0.5	0.25	1	2	5
TLC_{opt}	0.78	1	0.79			

Table 5: A summary of the key parameters of the proposed NAND read-disturbance models.

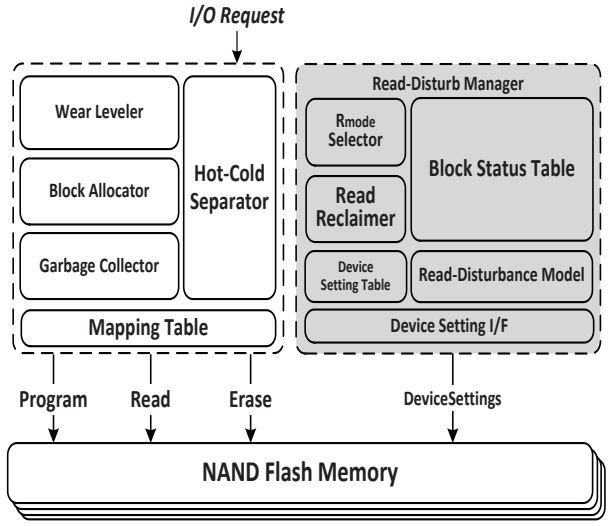


Figure 18: An organizational overview of `redFTL+` with a read-disturb manager.

4.3 Design and Implementation of RedFTL+

4.3.1 Overview

Based on the read-disturb management techniques described in Sections 4.2.1 and 4.2.2, and the new NAND read-disturbance model presented in Section 4.2.3, we have implemented `redFTL+`. Figure 18 shows an organizational overview of `redFTL+`. The read-disturb manager, which is a key module of `redFTL+`, implements the proposed read-disturb management techniques based on the NAND read-disturbance model. For an incoming read request to a block B_{src}, the read reclaimer checks whether the block B_{src} requires a read reclaim or not based on the sum of effective read disturbance of the block B_{src}. If the sum of effective read disturbance of the block B_{src} reaches the preset *proactive RR threshold*, the read reclaimer finds a

target free block B_{tgt} from the block allocator. The read reclaimer copies the valid pages in B_{src} to B_{tgt} by a background thread, thus mitigating fluctuations of I/O response times. Before copying the first valid data in B_{src}, a read mode R_{mode_i} for the block B_{tgt} is determined by the read mode selector submodule. Since a block written by a program mode W_{mode_i} can be read only by the read mode R_{mode_i}, the selected mode information for B_{tgt} is maintained in the block status table. This table is also consulted to decide the $mode_i$ of a block when the block is read or written. Once the write mode is decided, `redFTL+` configures a NAND chip to be accessed using the selected $mode_i$ through a new interface *DeviceSettings* between `redFTL+` and NAND chips. `RedFTL+` sends proper NAND chip configuration parameters (such as the reference voltages V_{r_i}'s and read voltage (V_{read}) for $mode_i$) through the *DeviceSettings* interface to NAND chips.

4.3.2 Dynamic Mode Selection

In `redFTL+`, in order to reduce the occurrence of RRs, when the first read reclaim is activated in a normal block, the valid data of the disturbed normal block are moved to an RRB using W_{mode_1}. Although different W_{mode_i}'s can be used for copying the valid data to the RRB, our current heuristic chooses W_{mode_1} for the first RR activation because using W_{mode_1} can increase the maximum read count of a block by about 100% with the smallest increase in the program time. For subsequent RR activations from the RRB, we gradually employ the more read-resistant write mode, W_{mode_2}, so that future RR activations can be avoided in a more aggressive fashion.

Valid pages stored in an RRB are moved back to a normal block during

GC. When an RRB is selected as a GC victim, the valid data of the RRB are copied using W_{mode_0} (that is, written back to a normal block). Since most GC techniques tend to choose a block with a large number of invalid pages as a GC victim block, if the RRB is selected as a GC victim, it is very likely that many pages in the RRB have been already invalidated. Therefore, when we copy the valid data of the RRB during GC, W_{mode_0} is a logical choice because it is less likely that a small number of valid pages (moved to a free block after GC) will cause another read reclaim.

4.3.3 Distributed Migration to RRBs

In addition to the adoption of an RRB, `redFTL+` reduces read disturbance of NAND flash blocks by distributing read-hot data in the partially disturbed block B_{src} to read-resistant pages in two target RRBs. If all the data in the block B_{src} are simply moved together to another block B_{tgt}, the moved data may cause additional read reclaims from the block B_{tgt} because the same read access patterns may be observed from B_{tgt} as B_{src}. In order to avoid such successive RR activations, `redFTL+` distributes the read-hot data of B_{src} to two target RRBs, thus effectively mixing the block B_{src} with other read-cold data during an RR procedure. When the disturbed block B_{src} in the NAND chip C_n incurs a read reclaim, `redFTL+` mixes the read-hot data of B_{src} with the data in the least disturbed block $B_{coldest}$ of the NAND chip C_n. Since $B_{coldest}$ is the block with the minimum read count in C_n, data in $B_{coldest}$ are the read-coldest data in C_n. Data from the blocks B_{src} and $B_{coldest}$ are copied to two healthy RRBs RRB_{tgt1} and RRB_{tgt2} in a mixed fashion.

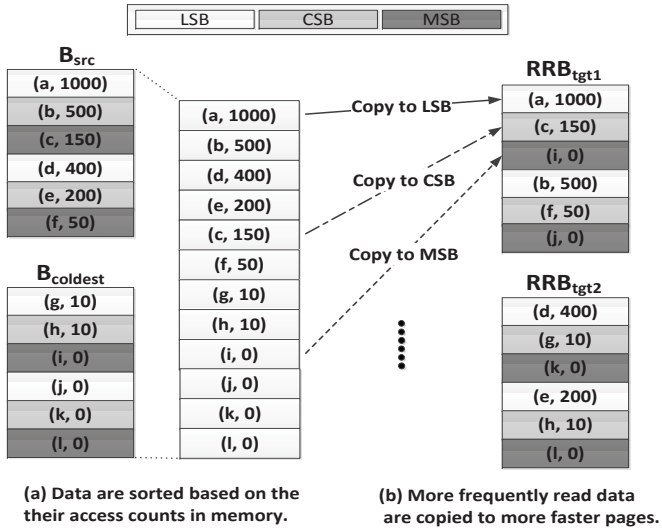


Figure 19: An example of data sorting and migration in `redFTL+`.

In order to exploit RRPs more effectively, `redFTL+` further distinguishes the read-hot data of B_{src} so that hotter data can be moved to more read-resistant pages of a target block. `RedFTL` sorts pages in the blocks B_{src} and $B_{coldest}$ into a descending order based on their read access counts, and copies the sorted pages to target blocks in the LSB-CSB-MSB order (for TLC NAND chips) or the LSB-MSB order (for MLC NAND chips).

Figure 19 shows how `redFTL+` copies data in the blocks B_{src} and $B_{coldest}$ to the target blocks RRB_{tgt1} and RRB_{tgt2}. In Figure 19, a block consists of 6 pages, and the tuple (d_B, t_B) in the rectangle represents that data d in a block B have been read t times since the block B has been erased. As shown in Figure 19(a), data in each page in the blocks B_{src} and $B_{coldest}$ are reordered based on the read count of each page. Once the pages are sorted, the sorted pages are copied to target blocks in the LSB-CSB-MSB order dur-

ing an RR procedure. For example, the frequently accessed read-hot data *a*, *b*, *d*, and *e* are copied to the fast LSB pages in target blocks RRB_{tgt1} and RRB_{tgt2}, while the read-cold data *i*, *j*, *k*, and *l* are moved to the slow MSB pages in target blocks. Since this data migration process is carried out in memory only when a read reclaim is triggered in a block, the performance and resource penalty of a distributed data migration is negligible.

4.3.4 Read-Hotness Detection

When an RR procedure is activated, `redFTL+` determines the read hotness of data in a disturbed block based on their read access history. In devising our read-hotness detection techniques, we have exploited a strong read locality of read-dominant workload. As shown in Figure 20, there exists a small set of dominant logical block addresses (LBAs) that are read very frequently. In Figure 20, the x-axis represents the logical request time which increments by one whenever a read request is performed, and the y-axis denotes LBAs read at that time. As shown in Figure 20, most read requests are repeatedly performed on similar LBAs. Based on the strong read locality of read-dominant workloads, `redFTL+` can decide the hotness of each data by comparing the total number of reads to each data.

However, if `redFTL+` keeps track of exact read counts for the entire LBA range, a memory overhead of storing read counts can be very large (close to the same order of magnitude of implementing the page-level mapping table). In order to reduce the memory requirement, `redFTL+` employs a two-level counter organization in maintaining read counts of LBAs, which requires one extra byte for each LBA in our current implementation. As ex-

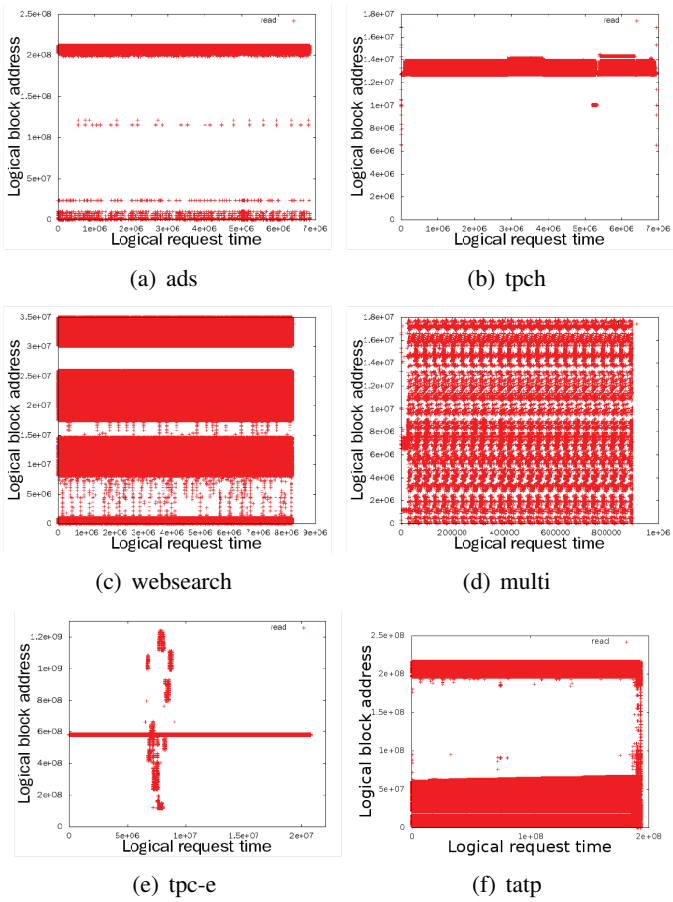


Figure 20: Distributions of read requests in some benchmark programs.

plained in Figure 20, since read-hot LBAs of read-dominant workload have a strong read locality, we can easily distinguish read-hot LBAs from read-cold LBAs by partially counting the number of read accesses to LBAs. For an efficient partial counting, we use a 4-bit saturating counter as the first-level counter. A 4-bit saturating counter increments by one each time its LBA is read, but its counter value saturates at 15. Whenever the total read count of a block B reaches $i \times \frac{maximum\ read\ count}{15}$ ($1 \leq i \leq 15$), `redFTL+`

checks if the first-level counter for a valid page in the block B was saturated or not. If the first-level counter was saturated to its upper-bound value, the second-level 4-bit counter is incremented by one. Once this sampling is completed, the first-level counters are all reset to zero. When an RR procedure is triggered for the block B, `redFTL+` can classify the hotness of each page based on the second-level counter values. In our current implementation, `redFTL+` sorts pages in the blocks B and $B_{coldest}$ into a descending order based on the second-level counter values. Pages with larger second-level counter values are copied to more read-resistant pages (e.g., the LSB page type in TLC chips) of target blocks. Once an RR procedure is completed and the block B is erased, the two-level counters in the block B are reset to zero.

4.4 Experimental Results

In order to evaluate the effectiveness of `redFTL+`, we have used a co-simulation environment, FlashBench++, for flash-based storage devices. FlashBench++, which is based on the existing unified development environment, FlashBench [23], was designed to seamlessly switch between two simulation modes, the speed mode and the accuracy mode, depending on an evaluation goal. In the speed mode, FlashBench++ runs fast on top of a functional NAND simulation model whose timing behavior is not accurate. When a high precision of timing accuracy is necessary, FlashBench++ runs in the accuracy mode on top of a timing accurate NAND emulation model although it runs slow. Since it takes a long time to activate RRs in the ac-

curacy mode, we accelerate our evaluation by running FlashBench++ in the speed mode until a large number of reads are performed. By using an execution snapshot function of FlashBench++, we then switch to the accuracy mode so that we can collect a detailed timing accurate evaluation data.

In our evaluation, each block was assumed to consist of 192 pages and the size of a page is set to 8 KB. The total number of blocks in NAND flash memory was set to seven times of the working set size of each benchmark trace. The latencies of read, program, and erase operations for a normal block were set to 100 μs, 1,600 μs, and 5 ms, respectively, reflecting recent TLC chips [19, 29]. we used TLC_{base} model as NAND read-disturbance model, which is most commonly used type of TLC. The maximum read count of a normal block is set to 40,000 based on the estimated of read disturbs in Section 3.1, and the proactive RR threshold value is set to 38,000, which is 95% of 40,000.

We used highly read-dominant benchmark traces whose characteristics are summarized in Table 6. The trace interval in Table 6 means the length of the time interval during which a corresponding trace was collected. In order to evaluate `redFTL+` under realistic conditions, we synthetically generated new traces which better reflect real-world read-disturb cases using traces in Table 6. Since the original benchmark traces listed in Table 6 were collected from HDD-based storage systems, they cannot accurately reflect the characteristics of recent SSD-based storage systems. For example, recent SSDs support at least several thousand times higher IOPS over HDDs [30]. Furthermore, thanks to high-speed network connections, such SSDs can support much higher number of concurrent clients.

To reflect these changes in storage systems, we increases the number of read requests in a new trace significantly. In order to concisely represent an increase in the number of reads in SSD-based storage systems over that of HDD-based storage systems, we defined a read amplification factor (RAF) for each trace that indicates how many times the number of read requests in the original trace should be increased for better representing read workload of SSD-based modern storage systems. Depending on a RAF value N of a trace B, we allocated the same trace B to N different I/O generation threads, and made them to generate I/O requests of the trace B. In order to mimic real-world users with various temporal I/O access patterns, however, each I/O thread varied randomly idle intervals between successive I/O requests. In Table 6, the RAF column represents this RAF for each trace. RAF values were set differently according to the characteristics of original traces.

Figure 21 shows how the execution times for data migrations and block erasures during RR change under different read-disturb management techniques. In addition to `baseline` and `redFTL+`, we evaluated two more techniques, `RRB` and `redFTL++`. `RRB` exploits only read-resistant blocks without considering the effect of RRPs. `RedFTL++` maintains exact read counts (but expensive) for all LBAs instead of our proposed two-level coun-

Benchmark	Description	Read (%)	Trace interval	RAF
ads [18]	A display ads platform	96	1 day	50
tpc-h [31]	Accesses to a database	92	10 hours	200
websearch [22]	A search engine	100	4 days	100
multi [31]	Cscope and gcc	98	40 mins	50
tpc-e [32]	Accesses to a database	95	10 hours	30
tatp [33]	Telecom application process	83	21 hours	3

Table 6: A summary of benchmark traces.

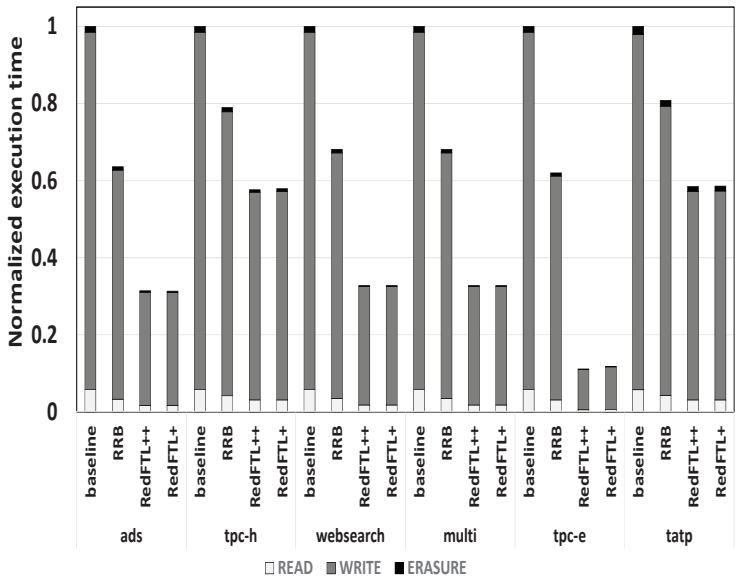


Figure 21: The normalized execution times for read reclaims over different techniques.

ters. In Figure 21, the x-axis indicates benchmark traces and applied techniques, and the y-axis represents the normalized execution time for read reclaims, which is normalized with that of `baseline`. `RedFTL+` and `RRB` decreased the overall execution times for RR, on average, by 62% and 30% over the `baseline`, respectively. Since `RRB` uses read-resistant RRBs with higher maximum read count over normal blocks, the number of RR activations is significantly decreased, thus reducing the RR execution time. `RedFTL+` further reduced the RR execution time overhead by distributing read-hot data in disturbed blocks to two RRBs. Moreover, the moved read-hot data are located in RRPs of the RRBs. In `redFTL++`, however, there were no significant reductions in RR execution times over `redFTL+` al-

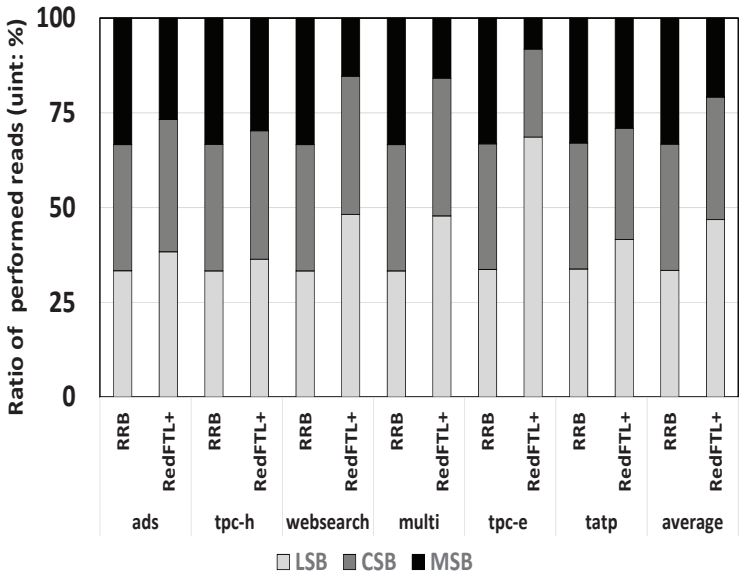


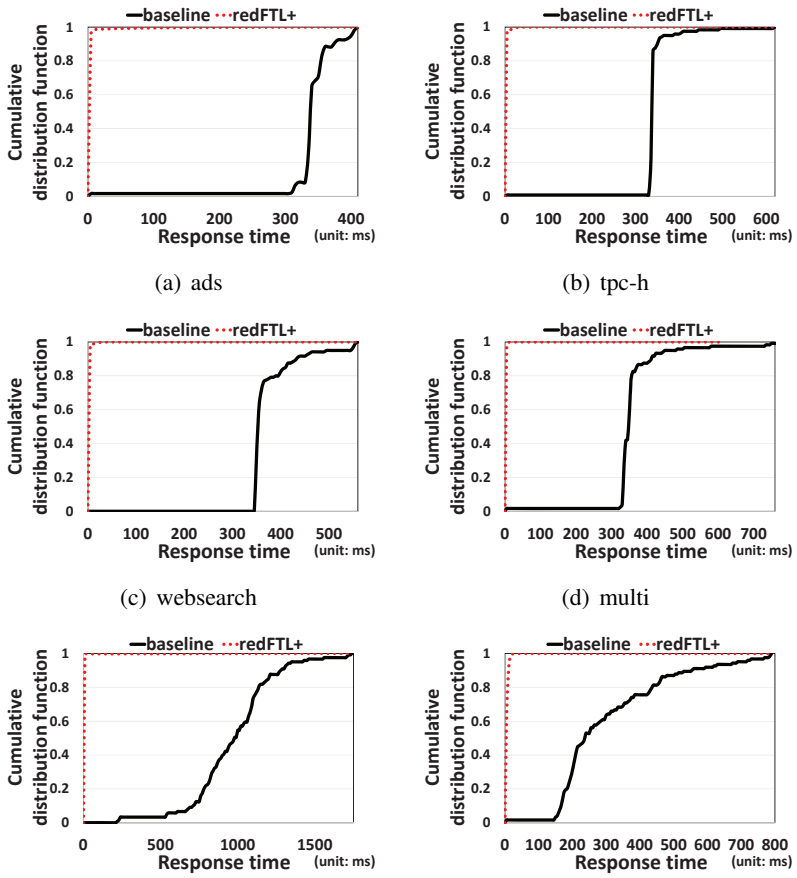
Figure 22: A breakdown of page type usages in `redFTL+`.

though `redFTL++` required four times more memory. This is because once data are classified as read-hot, it remains as read-hot for a long interval, thus making `redFTL+`'s early checks of data's read-hotness sufficient for read-hot data separation.

Figure 22 shows how effective our proposed page relocation policy is during an RR procedure. The x-axis represents benchmark traces and applied techniques, and the y-axis indicates the ratio of performed reads for each page type. As shown in Figure 22, LSB and CSB pages, which are more read resistant than an MSB page, were read more frequently than MSB pages in `redFTL+`. Since the page allocation policy in `redFTL+` allocates fast LSB pages and slow MSB pages to read-hot and read-cold data during an RR procedure, respectively, the percentage of LSB-page reads is in-

creased, on average, by about 40% over the RRB technique By making more reads from the read-resistant pages, redFTL+ can reduce RR activations.

In order to investigate the effect of our proactive read reclaim approach, we observed the response time fluctuations when RR procedures are activated (using the accuracy mode in FlashBench++). Figure 23 shows the cumulative distribution functions (CDFs) of read response times while 120 RRs are activated for six benchmarks. As shown in Figure 23, the read response times of most reads are less than 5 ms in redFTL+. On average, redFTL+ reduced the read response time during 120 RR activations by 150 times over the baseline technique. RedFTL+ spreads performance overhead for a long time period, thus dramatically reducing the read response time. On the other hand, the baseline technique does not activate an RR until the number of reads reaches the maximum read threshold. When the RR is activated, a large number of valid pages in the disturbed block are simultaneously moved. As a result, the response times of most reads in the baseline technique are over 350 ms.



(a) ads (b) tpc-h

(c) websearch (d) multi

(e) tpc-e (f) tatp

Figure 23: CDFs of read response times under `redFTL+` and `baseline`.

Chapter 5

A Single-Layered Data Separation Technique

5.1 Motivations

5.1.1 Frequency-Based Data Separation

One of the most widely used data separation heuristics is to separate data by their update frequency. The basic idea of this data separator is to classify data based on their write temporal locality, and treat data with different temporal locality in a different fashion [10, 15]. This technique assumes that data with high write temporal locality are likely to be invalidated soon by successive update requests, thus the number of dead blocks can increase if data with high locality are gathered in the same block. For example, the simplest version of this locality-based data separator divides data into two groups, *hot* data and *cold* data, based on the number of updates in a given time window. When data are written to NAND flash memory either by a write request or by a move request from a garbage collector, hot data and cold data are written in separate hot blocks and cold blocks, respectively. By gathering hot data into separate hot blocks, they are more likely to be dead blocks soon.

Although locality-based heuristics work reasonably well, it is not clear

if knowing the relative frequency of block updates is sufficient or not in minimizing the garbage collection overhead. In order to better understand the performance of the existing data separator heuristic, we first introduce predictor on *future* update times, which has complete knowledge on future data update times. Using an FTL based on the oracle predictor as an *off-line* optimal FTL, we evaluate the performance of the existing data separation heuristic (based on the write temporal locality). The oracle predictor on data update times is denoted as `ORA` in this dissertation. The off-line optimal FTL is based on the idea that the garbage collection overhead is minimized when a garbage collector always selects dead blocks as victim blocks. The most obvious way to make more dead blocks is to predict future data update times and gather data with similar update times in the same blocks. Since the data in such blocks will be invalidated almost simultaneously, these blocks will become dead blocks quickly once the first data in the block is overwritten. Therefore, such blocks will not incur unnecessary page migration overhead from moving valid pages in victim blocks to new blocks during garbage collection. Clearly, the `ORA` is not implementable in practice because it requires the perfect information on *future* block update times. In this dissertation, we use the garbage collection overhead of an FTL based on `ORA` as an upper bound in evaluating garbage collection heuristics.

From the comparative study using `ORA`, we have observed that gathering data with similar future update times to the same blocks, not data with high update frequencies, is a more important factor in minimizing garbage collection overhead. We have found that data with similar update frequencies were not necessarily updated at similar times. For example, there is no

clear correlation on their update times among hot data if they were classified as hot data at different times. If several hot data groups with different locality are stored in the same block, the probability that all data in that block are updated together is small because data with different locality have different update times. One of the main reasons of the poor performance of existing garbage collection heuristics can be attributed to the fact that they ignore data update times in devising their data separation techniques.

Based on the observations, we propose a novel data separation technique which predicts data update times by exploiting program contexts [34, 31] as hints. The proposed technique estimates what data will be updated together based on the data update history of the program context *PC*. Once data with similar future update times are predicted, the data are allocated into the same block by an FTL using the proposed technique, both when a write request is processed and valid data in a victim block are moved during garbage collection. In this dissertation, we assume that there is an appropriate interface between an operating system and an FTL to pass the program context information from the operating system to the FTL.

Conceptually, a program context represents one execution phase of a program. Since the program behaves similarly when the same phase is executed, program contexts can be used in predicting future block update patterns when a particular program context is identified with its previous block update history. For example, if a program context *PC* generates update requests R_1, R_2, and R_3, it is very likely that the same program context *PC* will generate the same update requests R_1, R_2, and R_3 again when the program context *PC* is re-executed. We can also group several program contexts into

a set of inter-related program contexts where each member program context follows similar update request patterns with other member program contexts. Using the program context-based predictions, the proposed technique identifies a group of data that will be updated at similar times.

In order to evaluate the proposed data separation technique, we have experimented using write traces collected from several programs. The experimental results show that the proposed technique reduces the total execution time of garbage collection on average by 58% compared to a hash-based locality separation technique [11].

5.1.2 Garbage Collection Using ORA

If a garbage collector can choose a dead block as a victim block whenever a garbage collector is invoked, the total execution time of the garbage collection process is reduced to the total execution time of erase operations performed during garbage collection. Although it is not trivial to devise such an optimal garbage collector (even if the complete details of write requests are known *a priori*), if we know future block update times in advance, we can design a very efficient garbage collector. In this section, we describe a garbage collection process based on ORA.

When a write request arrives, an FTL consults the ORA to get the future update time of the written data. Based on the future update time, the FTL allocates the requested data to the block where data with similar update times were already stored. During garbage collection, a garbage collector moves valid data in a victim block to the block with similar update times, using ORA. Although it is impossible to implement an on-line version of ORA in

practice (because we cannot build such an oracle predictor on future block update times), ORA can be built off-line if we have a complete trace of write requests including their request times. In this dissertation, we implement an off-line version of ORA, which will be used as a data separator, in evaluating other data separation heuristics.

5.1.3 Evaluation of Existing Locality-based Heuristic

A locality-based data separator has been widely used in various FTLs. In particular, many researchers have proposed different data separation techniques that aim to increase the accuracy of data locality classification. For example, recently proposed techniques include 2-level LRU-based heuristic [10], hash table-based heuristic [11], and request size-based approach [15]. Since the hash table-based heuristic can accurately classify data with a small memory footprint and low time complexity, we use the hash table-based heuristic as a representative locality-based data separator. The hash table-based data separation heuristic is denoted as `HASH` in this paper. We also assume that a page-level mapping FTL is used in this paper. To evaluate different data separation techniques under the equal conditions, we use a garbage collector (except for a data separation technique) in the same page-level FTL. We use an FTL based on the cost-age-time heuristic [35] which takes account of the cleaning cost, erased counts, and the time elapsed (since the last modification) in selecting a victim block. (Unless confusion arises, we use `ORA` and `HASH` to indicate both data separation techniques and FTLs based them.)

Compared to the ORA algorithm, however, the `HASH` cannot achieve a

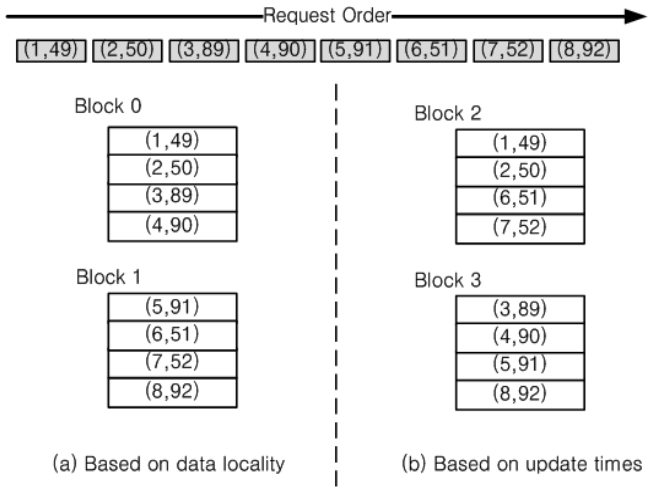


Figure 24: A comparison of data allocation using (a) HASH and (b) ORA.

high performance even if HASH can perfectly identify data update locality. For example, although hot data are clustered in the same block, if the update times are different among the hot data, the block may be half-dead, that is, some hot data in the block remain valid while other data in the same block are invalid. Such half-dead blocks significantly increase the amount of copied data during garbage collection.

Figures 24 and 25 show examples which illustrate a poor performance of HASH over ORA. In Figures 24 and 25, we assume that eight write requests, which are shown on top of Figure 24 as eight rectangles. Since we assume a page-level mapping FTL, each rectangle represents a page write request. A tuple (i, t_u) in a rectangle represents the *i-th* request with the next update time t_u. We further assume that data written by the write requests in the examples have been already classified as hot data. In Figure 24(a), since all the data requests were classified as hot data by data locality, an

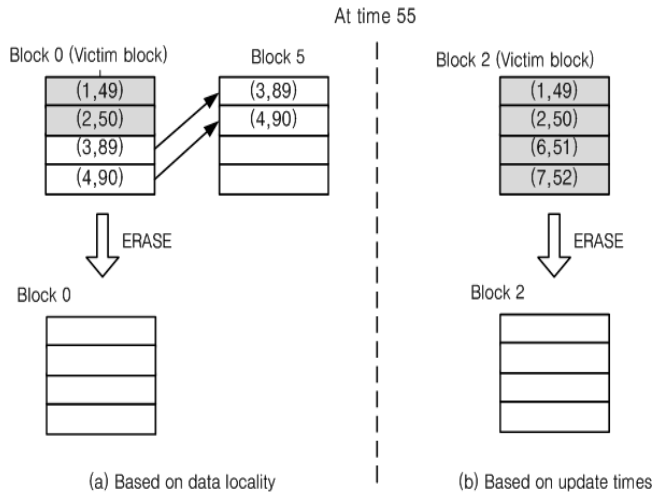
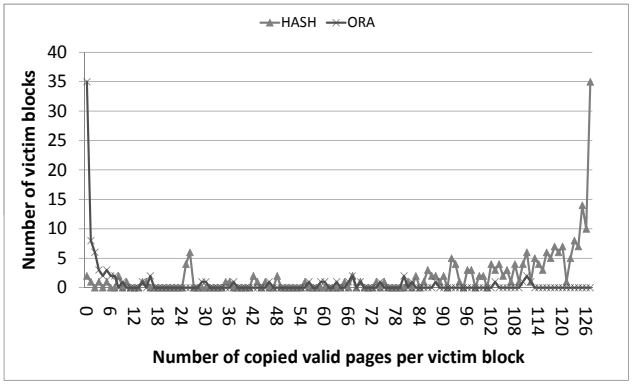


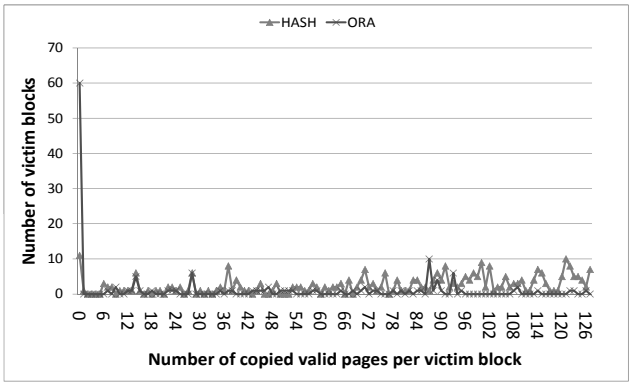
Figure 25: A snapshot comparison of garbage collection using (a) HASH and (b) ORA.

FTL writes them to the same block according to the request order. On the other hand, in Figure 24(b), an FTL using ORA allocates the requested data to Blocks 2 and 3 grouping requested page writes according to their future update times. Figure 25 shows snapshots of garbage collection using HASH and ORA. Assuming that garbage collection is invoked at time 55, in HASH as shown in Figure 25(a), because bottom two pages in Block 0 remain valid at time 55, although all pages in Block 0 are full of hot data, the valid pages are moved to Block 5, which is a newly allocated block. Since these pages will not be invalidated until their respective update times, time 89 and time 90, they may be copied to other victim blocks several times whenever these pages belong to a victim block. In ORA as shown in Figure 25(b), Block 2 has been changed to a dead block at time 52, hence Block 2 is reclaimed with only one erase operation.

In order to investigate the efficiency of a locality-based data separator in reducing garbage collection overhead, we have compared ORA and HASH using several benchmark programs. (For a more detailed description of the experimental setup, refer to Section 5.4.) In HASH, only 66.3% of hot data pages in hot blocks were updated in similar times, leaving about 34% of hot data pages still valid after other pages got invalidated. These valid pages produce a large number of half-dead blocks which can increase the garbage collection overhead when selected as victim blocks. Figure 26 shows distributions of the number of copied pages per victim block when HASH and ORA are applied in two programs used in experiments. The X-axis and the Y-axis denote the numbers of victim blocks and copied valid pages per victim block, respectively. In Figures 26(a) and 26(b), HASH tends to copy more valid pages per victim block than ORA. Since these copied valid pages invoke more garbage collection processes, HASH may suffer poor performance of garbage collection. In the observations, HASH increases the total number of copied valid pages during garbage collection by 44 times over ORA. These results strongly suggest that a better data separation technique can be developed if data with similar future update times can be found.



(a) gcc



(b) cscope

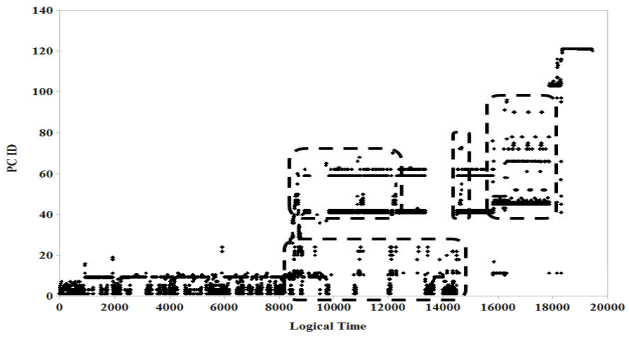
Figure 26: Distributions of the number of copied pages per victim block.

5.2 Correlation between Program Contexts and Updates

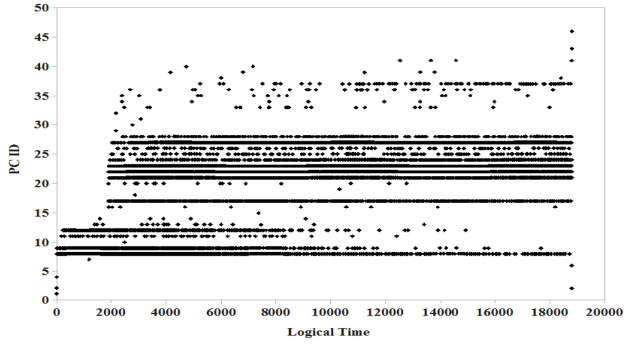
Although taking account of update times of data is useful in reducing garbage collection overhead, it is impossible to know accurate update times before actual update requests occur. In this dissertation, we indirectly

identify data which are updated together in similar time periods by exploiting typical update behavior of a program. We use program contexts to keep track of program's write/update behavior. Since a program context represents a program phase and the same phase is likely to be executed multiple times, program contexts have been used in predicting future behavior of programs (e.g., [36, 37]). Since a program phase consists of consecutive instructions executed and the instructions can be specified with instruction counters, instruction counters can be used to distinguish different program contexts.

We use the *signature program counter* [31] to identify significant program contexts, which are distinguished by different program context (*PC*) IDs. Each *PC* is identified by summing instruction counter *ic* values of each execution path of function calls that lead to system calls which cause write requests. For instance, a program can issue a write request through system functions such as write() and writev() in the Linux kernel. If the functions a(), b(), and c() were called in sequence before reaching the system functions, the *ic* values of these three functions can be obtained by a stack frame traversal when the write request is processed. The signature program counter for this write request is computed by summing those three *ic* values. Although this traversal is carried out whenever a write request occurs, it is negligible in processing a write request because the additional overhead of getting a signature program counter is 0.19 microseconds on a 2 Ghz Intel Pentium personal computer with 2 GB RAM [31], while performing a write operation in NAND flash memory requires several hundred microseconds. We assume that *PC* IDs are passed to an FTL through APIs between a file



(a) cscope+gcc



(b) tpc-r

Figure 27: Distributions of program contexts in some benchmark programs.

system and an FTL.

In order to evaluate the feasibility of using *PC*s in predicting data update patterns, we investigated the relationship between updated data and their corresponding *PC*s (which have generated the updated data). Figure 27 shows distributions of *PC*s which have produced updated data requests in benchmark programs. A horizontal axis indicates the logical times which increase by one whenever data are updated. A vertical axis denotes distinct *PC*s which have generated the updated data requests at each logical time. For example, if data are written from a *PC* at logical time t, and the data are

updated by new write requests from the *PC* at logical time $t + \alpha$, where α is a positive number, the *PC* is shown at logical time $t + \alpha$.

From the analysis, we have observed two key *PC* characteristics that can be used in designing our heuristic on data update times. First, a small number of *PC*s dominate. These dominating *PC*s generate repeatedly a large number of write requests, and the data from the dominating *PC*s are updated. Figures 27(a) and 27(b) show such cases; a few *PC*s produce repeatedly most of write requests, and the written data are updated in similar times. In the benchmarks used in experiments, top five dominating *PC*s of each benchmark generate about 76% of updated data requests. Since data from these *PC*s tend to be updated consecutively, if an FTL allocates these data to the same blocks, it is likely that data from these *PC*s in the same blocks will be updated together. Second, data from non-dominating *PC*s are often highly correlated with dominating *PC*s, because consecutive update requests are often generated from several *PC*s. For example, in the dotted boxes in Figure 27(a), data from non-dominating *PC*s which appeared infrequently are updated together with data from dominating *PC*s in similar time periods. In the observations, 63% of *PC*s which generate updated data requests are involved in sequential update patterns. If an FTL can find these *PC*s by checking update access patterns of *PC*s, the FTL can gather data from these *PC*s into the same blocks. The observations suggest that *PC*s are closely related with updated data, which means that *PC*s can be used as important hints in estimating data updated in a similar time.

5.3 PDS: Program Context-Aware Data Separation Technique

We designed a program context-aware data separation technique based on the correlation between *PC*s and update requests. Figure 28 briefly explains how an FTL based on the proposed technique works. The technique is used both when a write request arrives from a host system and valid pages in a selected victim block are written to a new block during garbage collection. When a write request arrives in both situations, the proposed technique checks the *PC* which has generated the current write request. If the proposed data separation technique determines that data request from the *PC* is likely to be updated later, the proposed technique predicts what data will be updated together with the requested data, and provides the update information

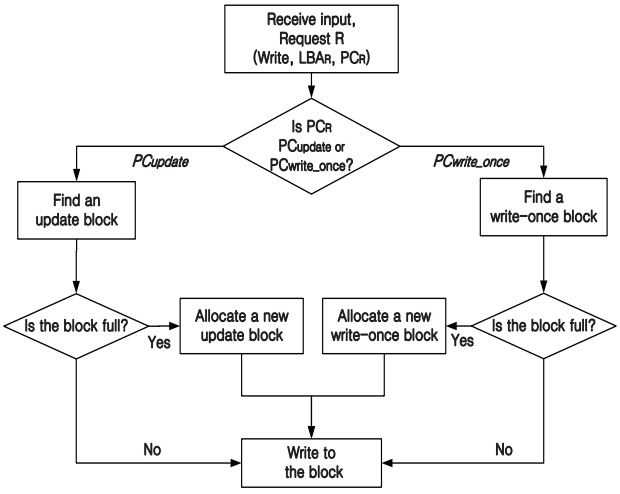


Figure 28: The flowchart of data allocation in an FTL using the proposed technique.

to the FTL. If a *PC* generates a write request *R* and the data written by *R* are updated later, we call such a *PC* the *update PC*, denoted as PC_{update}. To allocate a page to the requested data, the FTL finds a block storing data that will be updated together with the requested data. A block is called an *update block* if its data are updated later. If the block is write-once, we call such a block *write-once block*. For a write request from an update PC, we store the requested data to an update block with similar estimated update times. If the proposed technique predicts that the requested data will not be updated based on the *PC* which has generated the request, the data are written to a write-once block. If a *PC* generates write requests, and if the written data are not updated later, such a *PC* is called *write-once PC*, denoted as PC_{write_once}.

Figure 29 shows an example of data allocation based on the proposed technique using the same example in Figure 24. Each write request tuple of Figure 24 is modified to include the *PC* identifier id_{PC} so that each write request (i, id_{PC}) indicates that the *i-th* write request has been generated from the *PC* id_{PC}. The id_{PC} of each request is sent to an FTL from a host system whenever a write request occurs. In this example, id_{PC} was computed to 5 by summing *ic*s of functions a() and b(), and it is delivered to an FTL. Assume that first seven write requests have been processed. When a write request arrives, the FTL searches the program context information table with the requested id_{PC}, 5, to predict whether the requested data will be updated or not. The program context information table stores update information of previously identified PC_{update}s to find data with similar estimated update times. A row in the table stores a tuple $(id_{PC}, id_{UG}, N_{update})$, which means that *PC* id_{PC} generated N_{update} write requests which were updated in similar

time periods together with data from the PC_{update}s in an update group UG, whose identifier is id_{UG}. An update group UG is a group of PC_{update}s that are expected to be updated in a similar time period. (We will describe how a tuple is created in the table in the next subsection.) Since the PC whose id_{PC} is 5 exists in the table, the proposed technique decides that data written by this request will be updated together with data from PC_{update}s in the UG whose identifier id_{UG} is 1, thus the FTL finds an update block for this UG to store the requested data. To keep track of update blocks allocated to UGs, an update block information table records an update block number which stores data from PC_{update}s in an UG. Since many update blocks can be used by an UG, the update block number used most recently is stored with id_{UG}. Since this table indicates that data from the PC_{update}s in the UG whose id_{UG} is 1 are stored in Block 4, the requested data are written to Block 4. Prior write requests have been processed in the same way except for the fifth write request. Since the id_{PC} 3 does not exist in the program context information table, the proposed technique regarded the PC whose id_{PC} is 3 as a write-once PC, thus the FTL wrote the data to Block 6, which is a write-once block.

In this way, the FTL gathers data from PC_{update}s in the same UG to the same update blocks. However, if an update block is selected as a victim block before all the pages in the block are invalidated, many read and write operations may occur during garbage collection. To avoid this situation, the FTL gives the lowest priority to update blocks when a victim block is chosen. By allowing enough time to update blocks, data in the update block are very likely to be invalidated before it is selected as a victim block.

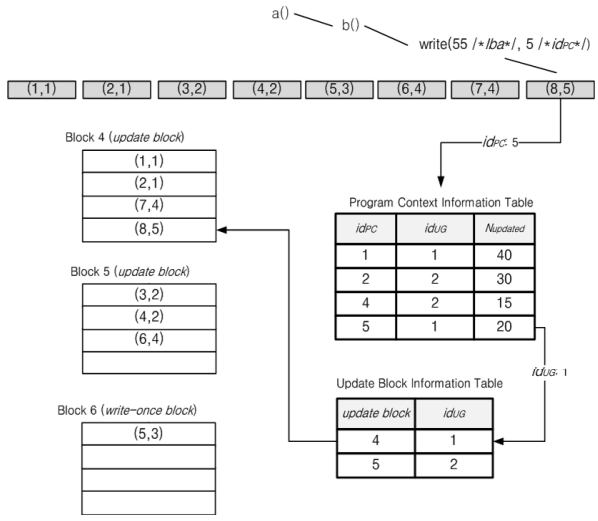


Figure 29: An example of data allocation using the proposed data separation technique.

Based on the relationship between *PC*s and updated data, the proposed technique groups *PC*s which generate data with similar estimated update times into the same UGs. As mentioned in Section 5.2, since a small number of *PC*s generate repeatedly many write requests which tend to be updated consecutively, the technique finds a *dominating PC*, PC_d, and inserts the PC_d to an UG. Moreover, the technique finds a *sequential PC*, PC_{seq}, which repeatedly generates updated data requests forming sequential update access patterns with data from other PC_ds, and inserts the PC_{seq} to the UG including the PC_ds.

Algorithm1 describes how the proposed technique groups update *PC*s into an UG. This algorithm takes as input *PC* ID id_{PC} and logical block address lba, and returns UG ID id_{UG}. Note that the id_{PC} in this algorithm is not the identifier of the *PC* which generates current data request, but the

Algorithm 1 Program Context Grouping Algorithm

Input : id_{PC}, lba

Output : id_{UG}

```
 1: I_pc ← program_context_information_table(id_PC)
 2: if I_pc = NULL then
 3:     I_PC.id_PC ← id_PC
 4:     I_PC.id_UG ← NULL
 5:     I_PC.N_update ← 1
 6:     insert_tuple(I_PC)
 7: else
 8:     I_pc.N_update ← I_pc.N_update + 1
 9:     if I_pc.id_UG = NULL then
10:         if (I_pc.N_update ≥ threshold) then
11:             N_UG ← N_UG + 1
12:             I_pc.id_UG ← N_UG
13:         else
14:             if ((prev_lba + size_of_page) = lba)
                && (prev_UG ≠ NULL) then
15:                 I_pc.id_UG ← prev_UG
16:             end if
17:         end if
18:     end if
19: end if
20: prev_lba ← lba
21: prev_UG ← I_pc.id_UG
22: return I_pc.id_UG
```

identifier of the *PC* which has generated the data updated by the current write request. Since the objective of this algorithm is to find update *PC*s which generate data expected to be updated together and to group into an UG, only the *PC*s of the updated data are considered. Whenever an update write request occurs, this algorithm searches the program context information table with the id_{PC} to check whether the *PC* has been determined as an update *PC* or not (line 1). If there is no update information of the *PC*, which is denoted as I_{PC}, a new tuple is created and its id_{PC}, id_{UG}, and N_{update} are set (lines 3-5). Since the data from the *PC* is updated for the first time, UG and N_{update} are set to NULL and 1, respectively, and this tuple is inserted

into the table (line 6). The size of the memory required for keeping the program context information table depends on how many *PC*s in a program are related to update requests. For the applications used in the experiments, update requests are generated from 48 to 89 *PC*s.

If the I_{PC} for the *PC* exists, the number of updates increments by one (line 8), because data from the *PC* are updated. To find out whether the *PC* has been included in an UG or not, this algorithm checks the UG information of the *PC* (line 9). If an UG including the *PC* does not exist, this algorithm decides either to create a new UG for the *PC* or insert the *PC* to one of existing UGs based on N_{update} and update request pattern. If N_{update} of the *PC* is greater than a predefined threshold value, this algorithm regards the *PC* as a dominating *PC*, thus making a new UG and including the current *PC* to the new UG (lines 11-12). If N_{update} is less than the predefined threshold value, the proposed technique checks the current *PC* is a sequential *PC* or not. If the current update request forms sequential update patterns with data updated previously, and if the previous data are from an update *PC* included in an UG, the proposed technique inserts the *PC* into the same UG (lines 14-15). For the determination, the *lba* and id_{UG} previously processed in this algorithm are stored (lines 20-21). By returning the id_{UG} to an FTL, proper blocks can be allocated to write requests.

5.4 Experimental Results

To evaluate the proposed technique, we used a trace-driven FTL simulator. The FTL in the simulator triggers garbage collection if the total num-

ber of remaining blocks in NAND flash memory is less than 5% of the total number of blocks in NAND flash memory, and the garbage collection process is finished if 15% of the total number of blocks in NAND flash memory are reclaimed. Since garbage collection does not occur if there is enough free space to write new data, we filled 90% of the NAND flash memory space with meaningless values, before each experiment is performed. Respective latencies of read, write, and erase operation are 25μs, 200μs, and 1.2ms. `HASH` and `ORA` are used for comparisons, and the technique is denoted as *PC-aware* in experiments.

In order to generate input traces for the simulator, we used four programs and two application sets as shown in Table 7. Although most applications used in the experiments do not generate many update write requests, they are enough to evaluate garbage collection overhead because they trigger a large number of garbage collection processes in NAND flash memory which is almost fully filled with data. For example, gcc is known as a CPU-bound application. However, it creates many object files and a kernel image file while it compiles Linux kernel source in the experiment.In the case of tpc-h, it is a read-dominant workload with a small number of update requests, which come from concurrent data modifications. Nevertheless, it causes a large number of garbage collection processes because we start with almost full NAND flash memory.In the case of viewperf, since it does not have a lot of update requests, we performed the experiments with multi2 combining viewperf with cscope and gcc.

Figure 30 shows the total execution time of garbage collection where each data separation technique is applied. The X-axis and the Y-axis denote

Benchmarks	Scenario	N_{write}	N_{update}
cscope	Linux source code examination	17,575	15,398
gcc	Building Linux Kernel	10,394	3,840
viewperf	Performance measurement	7,003	119
tpc-h	Accesses to database	23,522	20,910
tpc-r	Accesses to database	21,897	18,803
multi1	cscope+gcc	28,400	19,428
multi2	cscope+gcc+viewperf	35,719	20,106

N_{write}: the number of write requests (unit : page)
N_{update}: the number of update write requests (unit: page)

Table 7: Summary of various benchmarks.

data separation techniques and the total execution time of garbage collection, respectively. The results show that the proposed technique reduces the total execution time of garbage collection on average 58% over HASH. Since HASH determines data locality based on data update frequency in a given time window, if data are not updated frequently in the time window, HASH does not work well. In the cases of tpc-h and tpc-r, since they have relatively random write patterns over other programs, it is unlikely that data are written again and again in the same time window.

In order to understand why the proposed technique outperforms HASH and how the technique is compared to ORA, we compared the number of dead blocks, the average number of copied pages per victim block, and the number of erased blocks. Table 8 shows the total number N_{dead} of dead blocks generated, the average number N_{avg_cped} of copied pages per victim block, and the total number N_{erase} of erased blocks when each data separation technique is applied.Compared to HASH, the proposed technique generates more dead blocks, and reduces valid page copies per victim block

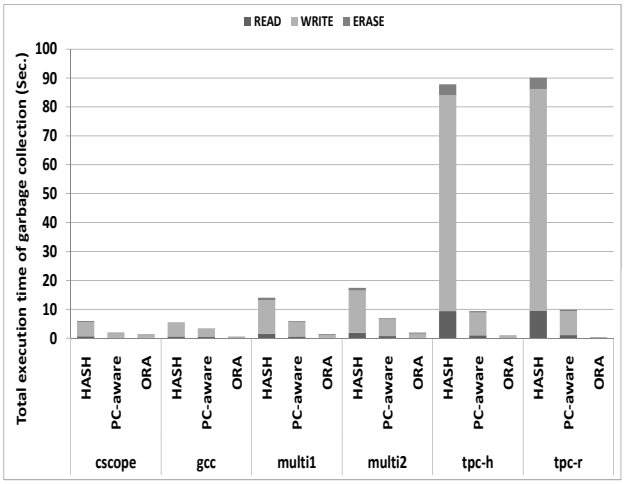


Figure 30: Total execution times of garbage collection in various traces.

by 25%. Moreover, HASH erases on average about five times more blocks than the proposed technique. These results clearly mean that predicting data to be updated together with program context hints is effective in reducing garbage collection overhead. On the other hand, ORA produces about four times more dead blocks than other techniques. Besides the number of generated dead blocks, the proposed technique copies about three times more valid pages per victim block than ORA. ORA erases on average 27% less blocks than the proposed technique. Although ORA works based on off-line analysis, these results strongly suggest that there may be a room for improving the proposed technique further.

Figure 31 shows the normalized garbage collection overhead under different threshold values of Algorithm 1. The X-axis and the Y-axis denote various thresholds in each program and normalized total execution time of garbage collection, respectively. The results of each program are normalized

Benchmarks	Separator	N_{dead}	N_{avg_cped}	N_{erase}
cscope	HASH	1.0	125.4	318
	PC-aware	42.3	60.6	189
	ORA	61.0	35.8	199
gcc	HASH	0.0	108.2	225
	PC-aware	0.2	105.6	154
	ORA	8.0	25.1	94
multi1	HASH	0.0	110.5	655
	PC-aware	39.8	88.7	303
	ORA	77.0	24.6	280
multi2	HASH	0.0	109.3	793
	PC-aware	34.7	89.1	345
	ORA	83.0	30.4	325
tpc-h	HASH	0.0	123.4	3178
	PC-aware	0.3	90.0	385
	ORA	144.0	26.9	250
tpc-r	HASH	0.0	123.8	3235
	PC-aware	0.2	93.6	423
	ORA	145.0	26.2	165
arithmetic average	HASH	0.2	116.8	1400.7
	PC-aware	19.6	87.9	299.8
	ORA	86.3	28.2	218.8

Table 8: A comparison of the proposed technique over `HASH` and `ORA` for garbage collection overhead.

to the worst case of each program. As shown in Figure 31, each program has different threshold values which cause the smallest execution time of garbage collection. Since a combination of *PC*s included in each UG can be changed by the threshold, the influence of the threshold on garbage collection overhead is noticeable. If the threshold is small, most of *PC*s which have produced updated data may be classified as update *PC*s, thus more UGs are created. If the *PC*s classified as update *PC*s generate many write requests, and the written data are updated later, the proposed technique works fine.

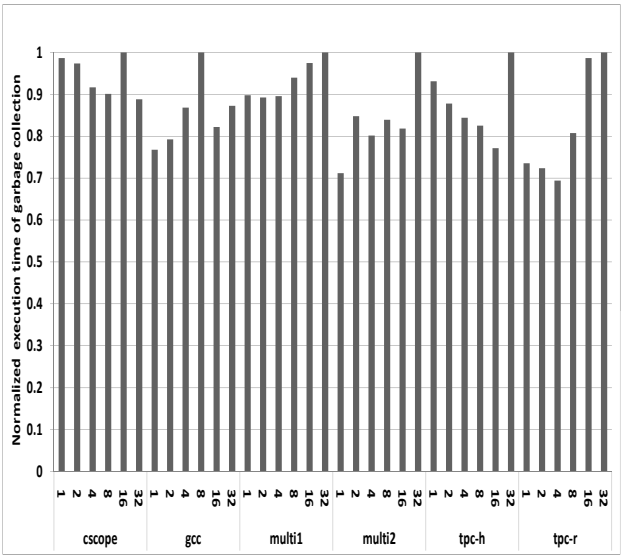


Figure 31: Normalized execution times of garbage collection under different threshold values.

On the contrary, if data from the *PC*s are not updated later, the data cause more extra copies and trigger more garbage collection. In the case of gcc, since *PC*s classified early as update *PC*s generate many write requests, and most of the data are updated later, gcc has the minimal total execution time when a threshold is 1. A large threshold may suppress creating UGs, because PC_{update}s which issue a lot of updated data requests can be inserted to an UG. In the case of tpc-h, by preventing *PC*s with a small number of updated write requests from being classified as update *PC*s, the total execution time of garbage collection is minimal when a threshold is 16.

Chapter 6

An Integrated Data Separation Technique

6.1 Limitations of Single-Layered Program Context-Aware Data Separation Technique

PDS can associate I/O behavior with program contexts, but potential exists for further reduction of the garbage collection overhead. First, PDS treats all data associated the same *PC* even if the data have different update patterns. A single program context can correspond to a large number of write requests with a similar pattern of updates, but the data associated with that program context are unlikely to be updated in exactly the same way. For example, if only some items of data are updated, and the rest are written once, the efficiency of garbage collection will be reduced. This problem can only be overcome by maintaining an update history of individual data items, allowing frequently updated data to be handled in different way to data that are written once.

Another issue with PDS is that it requires a large number of active blocks, thus triggering further garbage collection. Each dominating PC requires at least one update block and thus it is disadvantageous to reduce the number of active blocks. PDS classifies a *PC* as a dominating *PC* when the

accumulated number of data updates reaches a certain threshold, and it is likely that a large number of *PC*s will be classified as dominating *PC*s as a program is executed. Moreover, since PDS allows an update block to be used by a dominating *PC*, several additional active blocks are required when a lot of data corresponding to multiple dominating *PC*s are updated. The resulting increase in the number of update blocks can cause frequent garbage collection if there are not enough free blocks in the SSD. If a rigid criterion on update counts for classifying a *PC* as dominating *PC* and the data corresponding to multiple dominating *PC*s are stored together in an update block together, the number of required active blocks can be reduced.

6.2 IDS: Integrated Data Separation Technique

6.2.1 Overview

We propose an integrated data separation technique (IDS), which is an improvement on PDS, because it uses both program context information and block access history as data separation hints. IDS addresses the shortcomings in the way that IDS represents the patterns of updates to individual data items using a block access history additionally, IDS dynamically clusters data corresponding to multiple update *PC*s into one update group if their associated data are updated with similar patterns. By allowing multiple program contexts to be stored in an update block, IDS reduces the number of active blocks, and hence the necessity for garbage collection.

As shown in Figure 32, IDS operates in the similar way to PDS. However, IDS uses both program context hints and the update history of data

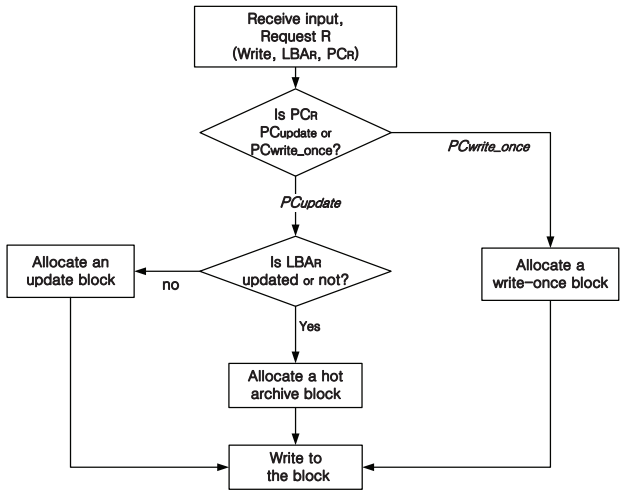


Figure 32: The flowchart of data allocation in an FTL using IDS.

to detect the hot data associated with an update program context. In order to respond to different update patterns exhibited by individual data items associated with the same program context, IDS stores the update history of data associated with an update *PC*. If data *D* have been updated since the data *D* were generated by an update *PC*, IDS classifies *D* as hot data, and it stores *D* in a special block called a *hot archive block*, which is likely to be a dead block. Figure 32 shows how IDS responds to when a write request. First, the *PC* associated with the data item is examined. If it is an update *PC*, the data are stored in the update block for the update group to which *PC* belongs. Then IDS decides whether the requested data are hot, based on its update history. If the LBA of the data is included in the update history, the requested data are stored in a hot archive block.

6.2.2 Determination of Update Program Context

IDS determines an update program context based on the consecutive updates of data. In order to maintain the consecutive updates, IDS keeps track of the associated *PC* of an previously updated data. For example, if data *D* are generated by program context PC_1 at logical update time *i*, which increases by one whenever an update happens, IDS compares the PC_1 with the corresponding program context of data updated at $i+1$. If both *PC*s are the same *PC*, a *consecutive update counter*, which represent the number of successive updates, increases by one. On the other hands, the consecutive update counter is set to zero. If the consecutive update counter is greater than a preset threshold, called *update threshold*, the program context is classified into an update *PC*. Since only program contexts of which data may be updated several times consecutively within a short time can be classified an update program context, the number of active blocks required can be reduced, thus reducing garbage collection activations.

6.2.3 Dynamic Clustering Program Contexts Based On Update Locality

IDS clusters multiple program contexts into one update group if their associated data are updated at a similar times. In order to detect similar update patterns, IDS periodically evaluates the update localities associated with program contexts, and then it rearranges update groups by adding and removing members. For example, if data items associated with *PC*s 1 and 2 are updated together, they are placed in the same update group. If the data

associated with *PC* 3 are also updated at the same time so the data associated with of *PC*s 1 and 2, *PC* 3 is added to the same update group.

In order to cluster program contexts with associated data that are updated together, IDS determines the *update time difference* between program contexts. IDS maintains a FIFO log in which the W most recent updates are recorded, together with the associated *PC*s. We can describe this log as a sequence of program contexts $U =< p_1, p_2, ..., p_w >$, where p_i is the ID of the *PC* associated with the i-th most recent update, within the window W. Let $\mathbb{PC}_U$ be the set of distinct program context IDs within the sequence U. Thus, in the example of Figure 33, $W = 7$, $U =< PC_1, PC_1, PC_2, PC_3, PC_1, PC_2 >$, in a given W, and there are three different program contexts, denoted as $\mathbb{PC}_U =< PC_1, PC_2, PC_3 >$.

To measure the similarity of update patterns between two *PC*s, IDS uses the *minimum update time difference*. Given a *PC* p_i in U and a recent *PC* PC_p, the minimum update time difference $ud_{p_i}(PC_p)$ is defined as follows:

$$\begin{aligned} & ud_{p_i}(PC_p) \text{ in } U_W \text{is defined as d iff} \\ & a)\ PC_p = p_j \text{in D} \quad \text{where } j \leq |D_W| \\ & b)\ d = \min(|i-j|) \end{aligned} \tag{6.1}$$

Thus, in the example of Figure 33, $ud_{p_1}(PC_2)$ and $ud_{p_1}(PC_3)$ are 2 and 3, respectively.

We can now define the temporal update locality between two program contexts PC_p and PC_q, denoted as $tl(PC_p, PC_q)$, as the average minimum

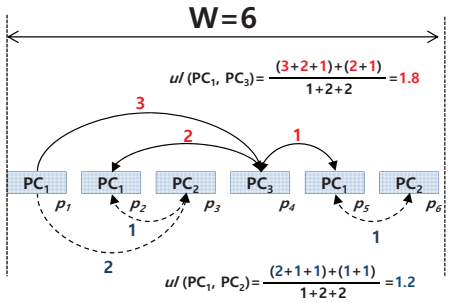


Figure 33: An example of calculating update locality between program contexts.

update time difference between PC_p and PC_q. If a program context corresponds to several data items, each of which may be updated several times within the window W, then $ud_{pc_i}(PC_p)$ and $ud_{pc_j}(PC_p)$ may be different, even if pc_i and pc_j are the same. For example, in Figure 33, $ud_{p_1}(PC_3)$ is 3, but $ud_{p_2}(PC_3)$ is 2, although both p_1 and p_2 are PC_1. To allow the update time differences of all data items to contribute to a measure of temporal locality of updates, the difference in temporal locality between PC_p and PC_q, denoted as $ul_{(PC_p,PC_q)}$, is obtained by averaging the update time differences of all the data items corresponding to PC_p and PC_q, as follows:

$$\overline{ud_{PC_q}(PC_p)} = \frac{\sum_{i \in \mathbb{U}_{PC_q}} (ud_{p_i}(PC_p))}{\left|\mathbb{U}_{PC_q}\right|} \tag{6.2}$$

, where $\mathbb{U}_{PC_q} = \{j \in \{1,2,...,\text{last}\} | p_j = PC_q \text{ in } U_W\}$. The more frequently that the data corresponding to program contexts PC_p and PC_q are updated, the smaller value of $\overline{ud_{PC_q}(PC_p)}$. The difference in temporal locality $ul_{(PC_p,PC_q)}$ can be expressed in terms of average update time differences as follows:

$$ul_{(PC_p,PC_q)} = \frac{(\overline{ud_{PC_q}(PC_p)}) \times \left|\mathbb{U}_{PC_q}\right| + (\overline{ud_{PC_p}(PC_q)}) \times \left|\mathbb{U}_{PC_p}\right|}{\left|\mathbb{U}_{PC_q}\right| + \left|\mathbb{U}_{PC_p}\right|} \quad (6.3)$$

.

The smaller the average update time difference between the data corresponding to PC_p and PC_q, the greater the difference in temporal update locality. Figure 33 shows an example in which the difference in temporal update locality is determined for these program contexts PC_1, PC_2, and PC_3. All of the update time differences between PC_1 and PC_2, and PC_1 and PC_3 are calculated, and they are used to obtain differences in temporal update locality using Equation 6.3. In this example, $ul_{(PC_1,PC_2)}$ and $ul_{(PC_1,PC_3)}$ are 1.2 and 1.8, respectively. These results mean that updates of the data items corresponding to PC_1 and to the updates corresponding to PC_2 take place at times that are relative similar, whereas the updates corresponding to PC_1 and PC_3 occur at more dissimilar times.

Once the differences in temporal update locality between *PC*s have been determined, `IDS` uses them to cluster program context by using the single-linkage clustering algorithm [38] which generates a hierarchy of clusters, as follows: to begin, the pair of program contexts (PC_p, PC_q) corresponding to the minimum value of $ul_{(PC_p,PC_q)}$ in $\mathbb{PC}_U$ is selected, and these *PC*s form the first cluster C_1. Then a second pair of program contexts (PC_s, PC_t), with the second smallest value of *ul* is selected, and these form a second cluster C_2. However, if either PC_s or PC_t is also in C_1, then both PC_s and PC_t are added to C_2, while remaining in C_1. Otherwise, C_1 and C_2 remain independent. In this way, program contexts are put into clusters in order of

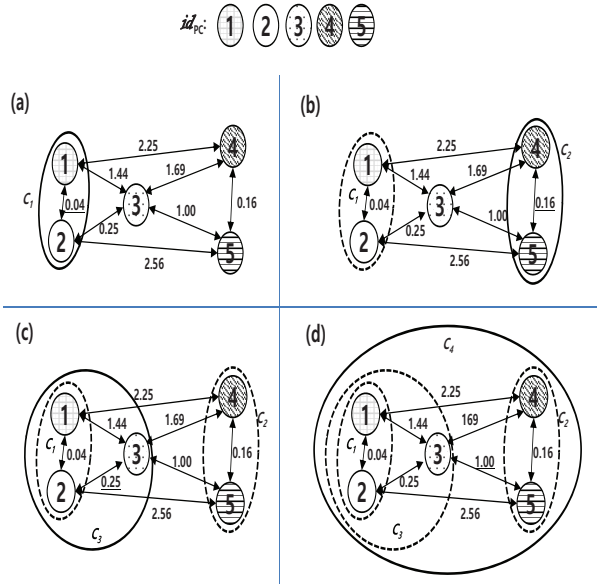


Figure 34: Clustering *PC*s based on temporal update locality: the circles are *PC*s, annotated with their IDs, and the links are annotated with temporal update values *ul*; at each step (a)-(d), the smallest values of *ul* is underlined.

increasing *ul*. Finally, all program contexts in the set $\mathbb{PC}_U$ are put into the top-level cluster $C_{|\mathbb{PC}_U|-1}$.

The order in which the clusters are created (i.e., the i in C_i) is related to the temporal update locality of the program contexts in each cluster: the smaller values of i corresponding to the higher temporal update locality. Figure 34 shows an example of clustering, starting with the *PC*s in the set $\mathbb{PC}_U = \{PC_1, PC_2, PC_3, PC_4, PC_5\}$. In Figure 34(a), PC_1 and PC_2 form cluster C_1 since their link has the smallest *ul* value. In step (b), we see that the next lowest value of *ul* is on the link between (PC_4, PC_5), which form an independent cluster C_2. In step (c), in Figure 34(c), PC_2 and PC_3 are selected; however PC_2 is already in C_1, so both $C1$ and P_3 are put into a new cluster

C_3. Finally, in step (d), C_2 and C_3 form the final cluster C_4. Even though program contexts PC_1, PC_2, and PC_3 are members of C_3, PC_1 and PC_2, which are members of cluster C_1, have the highest of three: PC_1 and PC_2, PC_2 and PC_3, and PC_3 and PC_1.

The clusters of *PC* based on temporal update locality are now used to make update groups, using Algorithm 2. To describe this algorithm, we need to introduce new notations. We call the cluster first joined by PC_i as its *original* cluster, written OC_{PC_i}. We also associate PC_i with a root cluster RC_{PC_i}, which is the earliest to be put into OC_{PC_i}. Program contexts in the root cluster have a higher temporal update locality than those in other clusters. In Figure 34(d), the original clusters of PC_3 and PC_4 (i.e., OC_{PC_3} and OC_{PC_4}) are C_3 and C_2, respectively. The root clusters of PC_3 and PC_4 (i.e., RC_{PC_3} and RC_{PC_4}) are C_1 and C_2, respectively.

The input to Algorithm 2 is a *PC* ID, written id_{PC}, and it returns ID of an update group ID, written $I_{PC}.id_{UG}$. Values of $I_{PC}.id_{PC}$ and $I_{PC}.id_{UG}$ are kept in the program-context information table. If program-context information of input id_{PC} is not present in the program-context information table, the I_{PC} including a program context ID and an update group ID are initialized to default values (lines 1-5).

Algorithm 2 preferentially puts program contexts that are in a root cluster into update groups. The program-context information table stores the current version of I_{PC}. If the program context is already in an update group, the algorithm decides whether to put that *PC* into an update group by examining order in which the original cluster and the root cluster were created, which provides a hint about update locality. If the root cluster and the origi-

Algorithm 2 Update grouping algorithm of IDS

Input : $id_{PC} \in \mathbb{PC}_U$

Output : id_{UG}

```
1:  I_PC ← program_context_information_table(id_PC)
2:  if I_PC = −1 then
3:      I_PC.id_PC ← id_PC
4:      I_PC.id_UG ← −1
5:      insert_tuple ( I_PC )
6:  else
7:      C_i = RC_PC
8:      C_j = OC_PC
9:      if ( j − i ) = 0 then
10:         N_UG ← |UG| + 1
11:         I_PC.id_UG ← N_UG
12:     else
13:         if j − i < threshold_cluster_level then
14:             I_pc.id_UG ← RC_PC.id_UG
15:         end if
16:     end if
17: end if
18: return I_pc.id_UG
```

nal cluster of the PC are the same (line 9), that *PC* is put into a new update group, together with another PC from the root cluster of the first *PC* (lines 10-11). Since the updates corresponding to program contexts in a root cluster have strong temporal locality, the *PC*s in the root cluster can be merged into an update group. Therefore, a new update group is created containing the program contexts in the root cluster. The ID of this new update group, N_{UG}, is stored in this new update group, together with the program context information I_{PC}. For example, in Figure 34(d), if PC_1 is the first input program context that is processed by this algorithm, PC_1 and PC_2 are grouped into a new update group UG_1, and the update group ID is returned with a value of 1.

A program context can be put into an existing update group if it shares

efficient temporal update locality with the existing members of the group. If the difference between the place of the original cluster and the root cluster in the order of cluster creation is less than a preset threshold $threshold_{cluster\_level}$, it suggests that the *PC*s in these clusters have a high temporal update locality, and that the data corresponding to this *PC* is likely to be updated at similar times to the data corresponding to the program contexts in the root cluster. Then this *PC* is put into the update group of its root cluster (lines 13-14). In our example in Figure 34(d), PC_3 is inserted into the update group UG_1 if $threshold_{cluster\_level}$ is set to 2. Finally, the *PC* information I_{PC} is inserted into the program-context information table so that it can be occurred by the FTL whenever a write request occurs (line 18). By dynamically adding and removing program contexts to and from update groups, `IDS` gathers program contexts with similar update locality into the same group.

6.2.4 Managing The Hot Data Associated with An Update Program Context

`IDS` maintains update histories information in the program-context information table. Since data associated with an update *PC* are likely to be updated than data associated with a write-once *PC*, `IDS` does not keep a full history of updates for all data items. Instead, it stores the LBAs of updated data as a binary tree. The address of the root node of the LBA tree is entered into the program-context information table . When a data item *D* associated with an update *PC* is updated, the LBA of *D* is stored in the binary tree which is linked with the entry of the program-context information

table where the update *PC* information is maintained. Since a node in an LBA tree can contain multiple LBAs, less memory is required for the LBA information. The update information for an data time is maintained until the associated program context is deleted from the program-context information table.

6.3 Experimental Results

We evaluated our proposed techniques using the FlashBench [23] simulation environment for flash-based storage devices. FlashBench [23] allows various SSD architectures to be modeled by setting parameters such as the number of buses, the number of chips per block, the number of blocks per chip, the number of pages per block, and the latencies of read, write, and erase operations. These device parameters were set based-on recent NAND chips [19, 29]. Table 9 summarizes our setup. In order to eliminate an influence of different NAND flash sizes from data separation evaluation, we set the number of blocks twice of the working set size of each benchmark program. Since a garbage collection process is not activated due to enough free space, we trigger a background garbage collection process after 30% of write requests of an input trace are processed. In actual NAND flash-based storages, the background garbage collection process works as a background thread.

We tested our data separation technique with I/O traces of four programs and two application sets shown in Table 10, which have been used in similar studies [31, 39, 40, 41, 42]. Cscope explores Linux kernel source

Table 9: FlashBench configuration parameters.

Parameters	Value
Read latency	100 us
Write latency	1,600 us
Erase latency	5 ms
Page size	4 Kbytes
Pages per block	128
Buses	4
Chips per bus	2
Mapping scheme	Page level
Victim selection algorithm	Greedy

code, and gcc builds a Linux kernel. Tpc-h and tpc-r are decision support benchmarks; they queries large database files, and can generate multiple concurrent access patterns. Multi1 consists of concurrent executions of cscope and gcc, and represents a program development environment. Multi2 consists of concurrent executions of cscope, gcc, and viewperf, and multi2 represents the workload on a workstation for graphical applications and simulations. Trace of the execution of these benchmarks were created using the *strace* program, which traces Linux system calls. I/O requests generated by system calls are passed through a page cache, we processed the trace file using an LRU-based page cache simulator to create a trace of block-level device.

Table 10: Summary of benchmark programs.

Benchmark	Scenario	Nwrite	Nupdate
cscope	Linux source code examination	17,575	15,398
gcc	Building Linux Kernel	10,394	3,840
tpc-h	Access to database	23,522	20,910
tpc-r	Access to database	21,897	18,803
multi1	cscope+gcc	28,400	19,428
multi2	cscope+gcc+viewperf	35,719	20,106

N_{write}: number of write requests (pages)

N_{update}: number of update write requests (pages)

Figure 35 shows the normalized garbage collection overhead incurred by an FTL using the `PDS`, `IDS`, `HASH` and `ORA` data separation techniques. The x-axis indicates the benchmark traces and the applied techniques, and the y-axis represents the normalized total execution times for garbage collection procedures. The total execution times are normalized to those of `HASH`. We see that `IDS` reduced the time required for garbage collection by 18% and 42%, when compared with `PDS` and `HASH`, respectively. These results suggest that separating data based on update patterns can reduce the overhead of garbage collection; and further that using the hints provided program contexts and the update history of blocks increases this reduction. In particular, `IDS` outperforms `PDS` for all workloads by exploiting both program context hints and the update history of block.

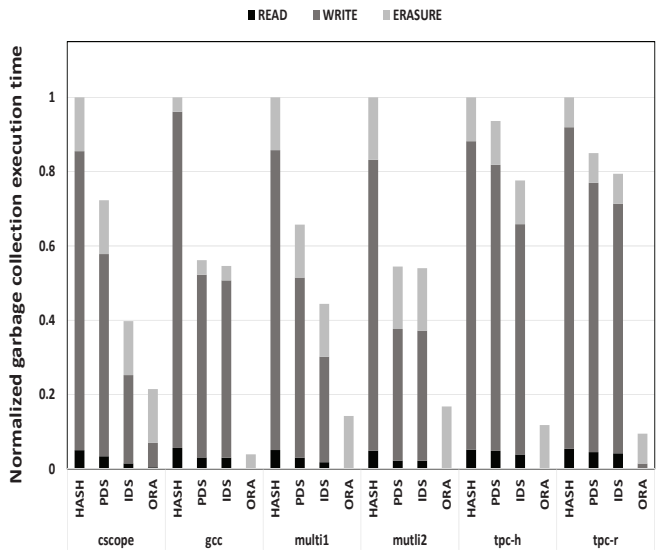
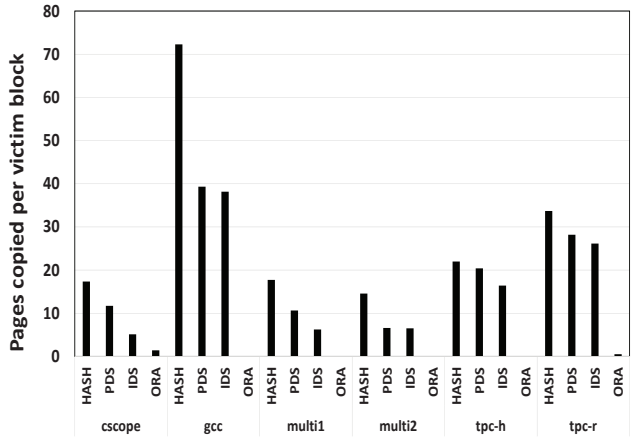
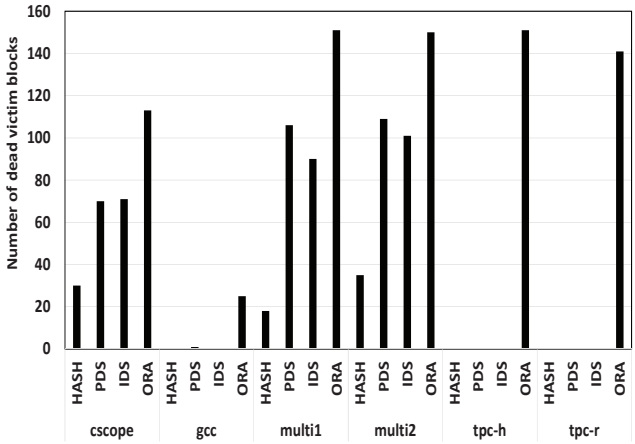


Figure 35: Garbage collection times normalized to HASH.

We went on to characterize the effect of data separation on garbage collection by counting the number of pages that had to be copied for each victim block, and the number of victim blocks that were already dead, and could therefore be reclaimed with a low overhead. The results are shown in Figure 36. In Figures 36(a) and 36(b), the x-axis denotes the benchmark traces and the applied techniques. The y-axis in Figure 36(a) indicates the number of copied pages per victim block during garbage collection procedures. For each victim block, IDS copied 15% and 44% fewer valid pages than PDS and HASH, respectively. This suggests that our integrated approach is successful in gathering data with similar update times into the same victim blocks. Figure 36(b) also shows that PDS and IDS created more dead blocks than HASH. In Figure 36(b), the y-axis represents the number of dead



(a)



(b)

Figure 36: Analysis of the garbage collection overhead in different traces: a) the ratio of the number of pages copied to the number of victim blocks; b) the number of victim blocks that were already dead.

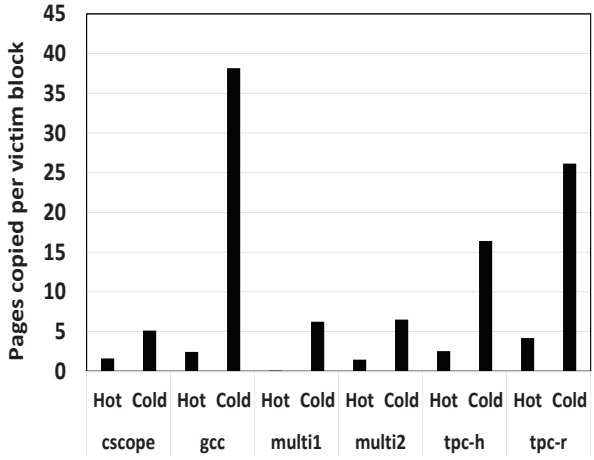


Figure 37: Number of copies per victim block for hot and cold data.

blocks which are erased during garbage collection processes. IDS outperformed HASH and PDS, but made little impact on the random workloads of gcc, tpc-h and tpc-r.

IDS generated more dead blocks because more frequently updated data within similar time periods are gathered in hot archive blocks. Figure 37 shows the effectiveness of exploiting a block-device level hint to segregate hot data. In Figure 37, the x-axis represents the benchmark traces and a victim block type, and the y-axis indicates the number of copied pages per hot archive block. In the x-axis, a hot archive block is denoted as Hot. This figure shows that most pages in hot archive blocks have already been invalidated when they are selected as victim blocks. This confirms that the data gathered in a hot archive block are updated more frequently, and so we can expect this segregation to make a significant contribution to reducing

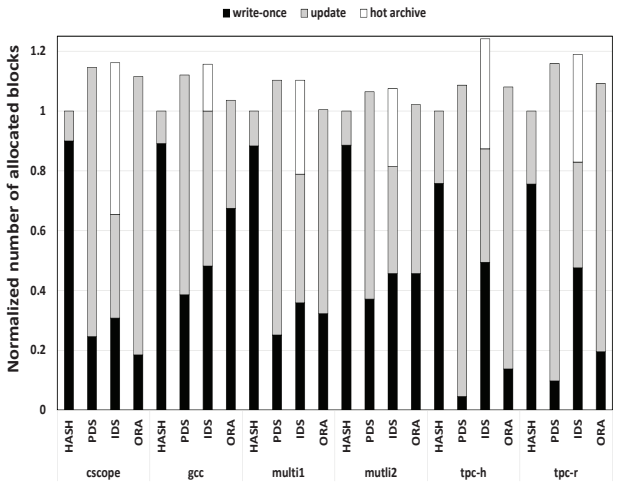


Figure 38: Number of blocks used by PDS, IDS, and ORA, normalized to HASH.

garbage collection overhead. IDS reduced the garbage collection overhead by detecting the very hot data based on the device block level and O/S level hints.

Although the results presented thus far suggest that IDS outperforms other data separation techniques, but we need to look at the technique own use of blocks, which will also affect the frequency of garbage collection. Figure 38 shows the storage requirement of each technique normalized to HASH. The x-axis denotes the benchmark traces and the applied techniques, and the y-axis indicates the normalized number of allocated blocks. PDS, IDS, and ORA classify data into multiple update groups based on estimated (PDS and IDS) or exact (ORA) update times, which is likely to require more blocks. Note, however that 28% of the blocks used by IDS are hot archive blocks, which are likely to be dead blocks, as shown in Figure 37. This

means that IDS is able to reclaim many free blocks without a large overhead during garbage collection.

Chapter 7

Conclusions

7.1 Summary

As NAND flash memory technologies improve, NAND flash-based storage devices are becoming one of attractive storage solutions for various systems from mobile devices to high-end enterprise systems. As the semiconductor process is scaled down and the multi-leveling technology is widely applied, however, the performance and reliability of NAND flash memory are significantly degraded. In order to resolve these problem, we proposed integrated optimization techniques which improve the performance and reliability of high-density NAND flash memory by exploiting various optimization hints in multiple system levels.

For read-disturb problem in NAND flash memory, we have proposed a read disturb-management technique which can reduce overheads from frequent read reclaim procedures in high-density NAND flash memory. Based on the technique, we designed and implemented `redFTL`. `RedFTL` distributes frequently-accessed data in a small number of blocks to multiple blocks, thus significantly reducing the frequency of read reclaim procedures. Moreover, `RedFTL` spreads the time overhead of data migrations during read reclaim by migrating frequently-read pages early. Experimental results show that `redFTL` can reduce the overhead execution time for read reclaim

by 50% over an existing read-disturb management technique.

As an improved version of redFTL, we have proposed a novel integrated approach, called redFTL+, for managing the read-disturb problem in high-density NAND flash memory. Based on read voltage scaling and read time scaling (motivated from the NAND device physics), we introduced read-resistant blocks at the device level, and developed a concept of read-resistant pages. By intelligently exploiting read-resistant blocks and read-resistant pages at the FTL level, we could dramatically reduce the overhead from frequent read reclaims. Experimental results show that redFTL+ can reduce the execution time for read reclaims by 62% over an existing read-disturb management technique.

In order to improve the efficiency of garbage collection, we developed a program context-aware data separation technique, called PDS, which can reduce the garbage collection overhead by exploiting program context hints. Taking account of the correlation between program contexts and update write patterns, PDS predicts update times of data by examining program contexts which have produced the write requests. By gathering the data with similar update times to the same blocks, an FTL based on our technique can reduce total execution time of garbage collection operations by 51% over a hash-based hot/cold separation scheme. The results prove that predicting update times of data with program contexts can be a remarkable method in data separation technique.

As an extended version of PDS, we have proposed an integrated data separation technique, called IDS. IDS uses both program context information and block access history as data separation hints. By detecting more

frequently updated data among data with a high probability to be updated, IDS predicts more accurately update behavior over PDS. Experimental results show that an FTL using IDS can reduce the overhead execution time for garbage collection by 18% over PDS.

7.2 Future Work

7.2.1 Improving QoS of RedFTL+ by Exploiting Program Context Hints

The current version of redFTL+ proactively performs read reclaim when idle time is detected and a fluctuation of I/O response times is significantly reduced. However, a few read requests still conflict with a read reclaim process, thus decreasing the quality of their service of I/O intensive applications. Since providing stable response times is the most important requirement for storage systems in some I/O intensive business applications, a response time delay from the I/O conflicts is required to be reduced.

In order to improve QoS of redFTL+ by eliminating the I/O conflicts, we plan to develop a novel background read reclaim algorithm which determines read reclaim trigger points based on anticipated I/O idle times. For more accurate prediction of idle times, we plan to exploit program context hints in the host system level. Since a program context can indirectly represent access patterns of an application, the program context can be used to predict idle times between I/O requests. Based on the accurate idle time prediction, redFTL+ can avoid I/O conflicts between a normal I/O request and a read reclaim request, thus improving QoS of redFTL+.

7.2.2 Mitigating Read-Disturb Problem by Read Disturb-Aware Read Buffer Management Technique

As mentioned in Chapter 3, an SSD read buffer may not be very effective in reducing read reclaims under read-intensive workloads with large (read) working sets. For such workloads, it might be more beneficial to manage the read buffer from the read disturb perspective instead of conventional hit-ratio centric perspective. For example, it might be a better idea to keep more frequently read data in the read buffer. As a more aggressive optimization, it might be also useful to store read-hot data in a partially disturbed NAND block to the read buffer in a proactive fashion. Considering such potential gains of the read buffer on the read-disturb problem, we plan to devise a read disturb-aware read buffer management technique which can eliminate the read-disturb problem by preferentially caching read-hot in the read buffer. Moreover, we plan to extend `redFTL+` by applying the read buffer management technique, and we will devise a data loading policy which loads read-hot data in a partially disturbed block to the read buffer. By using the read disturb-aware read-buffer management technique, the `redFTL+` can further mitigate the read-disturb problem.

7.2.3 Improving Efficiency of Garbage Collection by Adjusting GC Trigger Points

In NAND flash-based storage systems, determining a garbage collection trigger points is also important to enhance the efficiency of garbage collection as well as locating data with similar update times in the same

block. Although our proposed data separation techniques can help an FTL gather data with similar update times into the same block, if garbage collection is activated before the data in the block are invalidated, unnecessary copies are performed for the valid data. Conversely, if garbage collection activation is deferred until there is no free block, a large number of read, write, and erasure operations must be performed at once to reclaim dirty and dead blocks. In this case, I/O latency can be seriously increased when a normal I/O request is conflicted with the lazy garbage collection process. Therefore, activating garbage collection at the proper time is an important decision factor to eliminate garbage collection overhead.

In this dissertation, our proposed techniques are devised under the condition that background garbage collection is statically triggered at the same period. This means that our proposed techniques can be further improved if we can intelligently control the garbage collection trigger points. An efficient garbage collection process must be activated during idle time and prepare enough free blocks before future writes are requested. If we know when and how many writes are requested, we can further enhance the efficiency of garbage collection.

Program contexts may provide effective hints on anticipating the idle time and the amount of required blocks because both idle time and amount of I/O are determined by a program behavior. Actually, program contexts have been used to predict idle time in hard disk drives [43]. In addition, in our observation, a program context indirectly represents whether it will generate a lot of I/O requests or not. In other words, we can design both a garbage collection triggering policy and a data allocation policy based on

program context hints. We plan to develop a garbage collection management technique which can dynamically adjust garbage collection triggering points based on program context hints. If an FTL can exploit both the GC triggering technique and our proposed data separation technique, it may choose GC victim blocks with mostly invalid pages when if requires garbage collection.

7.2.4 Improving Performance and Reliability of NAND Flash Memory by Integrating Various Techniques

In order to improve performance and reliability of high-density NAND flash memory, we plan to integrate our proposed techniques. By combining `RedFTL+` and `IDS`, we can devise a novel FTL which can reduce the overheads of read-disturb management and garbage collection. Over the integrated FTL based on our proposed techniques, we will extend the FTL by merging other cross-layer optimization techniques. For example, in order to address weak endurance of high-density NAND flash memory, we can integrate our integrated FTL with dynamic program and erase scaling (`DPES`) [44], which extends SSD lifetime by modifying the voltage of write and erase operations. Because of the enhanced endurance, the cross-layer FTL can further improve the reliability of NAND flash memory.

Bibliography

[1] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012.

[2] A. A. Chien and V. Karamcheti, "Moore's Law: The First Ending and A New Beginning," 2012. Technical Report.

[3] M. Kang, K.-T. Park, Y. Song, S. Hwang, B. Choi, Y. Song, Y.-T. Lee, and C. Kim, "Improving Read Disturb Characteristics by Self-boosting Read Scheme for Multilevel NAND Flash Memories," *Japanese Journal of Applied Physics*, vol. 48, no. 4, p. 04C062, 2009.

[4] H. H. Frost, C. J. Camp, T. J. Fisher, J. A. Fuxa, and L. W. Shelton, "Efficient Reduction of Read Disturb Errors in NAND FLASH Memory," 2010. US Patent 7,818,525.

[5] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of Flash Translation Layer.," *Journal of Systems Architecture - Embedded Systems Design*, vol. 55, no. 5-6, pp. 332–343, 2009.

[6] 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 *Proc. of the IEEE International Reliability Physics Symposium*, 2008.

[7] M. Lenzlinger and E. Snow, "Fowler-nordheim Tunneling into Thermally Grown SiO2," *Journal of Applied Physics*, vol. 40, no. 1, pp. 278–283, 1969.

[8] J. Thatcher, T. Coughlin, J. Handy, and N. Ekker, "NAND Flash Solid State Storage for The Enterprise: An In-depth Look at Reliability," *Solid State Storage Initiative*, 2009.

[9] S. Lee, "Scaling Challenges in NAND Flash Device toward 10nm Technology," in *Proc. of the IEEE International Memory Workshop*, 2012.

[10] L.-P. Chang and T.-W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," in *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, pp. 187–196, IEEE, 2002.

[11] J.-W. Hsieh, L.-P. Chang, and T.-W. Kuo, "Efficient on-line identification of hot data for flash-memory management," in *Proceedings of the 2005 ACM symposium on Applied computing*, 2005.

[12] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient Identification of Hot Data for Flash Memory Storage Systems," *ACM Transactions on Storage (TOS)*, vol. 2, no. 1, pp. 22–40, 2006.

[13] D. Park and D. H. Du, "Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters," in *Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies (MSST 2011)*, 2011.

[14] S. Jung, Y. Lee, and Y. H. Song, "A process-aware hot/cold identification scheme for flash memory storage systems," *Consumer Electronics, IEEE Transactions on*, vol. 56, no. 2, pp. 339–347, 2010.

[15] L.-P. Chang, "Hybrid Solid-State Disks: Combining Heterogeneous NAND Flash in Large SSDs," in *Design Automation Conference, 2008. ASPDAC 2008. Asia and South Pacific*, pp. 428–433, IEEE, 2008.

[16] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, M. Kawaba, and C. Pu, "Response Time Reliability in Cloud Environments: An Empirical Study of n-Tier Applications at High Resource Utilization," in *Proc. of the IEEE Symposium on Reliable Distributed Systems*, 2012.

[17] M. Phil and A. D. Sena, "Reducing NAND Defects and Failures with Micron NFTL," Mar 2011. Micron Software Article Archive.

[18] "http://iotta.snia.org/traces/158."

[19] S.-H. Shin, D.-K. Shim, J.-Y. Jeong, O.-S. Kwon, S.-Y. Yoon, M.-H. Choi, T.-Y. Kim, H.-W. Park, H.-J. Yoon, Y.-S. Song, Y.-H. Choi, S.-W. Shim, Y.-L. Ahn, K.-T. Park, J.-M. Han, K. K.-H., and Y.-H. Jun, "A New 3-bit Programming Algorithm Using SLC-to-TLC Migration for 8MB/s High Performance TLC NAND Flash Memory," in *Proc. of the IEEE Symposium on VLSI Circuits*, 2012.

[20] D. Narayanan *et al.*, "Write Off-Loading: Practical Power Management for Enterprise Storage," *ACM Transactions on Storage*, vol. 4, no. 3, p. 10, 2008.

[21] J. Zhang *et al.*, "Synthesizing Representative I/O Workloads for TPC-H," in *Proc. of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[22] "Websearch Trace from UMass Trace Repository. http://traces. cs.umass.edu/index.php/Storage/Storage."

[23] S. Lee, J. Park, and J. Kim, "FlashBench: A Workbench for A Rapid Development of Flash-based Storage Devices," in *Proc. of the IEEE International Symposium on Rapid System Prototyping*, 2012.

[24] J. Jeong, S. S. Hahn, S. Lee, and J. Kim, "Improving NAND Endurance by Dynamic Program and Erase Scaling," in *Proc. of the USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.

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

[26] K. Fukuda, Y. Shimizu, K. Amemiya, M. Kamoshida, and C. Hu, "Random Telegraph Noise in Flash Memories-Model and Technology Scaling," in *Electron Devices Meeting*, 2007.

[27] Y. Pan, G. Dong, and T. Zhang, "Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 2011.

[28] I. J. Chang and J.-S. Yang, "Bit-error Rate Improvement of TLC NAND Flash Using State Re-ordering," *IEICE Electronics Express*, vol. 9, no. 23, pp. 1775–1779, 2012.

[29] M. Goldman, K. Pangal, G. Naso, and A. Goda, "25nm 64Gb 130mm^2 3bpc NAND Flash Memory," in *Proc. of the IEEE International Memory Workshop*, 2013.

[30] "Getting the Hang of IOPS." http://www.symantec.com/connect/articles/getting-hang-iops-v13.

[31] C. Gniady, A. R. Butt, and Y. C. Hu, "Program-counter-based Pattern Classification in Buffer Caching," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[32] "http://iotta.snia.org/traces/133."

[33] "Telecom Application Transaction Processing Benchmark." http://tatpbenchmark.sourceforge.net.

[34] F. Zhou, J. R. von Behren, and E. A. Brewer, "Amp: Program context specific buffer caching.," in *USENIX Annual Technical Conference, General Track*, 2005.

[35] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *Journal of Systems and Software*, vol. 48, no. 3, pp. 213–231, 1999.

[36] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pp. 42–53, ACM, 2000.

[37] A.-C. Lai and B. Falsafi, "Selective, Accurate, and Timely Self-invalidation Using Last-Touch Prediction," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pp. 139–148, IEEE, 2000.

[38] R. Sibson, "SLINK: An Optimally Efficient Algorithm for the Single-Link Cluster Method," *The Computer Journal*, vol. 16, no. 1, pp. 30–34, 1973.

[39] Y. Zhu and H. Jiang, "RACE: A Robust Adaptive Caching Strategy for Buffer Cache," *Computers, IEEE Transactions on*, vol. 57, no. 1, pp. 25–40, 2008.

[40] A. R. Butt, C. Gniady, and Y. C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms," *Computers, IEEE Transactions on*, vol. 56, no. 7, pp. 889–908, 2007.

[41] M. Wang and Y. Hu, "An I/O Scheduler Based on Fine-grained Access Patterns to Improve SSD Performance and Lifespan," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pp. 1511–1516, ACM, 2014.

[42] S. H. Baek and K. H. Park, "Prefetching with Adaptive Cache Culling for Striped Disk Arrays," in *USENIX Annual Technical Conference*, pp. 363–376, 2008.

[43] C. Gniady, Y. C. Hu, and Y.-H. Lu, "Program Counter Based Techniques for Dynamic Power Management," in *Proc. of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[44] J. Jeong, S. S. Hahn, S. Lee, and J. Kim, "Lifetime Improvement of NAND Flash-based Storage Systems Using Dynamic Program and

Erase Scaling," in *Proc. of the USENIX Conference on File and Storage Technologies*, 2014.

초 록

낸드 플래시 기반의 저장장치는 스마트 폰과 같은 모바일 임베디드 시스템에서부터 랩톱, 데스크톱 컴퓨터, 기업용 서버 시스템에 이르기까지 다양한 컴퓨팅 환경에서 널리 사용되고 있다. 낸드 플래시 기반의 저장장치가 이와 같이 다양한 시스템에 활용될 수 있었던 것은 메모리 공정의 미세화, 멀티 레벨 셀 기술 등의 발전에 있다. 낸드 플래시 메모리의 집적도의 향상은 낸드 플래시 메모리의 용량 증가를 견인하기는 했으나, 플래시 메모리 칩의 성능, 수명, 데이터의 신뢰성 등이 급격히 감소하는 부작용을 초래했다. 이러한 문제를 해결하기 위해서 기존의 하나의 시스템 레벨에서의 시스템 정보나 자원을 활용하여 최적화하는 단일 계층 중심의 최적화 기법들이 제안되어 왔으나, 단일 계층 최적화 기법들의 개선 수준은 한계에 부딪힌 상황이다. 따라서, 기존 단일 계층 최적화 기법의 한계를 뛰어넘을 수 있는 새로운 차원의 최적화 기법이 요구된다.

본 논문에서는 낮은 데이터 신뢰성과 성능을 갖는 대용량 낸드 플래시 메모리의 관리 상에서 발생할 수 있는 성능 저하를 최소화 하기 위한 계층 교차적 최적화 기법을 제안한다. 계층 교차적 최적화 기법이란 디바이스 수준, 운영체제, 응용 프로그램 등의 다양한 수준에서의 최적화 요소들을 함께 고려하고 활용하여 성능 및 데이터 신회성을 개선하는 방법을 말한다. 본 논문에서 제안된 계층 교차적 회적화 기법들은 낸드 플래시 메모리의 물리적 특성과 응용 프로그램에서의 I/O 동작 정보 등을 저장장치 시스템 내에서 효과적으로 활용함으로써 대용량 낸드 플래시 메모리에서의 가비지 콜렉션 수행에 따른 성능 저하, 가용 읽기 쓰기 수의 감소에 따른 데이터 신뢰성 유지를 위해 발생하는 부가적인 연산에 의한 성능 부하 증가 등의 문제를 해결한다.

대용량 낸드 플래시 메모리에서의 제한된 읽기 횟수 감소에 따른 성능 부하 증가 문제를 해결하기 위해서 RedFTL과 RedFTL+ 라는 플래시 변환 계층을 제안한다. 먼저, RedFTL은 응용 프로그램들이 저장장치 내에 영역 별 읽기 불균형 현상에 의한 읽기 신뢰성 저하 문제를 해결하기 위해서, 읽기 부하를 저장장치 모든 영역에 골고루 분해함으로써 과도한 읽기에 의한 특정 영역 내의 데이터 유

실을 방지한다. RedFTL의 개선된 버전인 RedFTL+는 플래시 메모리 내에서 사용되는 읽기 전압과 읽기에 대한 내성과의 관계를 활용하여, 읽기 전압을 낮추는 방법을 통하여 근본적으로 보다 많은 읽기 연산을 처리할 수 있게 한다. RedFTL과 RedFTL+는 읽기에 의한 데이터 유실 가능성을 낮춰 주는 동시에 데이터 신뢰성 유지를 위해 필요한 데이터 복사 및 블록 삭제 연산의 횟수를 줄임으로써 성능 저하를 최소화한다.

또한, 대용량 낸드 플래시 메모리에서의 느린 연산 속도에 의한 가비지 콜렉션 성능 저하 문제를 해결하기 위하여 응용 프로그램의 쓰기 동작을 고려한 계층 교차적인 데이터 분류 기법을 제안한다. 본 연구에서 제안한 데이터 분류 기법은 응용 프로그램 내에서의 특정 함수들로부터 발생하는 반복적인 쓰기 경향을 분류하고, 이 정보를 활용하여 비슷한 덮어쓰기 성향을 갖는 데이터들을 저장장치 내에 동일한 블록에 저장함으로써 가비지 콜렉션의 효율성을 증가시킨다.

본 논문에서 제안한 기법들은 자체 개발한 저장장치 에뮬레이터 및 시뮬레이터에 구현되었다. 실제 응용 프로그램들을 수행하면서 추출된 I/O 명령들을 활용하여 제안된 기법들의 효과를 검증하였으며, 실험 결과들을 통하여 우리는 본 논문에서 계층 교차 최적화 기법들이 대용량 낸드 플래시 메모리의 성능 및 읽기 신뢰성 개선에 기여함을 확인할 수 있었다.

키워드: 낸드 플래시 메모리, 플래시 기반 저장 장치, 저장장치 성능 최적화, 저장장치 신뢰성 관리, 운영 체제, 임베디드 시스템
학번: 2005-21527