Fall 2012

Improving Performance of Solid State Drives in Enterprise Environment

Jian Hu
University of Nebraska-Lincoln, jhu@cse.unl.edu

IMPROVING PERFORMANCE OF SOLID STATE DRIVES IN ENTERPRISE ENVIRONMENT

by

Jian Hu

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Engineering

Under the Supervision of Professor Hong Jiang

Lincoln, Nebraska

December, 2012

IMPROVING PERFORMANCE OF SOLID STATE DRIVES IN ENTERPRISE ENVIRONMENT

Jian Hu, Ph. D.

University of Nebraska, 2012

Adviser: Hong Jiang

Flash memory, in the form of Solid State Drive (SSD), is being increasingly employed in mobile and enterprise-level storage systems due to its superior features such as high energy efficiency, high random read performance and small form factor. However, SSD suffers from the *erase-before-write* and endurance problems, which limit the direct deployment of SSD in enterprise environment. Existing studies either develop SSD-friendly on-board buffer management algorithms, or design sophisticated Flash Translation Layers (FTL) to ease the erase-before-write problem. This dissertation addresses the two issues and consists of two parts.

The first part focuses on the white-box approaches that optimize the internal design of SSD. We design a write buffer management algorithm on top of the log-block FTL, which not only optimizes the write buffer effect by exploiting both the recency and frequency of blocks in the write buffer, but also minimizes the destaging overhead by maximizing the number of valid pages of the destaged block. We further identify that the low garbage collection efficiency problem has a significantly negative impact to the performance of the page-mapped SSD. We design a GC-Aware RAM management algorithm that improves the GC efficiency even if the workloads do not have updating requests by dynamically evaluating the benefits of different destaging policies and adaptively adopting the best one. Moreover, this algorithm minimizes the

address translation overhead by exploiting the interplay between the buffer component and the FTL component.

The second part focuses on the black-box approaches that optimize the SSD performance externally. As an increasing number of applications deploy SSD in enterprise environment, understanding the performance characteristics of SSD in enterprise environment is becoming critically important. We identify several performance anomalies of SSDs and their performance and endurance impacts on SSD employed in enterprise environment by evaluating several commercial SSDs. Our study provides insights and suggestions to both system developers and SSD vendors. Further, based on the performance anomalies identified, we design an IO scheduler that takes advantage of the SSD features and evaluate its performance on SSD. The scheduler is shown to improve performance in terms of bandwidth and average response time.

ACKNOWLEDGMENTS

First of all, I would like to express my thanks to my advisor, Dr. Hong Jiang, for his support, guidance and encouragement during my five-year Ph.D. study at University of Nebraska-Lincoln. Dr. Jiang not only teaches me how to do research, but also teaches me how to improve my writing skills and presentations. Without his patient advise and input, this dissertation would never have been possible. The training I get here from Dr. Jiang will help my future career development significantly. It has been a great honor for me to do research under Dr. Jiang's supervision.

I would also like to express my thanks to my advisory committee members Dr. Lisong Xu, Dr. Witawas Srisa-an and Dr. Song Ci for their very helpful input during my study. Their valuable advise and support have played an important role in my completing my Ph.D. study.

I'm particularly grateful to Lei Tian and Bo Mao for their input and discussions on my research. They are always ready when I have questions that I can not solve on my own. Without their patient help and time, my research would not have gone so smoothly.

I also would like to thank all members of the ADSL group. These people include Lei Xu, Hao Luo, Dongyuan Zhan, Ziling Huang, Yaodong Yang, Junjie Qian and Stephen Mkandawire. They are very helpful in our group meetings and discussions. I feel very glad to work with them in the ADSL group for the past five years.

I thank my parents for their support and love. They always encourage me when I feel discouraged and want to give up.

Most of all, I must thank my wife, Chijun Deng, for her endless love and encouragement. During the past five years, Chijun sacrificed a lot of her time to help me study and research. Without her, I would never have finished my dissertation.

Contents

Contents **v**

List of Figures **x**

List of Tables **xiii**

1 Introduction **1**

1.1 NAND flash memory SSD characteristics 1

1.1.1 Characteristics 1

1.1.2 Two physical limitations of SSD 2

1.2 SSD functional components 3

1.2.1 Flash translation layer (FTL) 5

1.2.2 SSD-friendly write buffer 8

1.3 The SSD design space and the scope of the dissertation 10

1.3.1 White-box approaches 10

1.3.2 Black-box approaches 12

1.4 Contributions of the dissertation 13

1.5 Dissertation Organization 16

2 Design and Implementation of a Write Buffer Management Algorithm of SSD 17

2.1 Background . . . 18

2.2 Design and implementation . . . 22

2.2.1 Predicted Average Update Distance . . . 23

2.2.2 Destaging Efficiency . . . 25

2.2.3 Implementation of PUD-LRU . . . 27

2.2.4 Data integrity concern . . . 30

2.3 Performance evaluation . . . 30

2.3.1 Experimental setup . . . 30

2.3.2 Number of erasures and average response time . . . 31

2.3.3 Sensitivity study . . . 35

2.3.4 Destaging efficiency . . . 37

2.3.5 Summary . . . 39

3 Design and Implementation of Garbage Collection Efficiency-Aware RAM Management 41

3.1 Background . . . 42

3.1.1 Performance impact of GC efficiency in SSD . . . 42

3.1.2 GC-efficiency obliviousness of write-buffer management . . . 46

3.1.3 Partitioning RAM for the write buffer and the mapping table 49

3.1.4 Reducing write-back traffic due to mapping entry replacement 50

3.2 Design and implementation of GC-ARM . . . 53

3.2.1 The design of FTL component . . . 54

3.2.2 GC-aware destaging . . . 56

3.2.3 Adaptive adjustment of RAM space partitioning . . . 58

3.2.4 Interplay between write buffer and cached mapping table . . . 59
3.2.5 Data integrity concern . . . 59
3.3 Performance evaluation . . . 60
3.3.1 Experimental setup . . . 60
3.3.2 Performance and GC efficiency . . . 60
3.3.3 Write traffic reduction . . . 69
3.4 Summary . . . 73

4 Identifying Performance Anomalies in Enterprise Environment 74
4.1 Background . . . 75
4.1.1 Low GC efficiency . . . 75
4.1.2 Predicting the residual lifetime of SSDs . . . 75
4.1.3 High random read performance . . . 76
4.1.4 Achieving high bandwidth and low average response time simultaneously . . . 77
4.2 Experiment . . . 77
4.2.1 Experiment setup . . . 77
4.2.1.1 Experimental evaluation environment . . . 77
4.2.1.2 SSD-specific considerations and evaluation methodology 78
4.2.1.3 Evaluation tools . . . 79
4.2.1.4 Commercial SSD products evaluated . . . 80
4.2.2 Performance impact of GC efficiency . . . 82
4.2.3 Performance impact of the mapping policy of FTL . . . 87
4.2.4 Predicting the residual lifetime of SSD based on a predefined space allocation scheme . . . 89
4.2.5 Unpredictability of random read performance . . . 92

4.2.6 Achieving high bandwidth and low average response time simultaneously . . . 98
4.3 Guidelines and suggestions . . . 102
4.4 Summary . . . 105

5 SSD-Friendly IO Scheduling 107
5.1 Background . . . 108
5.1.1 IO schedulers and its ineffectiveness to SSD . . . 108
5.1.2 Ineffectiveness of conventional IO schedulers to SSD . . . 110
5.1.3 Impact of mixing read and write requests . . . 112
5.1.4 Key factors determining the performance of SSD . . . 114
5.2 Design and implement of SSD-friendly IO scheduler . . . 115
5.2.1 Dividing (logical block address) LBA into zones to maximize parallelism . . . 116
5.2.2 Splitting read and write requests . . . 117
5.2.3 Merging to increase the request size . . . 117
5.2.4 Flowchart of SSD-friendly IO scheduler . . . 118
5.3 Performance evaluation . . . 119
5.3.1 Experiment setup . . . 119
5.3.2 Result analysis . . . 120
5.4 Summary . . . 124

6 Related Work 125
6.1 NVRAM . . . 125
6.2 Study categorization . . . 126

7 Conclusion and Future Work 128

7.1 PUD-LRU write buffer management algorithm 128
7.2 GC-ARM on-board RAM management algorithm 129
7.3 Identifying performance anomalies of SSD in enterprise environment . 130
7.4 SSD-friendly IO scheduler 132

Bibliography **133**

List of Figures

1.1 Overview of SSD design space 4

2.1 Analysis of temporal locality of two workloads 20
2.2 An architectural view of the PUD-LRU-based SSD 23
2.3 Number of erasures and avg response time of Financial 1 workload . . . 32
2.4 Number of erasures and avg response time of Financial 2 workload 32
2.5 Number of erasures and avg response time of Exchange workload 33
2.6 Number of erasures and avg response time of Build workload 33
2.7 Number of erasures and avg response time of TPC-E workload 34
2.8 Number of erasures and average response time based on different threshold 36
2.9 Destaging efficiency of five workloads 38

3.1 Performance impact of GC efficiency in SSDs 44
3.2 Comparison between different destaging policies showing the advantage of GC awareness 48
3.3 Number of distinctive blocks accessed per 500 requests 51
3.4 Interplay between pages in the write buffer and the mapping entries . . . 52
3.5 The GC-ARM architecture 54
3.6 GC efficiency evaluation of Financial 1 workload 61
3.7 GC efficiency evaluation of Financial 2 workload 62

3.8 GC efficiency evaluation of Exchange workload 63
3.9 GC efficiency evaluation of MSN workload 66
3.10 GC efficiency evaluation of TPC-E workload 67
3.11 Write traffic distribution and the total number of writes of Financial 1 . 69
3.12 Write traffic distribution and the total number of writes of Financial 2 . 70
3.13 Write traffic distribution and the total number of writes of Exchange . . 70
3.14 Write traffic distribution and the total number of writes of MSN 71
3.15 Write traffic distribution and the total number of writes of TPC-E 71

4.1 Performance impact of GC efficiency in SSDs – bandwidth as a function of the amount of data written and the percentage of reserved LPN range 81
4.2 Examples illustrating GC efficiency when different LPN range is reserved 83
4.3 Performance impact of GC efficiency in SSDs – write response time as a function of the amount of data written and the percentage of reserved LPN range 85
4.4 Impact of mapping policies – bandwidth of 120GB OCZ Vertex 3 SSD as a function of the amount of data written and the percentage of reserved LPN range 89
4.5 Residual lifetime as a function of reserved LPN range and the amount of data written 92
4.6 Average response time, standard deviation and distribution of response time of the Samsung 470 Series SSD 93
4.7 Average response time, standard deviation and distribution of response time of the Intel 320 SSD 94
4.8 Average response time, standard deviation and distribution of response time of the Fusion IO ioDrive 95

4.9 Average response time of read requests of Intel 320 SSD at different GC efficiency 98
4.10 Tradeoff between bandwidth and average response time of Samsung 470 SSD 100
4.11 Tradeoff between bandwidth and average response time of Intel 320 SSD 101
4.12 Tradeoff between bandwidth and average response time of Fusion IO ioDrive 102

5.1 Bandwidth as a function of different schedulers and workloads of Fusion IO ioDrive 111
5.2 Bandwidth as a function of different schedulers and workloads of Sandisk SSD 112
5.3 Impact of mixing read and write requests – bandwidth as a function of the percentage of sequential write requests 113
5.4 Overall architecture of SSD-friendly IO scheduler 115
5.5 Flowchart of the SSD-friendly IO scheduler 119
5.6 Bandwidth and average response time as a function of number of zones and schedulers for Financial 1 workload 121
5.7 Bandwidth and average response time as a function of number of zones and schedulers for Financial 2 workload 121
5.8 Bandwidth and average response time as a function of number of zones and schedulers for Exchange workload 122
5.9 Bandwidth and average response time as a function of number of zones and schedulers for MSN workload 122
5.10 Bandwidth and average response time as a function of number of zones and schedulers for Build workload 123

List of Tables

2.1 A performance comparison between BPLRU and page-mapping FTL . . 18

2.2 I/O characteristics of workloads studied 21

2.3 An illustrative comparison between PUD-LRU and BPLRU 26

2.4 Configuration parameters of SSD 31

4.1 Specifications of SSDs 80

Chapter 1

Introduction

NAND Flash memory, in the form of Solid State Drive (SSD), is being increasingly employed in mobile and personal storage systems to improve performance, save energy and increase mobility due to its attractive features such as energy efficiency, good random read performance, strong shock resistance and small form factor, while enterprise-level storage systems and many enterprise applications also employ SSD as a cache device to meet mission-critical requirements [10, 19, 28, 32, 47].

1.1 NAND flash memory SSD characteristics

1.1.1 Characteristics

NAND flash memory consists of a large amount of flash blocks, each of which in turn consists of many fix-sized pages (4KB or 2KB). The access to NAND flash memory is at the granularity of a page. These blocks are called *erase blocks*. Each page inside an erase block can only be written once before the entire block is erased, a NAND flash memory specific requirement known as *erase-before-write* [29]. The value of each target bit in NAND flash can only be programmed from "1" to "0" in a write

operation, but not in the reversed direction. Once a page is written, it must be erased, where all bits are reset to "1", before the next write operation can be performed on the same page. Furthermore, while the write operation can be applied to an individual page, the erase operation can only be applied to an entire block at a time. Moreover, an erase operation is much slower than both a write and read operation. Therefore, many studies try to reduce the number of erasures, delay the erase operations or parallelize them.

When an update (write) request arrives, the targeted pages need to be invalidated and new pages are allocated to store the new content, also known as *out-of-place* update. Because NAND flash memory does not have a moving head as in Hard Drive Disk (HDD), the random read performance is much higher than that of HDD. The high random read performance is one of the best features that system developers rely on to deploy SSD in either enterprise systems or personal computers.

Despite of NAND flash memory SSD's many attractive features, it suffers from the following two physical limitations that restrict the direct deployment of SSD in the enterprise-level systems.

1.1.2 Two physical limitations of SSD

The first restriction comes from the *erase-before-write* and *out-of-place* update characteristics of SSD mentioned above. Over time, after update requests have generated many invalid pages, blocks containing invalid pages must be erased after valid pages in these blocks are copied to other newly allocated blocks, a process often referred to as *Garbage Collection* (GC). The GC process is a read-write-erase operation sequence that first moves the valid pages from the victim block to a free block, then erases the victim block. The GC process affects the performance of SSD significantly not only

because an erase operation takes much more time than a read (e.g., X100) or write (e.g., X10) [41], but also because GC induces significant write amplification [45], i.e., copying valid pages in the victim blocks to other available blocks before they are erased. Due to the time-consuming erase operations and data movement, this GC problem results in poor write performance, especially for a random-write workload, thus negating the advantages of SSD in case of write-intensive workloads. Previous studies [11, 22, 51] have shown that as more and more data is written to SSD, the performance of SSD drops sharply because of the slow erase operations caused by the GC process.

Another limitation of SSD that affects its direct application in an enterprise environment is the endurance problem. The count of erasures per block for the SLC flash memory, where each cell contains a single bit of value, is typically limited to 100,000 while that for the MLC flash memory, where each cell contains 2 bits of value, is limited to only 10,000 or 5,000 [18]. As the density of the NAND flash memory in SSD further increases, the count of erasures of the TLC flash memory, where each cell contains 3 bits of value, is reduced still further. It was reported that SSDs based on the MLC flash in an enterprise environment could easily wear out in 23 days of use [42], which is far too short to be useful.

1.2 SSD functional components

Figure 1.1 represents the general functional components and overall design space of SSD. A typical SSD is composed of an interface, a controller in the form of firmware, a Flash Translation Layer (FTL), a write buffer or cache and an error-correction-code functional component. The SSD is connected to the host machine, which has an OS, file system, device driver and IO scheduler from top to bottom. Both the FTL and

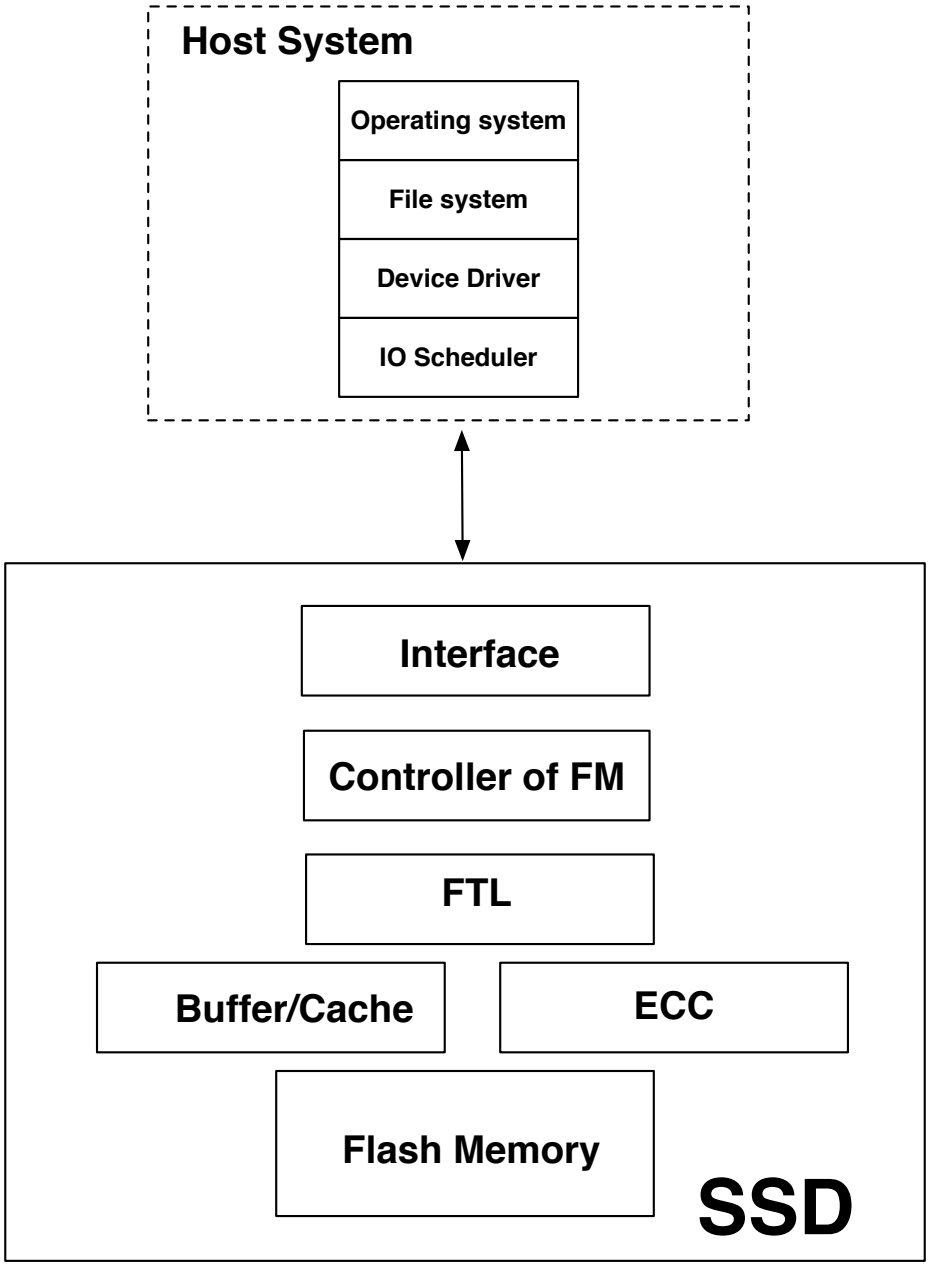


Figure 1.1: Overview of SSD design space

write-buffer functional components use the on-board RAM to store mapping entries or buffer data.

1.2.1 Flash translation layer (FTL)

Flash Translation Layer (FTL) is a software layer sitting on top of the physical medium of SSD, and maintains a mapping table in a small RAM in SSD that maps a request's logical page number (LPN) to its physical page number (PPN) to emulate a hard disk drive. Further, by means of intelligent wear-leveling and GC mechanisms, FTL can evenly distribute erasures to flash blocks, achieving the goals of improving the performance and lengthening the lifetime of SSD.

Many studies try to utilize the on-board RAM as an FTL on the FTL design to ease the *erase-before-write* problem [3, 4, 9, 13, 25, 29, 30, 33, 48], by either employing multiple mapping granularities [3,4,25,33] or exploiting the content similarities in the write requests [9].

Page-mapping FTL and block-mapping FTL are two classic FTL algorithms. In a page-mapping FTL [3], a mapping table is used to store and manage the mapping between LPN and PPN. LPN is used as an application-visible data index above the block interface while PPN is used to index the actual data by SSD at the device level. Page-mapping FTL can map an LPN to wherever a PPN is available in SSD and garbage-collect only the erase blocks with invalid pages, thus minimizing the number of erase operations. However, storing the page-mapping table requires a very large memory capacity. With flash memory's continued growth in size, this memory overhead increases dramatically. For example, a 128GB SSD with a page size of 2KB has 2^{26} entries in its mapping table. If an entry contains 16 bytes, the page-mapping table will occupy 1GB memory, which is too expensive for the SSD to be cost-effective.

To reduce the space overhead of page-mapping FTL, block-mapping FTL [4] divides a user request's LPN into two parts: a logical block number (LBN) and a page offset within the block, and maps the LBN to any physical block number (PBN) in

SSD. As a result, the block-mapping table size is reduced to $\frac{1}{N}$ of that of a page-mapping table, where N is the number of pages in each erase block. However, an LPN can only be mapped to a fixed page offset in any physical block. The page offset is calculated as LPN%N. So, it is very likely that two LPNs having the same page offset will conflict, if there is no other block with a free page at the same page offset. In this case, up to two erase operations must be initiated, which adversely affects the performance.

Several hybrid FTL algorithms have been proposed to address the intrinsic problems of page-mapping FTL and block-mapping FTL. Log-block FTL [25], also known as Block Associative Sector Translation (BAST), is a popular FTL algorithm that is widely used in commercial SSD products, which uses multiple blocks as log blocks to log users' writes in flash memory. The block-length sequence of write requests will be first issued to a log block from its first page to its last page sequentially. Once every page in this log block is written, the log block will replace the corresponding data block. This operation is called *switch merge*, in which only one erase operation is required. However, if the log block is not written sequentially from the first page to the last page, a new block is allocated and a *full merge* operation is initiated, meaning that it will copy the valid pages from the log block and its corresponding data block to the newly allocated data block. Then, the log block and its corresponding data block are erased. In this case, two erase operations are required. Although log-block FTL is more flexible than block-mapping FTL, it suffers from the block thrashing problem [29]. In other words, intensive random writes consume log blocks very rapidly, and thus the FTL has to spend much of its time merging and erasing while postponing processing newly arrived requests, which significantly degrades the performance.

Several improved versions of the original log-block based FTL algorithm have been proposed to exploit temporal locality or sequential locality in the access patterns.

Fully Associative Sector Translation (FAST) [29] allocates the same log block to more than one data block, which increases the utilization of log blocks. However, it suffers from a slow merge problem because the algorithm has to search for more than one data block to execute a full merge. Moreover, it can not handle more than one sequential stream or differentiate a sequential stream from among random requests.

Based on FAST, the Locality-Aware Sector Translation (LAST) [30] scheme employs an access detector to detect whether a request is sequential or random. However, the fact that it simply declares that consecutive requests larger than 4KB are sequential makes it somewhat arbitrary and potentially ineffective.

Demand-based page-level FTL (DFTL) [13] was recently proposed to address the problem caused by the limited number of log blocks in the log-structured FTLs by employing a locality-aware and space-efficient page-level mapping scheme. It reduces the RAM space required to store the page-mapping table by storing only a small subset of the mapping entries in RAM in an LRU queue while leaving the rest in reserved pages in SSD. Mapping entries are stored consecutively based on their logical page numbers (LPNs) in the reserved *translation pages* in SSD. Translation pages in SSD are further organized into *translation blocks*. For example, a typical $4KB$ translation page can store as many as 512 8-Byte logically contiguous mapping entries. Mapping entries in RAM are destaged to translation pages in SSD when the RAM-cashed LRU queue is full and a request miss on this queue causes an entry replacement. Each time an entry from RAM is destaged to a translation page, the latter is invalidated after its old content and the newly destage entry are copied to a newly allocated translation page from the reserved SSD pool for translation blocks, as a result of the *out-of-place* update property of SSD [13]. When no valid translation pages can be found in the reserved SSD pool for a destaged entry, the garbage collection process is invoked to recycle translation blocks, which results in a significant number of wasteful and harmful

translation erasures that would never happen in a pure page-mapping scheme.

1.2.2 SSD-friendly write buffer

The on-board RAM can be utilized as a write buffer [16, 21, 23, 24, 34, 43]. As a write buffer, the frequently updated data blocks can be kept in the buffer for as long a time as possible before being destaged to the flash physical medium. As a result, a write buffer can not only help decrease the number of erasures but also offer a better I/O performance since a majority of requests can be serviced at the RAM speed.

BPLRU (Block Padding Least Recently Used) [23] is a recently proposed SSD-friendly write buffer algorithm that aims to improve the random write performance of SSDs. It sits on top of a log-block FTL [25, 29] and converts random write requests to sequential ones by first reading the missing pages from flash memory and writing them back to SSD again to fill the "holes", a process known as *page padding* [24]. However, when the request size of the workloads is small and the LPNs of the write requests are scattered (i.e., non-sequential), BPLRU becomes very inefficient because the padding process increases the traffic by reading pages from SSD and writing them back again. More importantly, the increased write-back traffic increases the write amplification because these writes are artificially created by page-padding, not the original workloads. Although BPLRU outperforms other flash-aware buffer management algorithms such as CFLRU (Clean First LRU) [34] and FAB (Flash Aware Buffer Policy) [21] in minimizing the number of erasures and reducing the average response time, it fails to consider the update frequency of each block in the buffer. This, based on our observations, may be harmful to SSD for some typical enterprise and OLTP workloads. For example, an application that updates a database may frequently write to the same block because several columns of a row in a table are often

updated at the same time and their addresses are adjacent to one another. These addresses are likely located in the same or nearby blocks with high probabilities. In other words, when the database frequently performs update operations, some blocks in the underlying storage device will be far more frequently updated than others. This will likely incur expensive erase operations if the storage device is SSD.

CFLRU (Clean First LRU) [34] is a flash-aware buffer-cache management algorithm. It was proposed to exploit the difference in latency between read and write operations. It considers not only the hit rate but also the replacement cost incurred by selecting a dirty page as the victim. It attempts to choose a clean page as a victim rather than a dirty page by splitting the LRU list into the working region and the clean-first region and evicts clean pages preferentially in the clean-first region until the number of page hits in the working region is preserved at a suitable level. CFLRU was found to be able to reduce the average replacement cost of the LRU algorithm by 26% in the buffer cache. However, this advantage diminishes when only write requests are involved. Thus, it is not useful for enhancing the random write performance of SSDs.

FAB (Flash Aware Buffer Policy) [21] is another flash-aware buffer management algorithm. It groups pages that belong to the same erase block together. These groups are managed in an LRU manner. When any page in a group is accessed, the group is moved to the head of the LRU queue. When destaging becomes necessary, a group that occupies the most space is chosen as a victim. FAB is suitable for mobile devices where workloads are predominantly sequential.

The adaptive partitioning scheme [43] for SSD's DRAM-based cache is the only approach that adjusts the size ratio between memory spaces allocated to the write buffer and the mapping table cache. In this scheme, a ghost buffer and a ghost mapping cache collect data or mapping information destaged by the data buffer and

the mapping cache to predict whether it is beneficial to increase the size of the data buffer and decrease the size of the mapping cache or vice versa. However, this adaptive partitioning scheme treats the FTL as a black box when adjusting the size ratio, thus ignoring the important impact of the interactions among FTL, write buffer, and workload characteristics on the SSD performance. Moreover, the ghost write buffer and ghost mapping table incur non-negligible and potentially significant overhead in maintaining the metadata destaged from the real write buffer and real mapping table.

1.3 The SSD design space and the scope of the dissertation

The studies of SSD are categorized into either white-box approaches or black box approaches. The white-box approaches address problems from the inside of SSD. The research topics include FTL design, write buffer design, on-board RAM partition, endurance issues, exploiting internal parallelism and garbage collection problems. The black-box approaches view the SSD as a black-box and optimize its performance externally by identifying its performance properties and anomalies and then scheduling the SSD requests appropriately.

1.3.1 White-box approaches

Many studies in this category focus on the FTL design to ease the *erase-before-write* problem [9, 13, 14, 25, 29, 30, 33, 48, 49], by either employing multiple mapping granularities or exploiting the content similarities among the write requests [9, 14, 49]. Other popular studies involve the management of the on-board RAM inside SSD [16, 21, 23, 24, 34, 43]. The next hot topics focus on the endurance issue of SSDs.

Gokiul Soundararajan et al. [22] use HDD as a cache to increase the sequentiality of workloads and reduce the amount of data being written to the SSD. Simona Boboila et al. [5] use a reverse engineering technique to determine whether the chip or the algorithm is the key factor to determine the endurance of SSD.

In this dissertation, we present two white-box approaches to improving the performance of SSD. The first approach is the design and implementation of an on-board write buffer management algorithm. It effectively reduces the number of erasures through judicious write-buffer management. More specifically, based on our experimental observations that transaction processing and other server workloads have strong temporal locality in the form of highly repetitive updates, an important workload property that the state-of-the-art flash-aware write-buffer management schemes such as BPLRU [23] fail to exploit, we propose a new write-buffer management algorithm, called PUD-aware LRU algorithm (PUD-LRU), based on the *P*redicted average *U*pdate *D*istance (PUD) as the key block replacement criterion on top of log-block FTL schemes. Moreover, to take advantage of the characteristics of log-block FTL and increase the erase efficiency, PUD-LRU maximizes the number of valid pages in the destaged block in each erase operation.

The second approach is an on-board RAM management that exploits the interaction between the write buffer and the FTL. The goal of this approach is to avoid frequent invocation of the garbage collection process. The GC process is slow not only because the erase operation is slow, but also because a great number of pages are copied to other blocks. We propose *GC-A*ware *R*AM *M*anagement algorithm for SSD, called GC-ARM, which not only improves the GC efficiency to alleviate write amplification, but also cooperates with the FTL to minimize the address translation overhead to reduce the write-back traffic to SSD of the mapping entries. Moreover, GC-ARM can also adjust the size ratio between the write buffer and the FTL in RAM

dynamically based on the workload characteristics.

1.3.2 Black-box approaches

This category of SSD studies treat SSD as a black box to efficiently and effectively deploy SSD in the storage hierarchy and identify the performance anomalies in enterprise environment. These studies also consider whether to deploy SSD as a main storage [27, 28], or as a secondary cache above the main storage [22, 31]. For example, several emerging enterprise-level products deploy SSD as a cache device and manage it by software [10, 19, 32, 47]. There are other topics in the black-box category that exploit the internal parallelism and identify performance anomalies of SSD [2, 7, 8, 15]. Agrawal Nitin et al. [2] expose the various internal structures and operations that affect the performance of SSD and give possible solutions to users. Feng Chen et al. [7,8] reveal some unanticipated aspects in the performance dynamics of SSD. They also reveal that high-speed data processing benefit from the rich parallelism of the SSD.

In this dissertation, we conduct intensive experiments on a number of commercial SSD products from high-end PCI-E SSD to low-end SATA SSD. Based on the anomalies we identify, we obtain useful insights into SSD performance in enterprise-level environment where workloads tend to be more random and intensive. Based on the findings of the experimental evaluation study, we then provide useful suggestions to designers and developers of enterprise-level applications to make the best use of SSDs and achieve and sustain a near-peak performance while avoiding some of the intrinsic SSD problems.

Based on the parameters we find out, we further design an IO scheduler on top of SSD. We simulate the IO scheduler in the user space by converting workloads into SSD friendly ones. Event-driven evaluation shows that the IO scheduler is effective

in improving SSD performance.

1.4 Contributions of the dissertation

The SSD research topics covered in this dissertation relate to all of the components discussed in the design space. These studies either fall into white-box approaches or black-box approaches. In what follows, we list the key contributions of the dissertation that have advanced the state of the art in SSD design by significantly improving the state-of-the-art approaches in performance and endurance, and revealing and quantifying new SSD properties and performance anomalies.

Revealing the indispensability of write buffer: Our experimental study concludes that a well-designed FTL cannot replace buffer management, which in turn proves that good and novel write-buffer designs on SSD are necessary and indispensable. Based on this conclusion, we design and implement a flash-aware write-buffer management algorithm PUD-LRU that differentiates blocks and judiciously destages blocks based on their update frequency and recency so as to minimize the number of erasures while maximizing the number of valid pages in each erase operation. We find that, based on the experimental results, for sequential traces with very low temporal locality, the pure page-mapping FTL, though flexible, underperforms log-block FTL because of the fact that for such workloads the blocks to be erased contain fewer invalid pages, thus making the garbage collection of the pure page-mapping FTL very inefficient.

Revealing and quantifying the significant performance and endurance impact of low GC efficiency: The GC efficiency is determined by the average number of invalid pages in each victim block to be erased. The more invalid pages there are in an erased victim block, the more free pages the GC process can generate

and the fewer valid pages need to be copied elsewhere, thus achieving the higher GC efficiency. We expose the seriousness of the low-GC-efficiency problem by customizing workloads to change the amount of invalid data in two SSDs when collecting garbage. We show that, when an SSD suffers from low GC efficiency, the SSD write bandwidth can drop to as low as 12% of its peak, which suggests the paramount importance of improving the GC efficiency.

Designing and evaluating GC-aware write buffer management algorithm (GC-ARM): A low GC efficiency implies that more valid pages must be copied from victim blocks to free blocks in the GC process, which constitutes the *GC-induced write traffic*. To address this problem, GC-ARM dynamically destages either contiguous pages in a block as a whole or a single page from the write buffer based on the benefit the two destaging schemes provide to improve GC efficiency.

Adaptive and dynamic partition of the RAM between write buffer and FTL mapping table: Based on our experimental observations revealing that the number of distinctive erase blocks accessed for different workloads vary dramatically over time, GC-ARM is designed to dynamically adjust the ratio of the RAM memory space allocated between the write buffer and the cache for the mapping table to further improve the performance of SSD by adapting to the characteristics of workloads.

Minimizing the write-back traffic resulting from mapping-entry replacement in RAM: GC-ARM enables the write buffer to effectively interact with the FTL that groups mapping entries based on the logical page number'(LPN) spatial locality. Based on this design, GC-ARM is able to minimize the write-back traffic induced by victim mapping entries replaced from the cached mapping table and reduce address-translation overhead.

Surmising the FTL mapping policy: In order to improve the GC efficiency by wisely allocating the space of SSD, it is necessary to surmise the mapping policy

of a given SSD because different mapping policies have different behavior under write-intensive workloads. Thus, we determine if it is possible to deduce the mapping policy a given SSD employs by examining the responses of the SSD to a given workload.

Estimating the residual lifetime of an SSD: Based on our experimental evaluation results, we develop an analytical model to estimate the residual lifetime of a given SSD. We believe that the model will be useful to the potential users of SSD in the enterprise-level application environment.

Determining whether random read is as fast as believed: The relatively high random-read performance is one of the advantages of SSD. While this is proven to be true in general, we are able to determine whether there are exceptions to the norm, the circumstances under which these exceptions occur, and their implications.

Finding a tradeoff between bandwidth, queue length and average response time: While it is possible to achieve the peak bandwidth by increasing the queue length of requests, we observe that the average response time increases proportionally. We further observe that the average response time is sensitive to the request size. Our investigation based on the experimental evaluation attempts to determine the optimal combination of queue length of requests and request size that achieve the peak bandwidth while minimizing the average response time.

Designing an SSD-friendly IO scheduling algorithm: Based on the parameters we identified in the experiment, we develop an IO scheduling scheme that converts workloads to SSD-friendly ones. Trace-driving experimental results show that it is effective in improving SSD performance.

1.5 Dissertation Organization

This dissertation is organized as follows. In Chapter 2, we introduce PUD-LRU, an SSD-friendly write buffer management algorithm. We first discuss enough background to motivate our work and then give more details about the design and evaluation.

In Chapter 3, we introduce GC-ARM, the on-board RAM management algorithm that focuses on the overall design of buffer, FTL and RAM partitioning between the two. We first give enough background to highlight the importance of the RAM management and then give the detailed design and evaluation of GC-ARM.

In Chapter 4, we evaluate several new SSD products to identify performance anomalies in enterprise environment. Giving the fact that of more and more SSDs are being deployed in enterprise environment, identifying performance anomalies becomes increasingly important. We will explain and analyze based on the experimental results from these SSDs. Further, we give guidelines and suggestions to both SSD vendors and system programmers based on our results.

In Chapter 5, we design and implement an SSD-friendly IO scheduling scheme based on the parameters drawn from Chapter 4. State-of-the-art research of IO scheduling is presented as a background and we evaluate our scheme by trace-driven experiments.

Chapter 6 presents the related work of SSD. We categorize the state-of-the-art related research into several categories.

The dissertation is concluded in Chapter 7 and future work is presented.

Chapter 2

Design and Implementation of a Write Buffer Management Algorithm of SSD

The unique SSD feature of *erase-before-write* imposes a real challenge to the performance and longevity of flash memory SSD. As has been discussed in the Introduction section, there are two general approaches to addressing this challenge at different levels, namely, by means of sophisticated design of address mapping modules in the Flash Translation Layer (FTL) and flash-aware write-buffer management. We find in our experimental study that flash-aware buffer management is necessary and important. Therefore, this chapter focuses on the latter approach to effectively reduce the number of erasures through judicious write-buffer management. More specifically, based on our experimental observations that transaction processing and other server workloads have strong temporal locality in the form of highly repetitive updates, an important workload property that the state-of-the-art flash-aware write-buffer management schemes such as BPLRU [23] fail to exploit, we propose a new write-buffer

Table 2.1: A performance comparison between BPLRU and page-mapping FTL

Approach	Number of erasures	Latency (ms)
BPLRU	22908	3.9
Page-mapping FTL	51373	7.9

management algorithm, called PUD-aware LRU algorithm (PUD-LRU), based on the *P*redicted average *U*pdate *D*istance (PUD) as the key block replacement criterion on top of log-block FTL schemes. Moreover, to take advantage of the characteristics of log-block FTLs and increase the erase efficiency, PUD-LRU maximizes the number of valid pages in the destaged block in each erase operation. This work is published in [16].

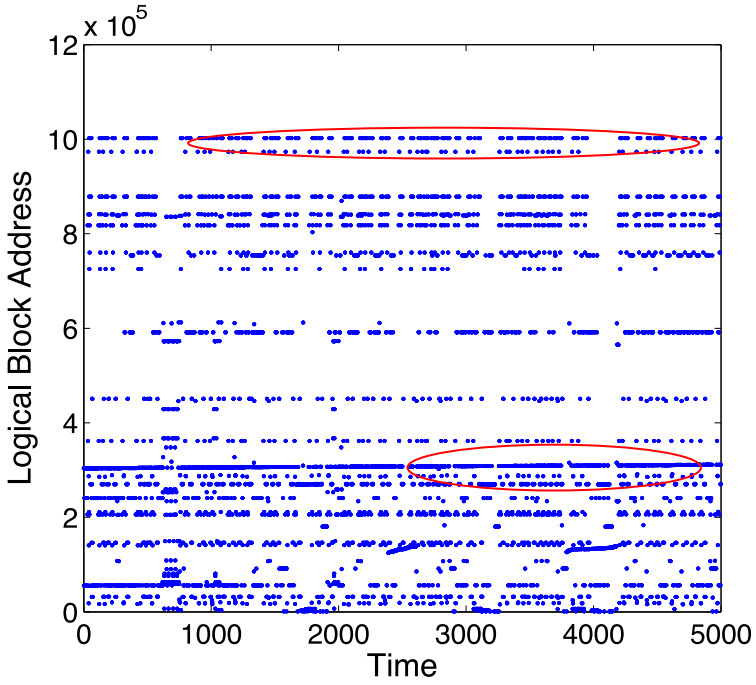
2.1 Background

Our experimental results show that for some workloads, a well-designed buffer management algorithm that is flash-aware and log-block-FTL-aware can outperform the pure page-mapping FTL, thus showing that the buffer management is necessary and indispensable. Table 2.1 shows the performances of BPLRU on top of the log-block FTL algorithm [25] and the pure page-mapping FTL without any write buffer, running on a 32GB SSD with a page size of 4KB and under the Financial 1 [39] workload. Assuming that each mapping entry is 4 bytes, then the page-mapping FTL requires at least 32MB to store the page-mapping table. On the other hand, we assume 1% of the total SSD blocks in the log-block FTL scheme is reserved as log blocks. So the log-block FTL scheme BAST requires 0.56MB for storing the block-mapping table and the page-mapping information for the log blocks while leaving 31.44MB RAM as the write buffer for BPLRU. The results shown in Table 2.1 suggest to us that designing an efficient buffer management algorithm will be a very worthwhile and

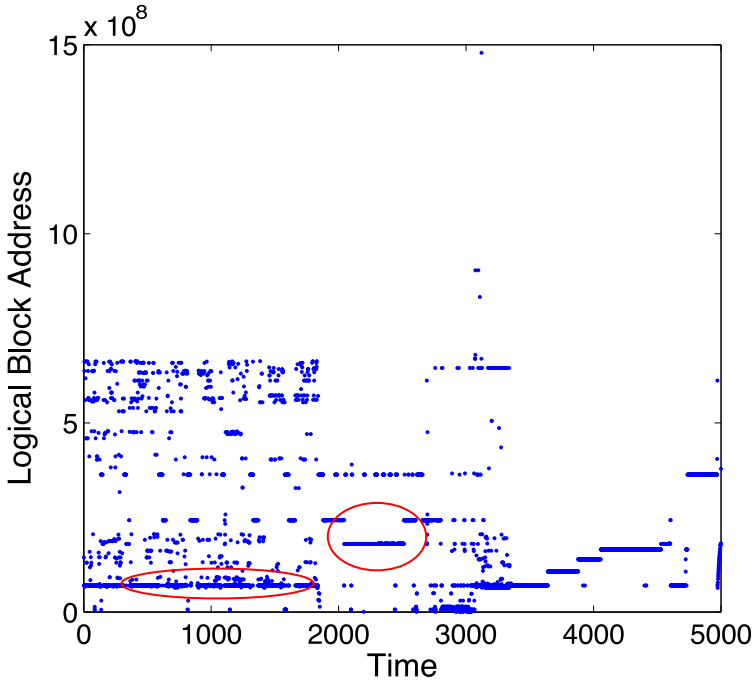
necessary effort that is likely to complement, if not substitute, a good page-mapping FTL scheme.

Furthermore, results of our experiments with several representative traces, shown in Figure 2.1(a) and Figure 2.1(b), reveal that some OLTP and server workloads exhibit strong temporal locality. In these figures, each request corresponds to a dot. With only $5,000$ requests from the beginning of the traces analyzed, it is amply evident that frequent and repeated update requests are very common in these traces. The ovals in these figures indicate where frequently and repeatedly updated requests take place. We also measured in the experiments how many requests are updated more than once. Table 2.2 shows the characteristics of the five traces that we use in the evaluation section. For the Financial 1 trace, we found that more than 30% of the requested addresses are updated more than once. For the Financial 2 trace, more than 40% of the requested addresses are updated more than once. The Exchange trace also includes a lot of frequently updated requests, where 35% of the requested addresses are updated more than once. The Build trace does not have so many frequent update requests, with only 13% requested addresses being updated more than once. Table 2.2 also indicates that some traces, such as TPC-E, do not exhibit temporal localities. We observe that a large proportion of frequent update operations tend to cluster around small portions of the storage address space in these traces and these addresses are updated extremely frequently. Thus, we believe that, if we can identify frequently updated blocks and buffer them in the write buffer, the number of erasures can be greatly reduced.

We also observe that another factor affecting the number of erasures in SSD is the number of valid pages involved in each destaging operation. For example, if only 32 pages of a 64-page flash erase block are valid and another 32 pages are padded, 32 pages are wasted for this destaging operation. On the other hand, if a data block



(a) Financial 1



(b) Build

Figure 2.1: Analysis of temporal locality of two workloads

Table 2.2: I/O characteristics of workloads studied

trace	description	average request size(KB)	write/read ratio	IOPS	updated more than once
Financial 1 [39]	an OLTP application	3.38	3.31	122	30%
Financial 2 [39]	an OLTP application	2.38	0.24	54	40%
Exchange [36]	Microsoft exchange server	12.5	1.53	611	35%
Build [36]	Microsoft build server	4.14	41	1980	13%
MSN [38]	MSN storage server	22.52	66.7	106	9%
TPC-E [37]	TPC-E benchmark	12.75	83.9	308	1%

with all of its 64 pages being valid is chosen to be destaged, then there is no need to destage another data block.

These important observations, combined with our study of the existing flash-aware buffer management algorithms, motivate us to propose the PUD-LRU algorithm, based on *P*redicted Average *U*pdate *D*istance (PUD) whose detailed definitions are given in next section. The main idea behind PUD-LRU is to differentiate blocks in the buffer based on their PUD values and judiciously destage blocks in the buffer according to their PUD values so as to minimize the number of erasures while maximizing the number of valid pages in each erase operation. More specifically, PUD-LRU organizes the buffer space into blocks of the erase-block size in the same manner as BPLRU. Furthermore, it divides blocks in the buffer into two groups based on their PUD values. The first group contains blocks with smaller PUD values and thus is defined as *F*requently *U*pdated *G*roup (FUG). The second group contains blocks with larger PUD values and thus is defined as *I*nfrequently *U*pdated *G*roup (IUG). When a free block is needed, PUD-LRU chooses a block that contains the most valid pages from IUG to destage, thus destaging as many valid pages as possible while avoiding destaging a block that will be updated soon, which in turn maximizes the efficiency of the destaging operation. The design goal of PUD-LRU is to decrease the number of erasures and thus improve the longevity and performance of SSDs.

2.2 Design and implementation

As shown in Figure 2.2, a PUD-LRU-based SSD is composed of three main components, namely, the PUD-LRU buffer management module, the Flash Translation Layer (FTL) and the physical flash memory. PUD-LRU sits on top of FTL inside the SSD, monitors and processes the I/O requests issued by the upper file system or database system. Similar to BPLRU, the proposed PUD-LRU is designed on top of a log-block FTL, and uses most of the RAM memory space as a write buffer since the upper storage cache will absorb most of the popular read requests. A very small fraction of the RAM space is used for the block-mapping table and page-mapping for the log blocks. It divides the write buffer into blocks of erase-block size and further groups these blocks into two groups, the frequently updated group FUG and the infrequently updated group IUG as shown in Figure 2.2, based on their PUD values, to be defined later. A read request from the standard block interface is first directed to the PUD-LRU write buffer to search in both groups. If it does not hit, the read request is sent to FTL to be serviced there. Similar to the read request, a write request from the standard block interface is first directed to the PUD-LRU write buffer. If it hits in either of the two groups, the PUD-LRU algorithm will update the metadata in the hit block. On the other hand, if it does not hit in either of the groups and the PUD-LRU write buffer is full, destaging must be performed by choosing a block with the most valid pages from the group whose blocks are not frequently updated, namely, IUG. In order to reduce the number of erasures, PUD-LRU employs the page-padding scheme of BPLRU [24], which reads some pages from the flash memory to pad the block to be destaged as a whole, which reduces the number of erasures by one. The destaging operation makes room for buffering the newly arrived write request that missed at the buffer. If the buffer is not full, a missed write request will be allocated a free

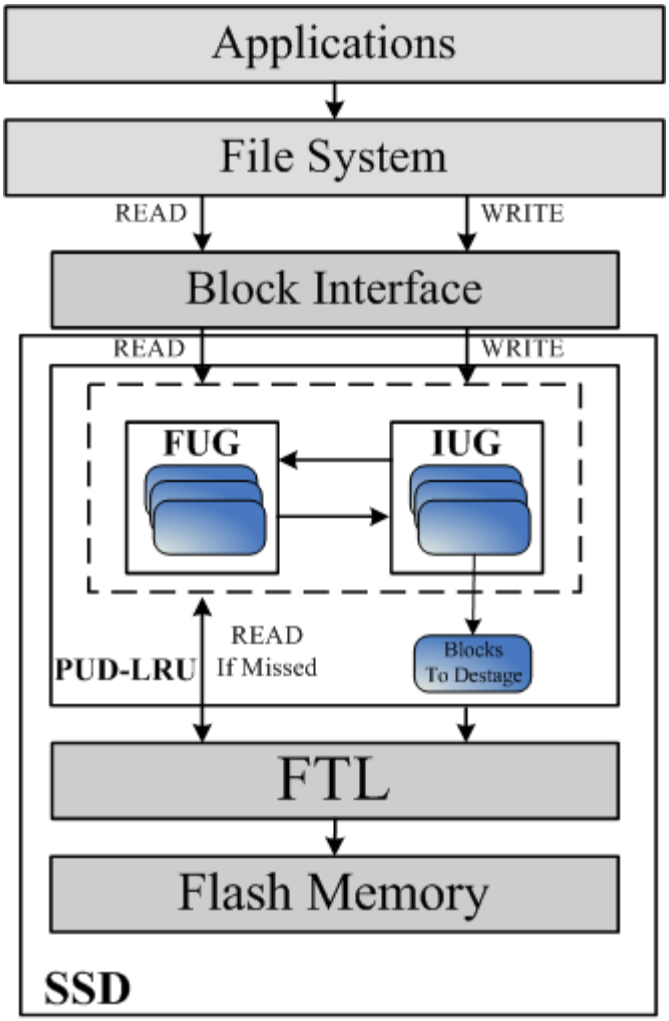


Figure 2.2: An architectural view of the PUD-LRU-based SSD

block in the write buffer and group membership of the newly allocated free block will be determined by its PUD value.

2.2.1 Predicted Average Update Distance

*P*redicted Average *U*pdate *D*istance (PUD) plays a key role in our PUD-LRU algorithm design that leverages the measure of PUD to determine which blocks to keep in or evict from the buffer when the buffer is full and a newly arrived write request

needs a free block to be allocated in the buffer. First, we define the *U*pdate *D*istance (UD) between two update operations to a block as the number of blocks between these two update operations in the update sequence. For example, "ABCDA" represents a sequence of update operations to blocks labeled "A", "B", "C", and "D", and UD between the two consecutive updates to block A is 3. Equation 2.1 formally defines UD of a block. In this equation, $d_j(i)$ represents the position of block j's ith appearance in the update sequence.

$$UD_j(i) = d_j(i+1) - d_j(i) - 1 \tag{2.1}$$

For example, given an update sequence of "ABXAXAXBXAXXXXB", where "X" represents a block other than block A or B, $d_A(0)$, $d_A(1)$, $d_A(2)$, and $d_A(3)$ are 0, 3, 5, and 9, respectively, and $d_B(0)$, $d_B(1)$, and $d_B(2)$ are 1, 7, and 14, respectively.

To identify the frequently updated blocks, we define *average UD*, denoted as $\overline{U_j}$, in Equation 2.2 below. In this equation, n represents the number of times block j is updated in an update sequence.

$$\overline{UD_j} = \frac{\sum_{i=0}^{n-1} UD_j(i)}{n-1} \tag{2.2}$$

A small $\overline{UD}$ of a block implies that the block is frequently updated. Otherwise, it is infrequently updated. For example, $\overline{UD_A}$ in the above example is 2 and $\overline{UD_B}$ is 5.5. So block A is updated more frequently than block B, suggesting that block A should not be destaged because updating block A will likely incur more erase operations than B. On the other hand, *recency* also needs to be taken into consideration in deciding a block's fate in the write buffer. It is possible that a block was updated frequently some time ago and will no longer be updated in the near future. In this case, we

also need to determine when to destage this kind of blocks. Considering the above example again, block A has a higher update frequency but a lower update recency than block B. To take *both recency and frequency* into consideration, we first define a measure of recency in our algorithm, called *R*ecency *D*egree (RD), as the number of blocks between the last update operation of a block and the latest update operation of any block. Clearly, similar to the $\overline{UD}$ measure, the lower the RD value of a block is, the more recently this block has been updated. In other words, a low $\overline{UD}$ (or RD) value of a block implies a high update *frequency* (or *recency*) of this block. For example, RD of block A in the example is 5 while RD of block B is 0, implying that B is more recently updated than A.

To take both of these frequency and recency measures into our consideration, we define the *P*redicted $\overline{UD}$ (PUD) of block j as PUD_j, which is shown in Equation 2.3 that weighs each measure equally.

$$PUD_j = \frac{\overline{UD}_j + RD_j}{2} \tag{2.3}$$

The intuition behind Equation 2.3 is that blocks with small PUD values tend to be updated more frequently within a short and recent period of time, and thus should be kept in the buffer for a longer time than other blocks. On the other hand, blocks that have larger PUD values are either less frequently updated or have a high RD value. Therefore, blocks with larger PUD values should have a higher preference of being destaged.

2.2.2 Destaging Efficiency

To minimize the number of erasures, it is important for each destaging operation to involve as many valid pages in one erase block as possible. However, it is possible that

Table 2.3: An illustrative comparison between PUD-LRU and BPLRU

page writes	BPLRU		PUD-LRU		
	buffer status	victim block	buffer status	victim block	update_counter
0	[0]		[0](1,0,0)		0
4	[0], [4]		[0](1,0,0), [4](1,1,0)		1
1	[4], [0,1]		[0,1](2,2,1), [4](1,1,0)		2
8	[4], [0,1], [8]		[0,1](2,2,1), [4](1,1,0), [8](1,3,0)		3
2	[4], [8], [0,1,2]		[0,1,2](3,4,2), [4](1,1,0), [8](1,3,0)		4
12	[4], [8], [0,1,2], [12]		[0,1,2](3,4,2), [4](1,1,0), [8](1,3,0), [12](1,5,0)		5
3	[8], [12], [0,1,2,3]	[4]	[4](1,1,0), [8](1,3,0), [12](1,5,0), [3](1,6,0)	[0,1,2]	6
4	[12], [0,1,2,3], [4]	[8]	[4](2,7,5), [8](1,3,0), [12](1,5,0), [3](1,6,0)		7
0	[12], [4], [0,1,2,3]		[4](2,7,5), [8](1,3,0), [12](1,5,0), [0,3](2,8,1)		8
16	[4], [0,1,2,3], [16]	[12]	[4](2,7,5), [8](1,3,0), [12](1,5,0), [0,3](2,8,1), [16](1,9,0)		9
1	[4], [16], [0,1,2,3]		[4](2,7,5), [8](1,3,0), [12](1,5,0), [16](1,9,0)	[0,3]	10
17	[16], [0,1,2,3]	[4]	[4](2,7,5), [8](1,3,0), [12](1,5,0), [16,17](2,11,1)		11

the block that has the most valid pages will be accessed very soon, so it is necessary to choose blocks that will not be updated soon and have the most valid pages to destage. Therefore, as mentioned earlier, our PUD-LRU scheme divides blocks in the buffer into two separate groups, the *F*requently *U*pdated *G*roup (FUG) and the *I*nfrequently *U*pdated *G*roup (IUG). Because blocks in IUG will not be updated soon, the number of valid pages in a block in IUG becomes important. When a free block is needed and the buffer is full, PUD-LRU chooses a block that has the most valid pages from IUG for replacement. More specifically, PUD-LRU first calculates the range of these blocks' PUD values. If the PUD value of a block is within a given threshold, 0.1% in our current simulation study, of the range, the block is considered a member of FUG, otherwise it becomes a member of IUG. The reason for choosing 0.1% as the threshold lies in the fact that only a relatively small portion of the request address space is very frequently accessed [12], as evidenced in our sensitivity study of

PUD-LRU detailed in the evaluation section. As explained in that section, there is no single fixed threshold that will possibly optimize all cases since the optimal threshold value varies with both the workload and the buffer size. By grouping blocks, our PUD-LRU scheme chooses the block that has the most valid pages from IUG with a large PUD value, thus making each destaging operation more efficient.

2.2.3 Implementation of PUD-LRU

In order to calculate the PUD value for each block and place the block into either FUG or IUG, PUD-LRU employs three integers to keep track of the update information for each block. The first integer tracks the position where a block was updated the last time. The second integer represents the summation of UD of a block over all updates to this block (i.e., Equation 2.1). The third integer records the number of times a block is updated, which is increased by one each time the block is updated. Our current implementation uses 4B for each of these integers. The space overhead due to these three integers per block, at 3×4B=12B per 512KB block or 0.002%, is negligible. For each request, the PUD-LRU module executes the PUD-LRU algorithm whose pseudo code is shown in Algorithm 1. In order to calculate the PUD value when necessary, PUD-LRU defines a global variable *update_counter* to indicate how many blocks have been updated. $frequency_j$ denotes the number of times block j is updated. $last\_update_j$ stores the previous global *update_counter* in block j when block j is updated some time ago. So when block j is updated again, PUD-LRU can calculate the summation of UD of block j(UD_j) by just increasing UD_j by the difference between $last\_update_j$ and the global variable *update_counter*. From Algorithm 1, we can see that PUD-LRU recalculates the PUD value of each block only when a block replacement is required. The time complexity of the algorithm is therefore

Algorithm 1 PUD-LRU Algorithm

Initialize: $update\_counter \leftarrow 0$
Input: Request R with logical block number R_{lba}**, request size** R_{size} **and request type** R_{type}
if $R_{lba} \equiv read$ **then**
  $offset \leftarrow 0$
  while $offset < R_{size}$ **do**
    for all blocks, search logical block number $R_{lba} + offset$
    if hit **then**
      return
    else
      send logical block number $R_{lba} + offset$ to FTL
    end if
    $offset \leftarrow offset + 1$
  end while
else {/*$R_ttype \equiv write$*/}
  $offset \leftarrow 0$
  while $offset < R_{size}$ **do**
    $update\_counter \leftarrow update\_counter + 1$
    for all blocks, search logical block number $R_{lba} + offset$
    if hit $block_j$ **then**
      $UD_j \leftarrow UD_j + update\_counter - last\_update_j - 1$
      $frequency_j \leftarrow frequency_j + 1$
      return
    else {/*does not hit any block in the buffer*/}
      if need replacement **then**
        $size \leftarrow number\_of\_blocks\_in\_the\_buffer$
        for $i \leftarrow 1$ to size **do**
          $PUD_j \leftarrow \frac{update\_counter + UD_j}{frequency_j}$
        end for
        $PUD\_RANGE \leftarrow get\_pud\_range()$
        {/*calculate the value range among all blocks in the buffer*/}
        for $i \leftarrow 1$ to size **do**
          if $PUD_i < \frac{1}{1000} \times PUD\_RANGE$ **then**
            group $block_i$ into FUG
          else
            group $block_i$ into IUG
          end if
        end for
        $victim \leftarrow block\_with\_most\_valid\_page()$
        send $block_{victim}$ to FTL
      else {/*the buffer still has space*/}
        allocate a new block k
        $last\_update_k \leftarrow update\_counter$
        $UD_k \leftarrow 0$
        $frequency_k \leftarrow 1$
      end if
    end if
    $offset \leftarrow offset + 1$
  end while
end if

$O(n)$, where n is the number of blocks in the write buffer. Because the buffer size is very limited, n is generally small.

Table 2.3 gives an illustrative example comparing how PUD-LRU and BPLRU process a page-write sequence. In this example, we assume, for simplicity and without loss of generality, that a block contains four pages and the buffer can store 6 pages at most. The page-write request sequence is 0, 4, 1, 8, 2, 12, 3, 4, 0, 16, 1, 17, which corresponds to the block sequence 0, 1, 0, 2, 0, 3, 0, 1, 0, 4, 0, 4. Page numbers are put in square brackets to represent a block. In the case of PUD-LRU, the three variables in the parentheses of a block j indicate the three integers $frequency_j$, $last\_update_j$ and UD_j, as shown in Algorithm 1, respectively. The last column of Table 2.3 shows the value of the global variable *update_counter*. The three variables are only updated when block j is hit. In this case, $frequency_j$ is increased by one, UD_j is set to $update\_counter - last\_update_j - 1 + UD_j(old)$ and $last\_update_j$ is set to the global *update_counter*.

Because both PUD-LRU and BPLRU employ a page-padding scheme, a destaging operation leads to one erase operation. For BPLRU, four blocks are destaged, which leads to four erase operations. For PUD-LRU, when the *update_counter* is 6, a destaging operation is initialized. The PUD values for blocks [0,1,2], [4], [8], [12] are 1, 2, 1, 0 respectively. In this example, the threshold is set to $\frac{1}{100}$, which groups blocks [0,1,2], [4], [8] to IUG while grouping block [12] to FUG. Block [0,1,2] is chosen as a victim because it contains the most valid pages among the three blocks in IUG. Similarly, when the *update_counter* is 10, block [0,3] is chosen as the victim because it belongs to the IUG group and contains the most valid pages. PUD-LRU requires two destaging operations in this example scenario, which leads to only two erase operations.

2.2.4 Data integrity concern

Data in the write buffer and mapping entries in RAM may be lost if there is a sudden power failure, giving rise the data integrity concern. Several existing technologies can be leveraged to ensure data integrity. For example, SandForce SF1500 [1], which is an enterprise-level SSD product, employs a super-capacitor as an insurance policy to guarantee data integrity. Similar data-integrity protection technologies, such as battery-backed RAM, continuous data protection (CDP), etc., may also be employed.

2.3 Performance evaluation

2.3.1 Experimental setup

In our trace-driven simulation study of PUD-LRU, we simulate a 32GB SSD, with a page size of 4KB and a block size of 512KB based on FlashSim [53]. FlashSim is an event-driven simulator designed to simulate flash memory SSDs. FlashSim can simulate multiple planes, dies, and packages for parallelism effects. We first extended FlashSim by implementing a log-block FTL [25], which employs 1% of the blocks as log blocks. Then we implemented both the PUD-LRU and BPLRU write-buffer management algorithms on top of the log-block FTL to simulate an SSD integrated with a write buffer. We then implemented DFTL [13] on top of FlashSim that consumes the same resources as those by BPLRU and PUD-LRU. We also implemented the pure page-mapping FTL that stores its entire page-mapping table in the RAM, which requires 64MB RAM space to store its mapping table for a 32GB SSD in our experiment, assuming that each table entry occupies 8-byte RAM space.

Our simulator takes block-level I/O traces as input and generates as output the number of erasures and average response time for a request after the simulation.

Table 2.4: Configuration parameters of SSD

Operation	Latency
Page read	20 us
Page write	200 us
Block erase	1.5 ms
Page read delay	25 us
Page write delay	200 us

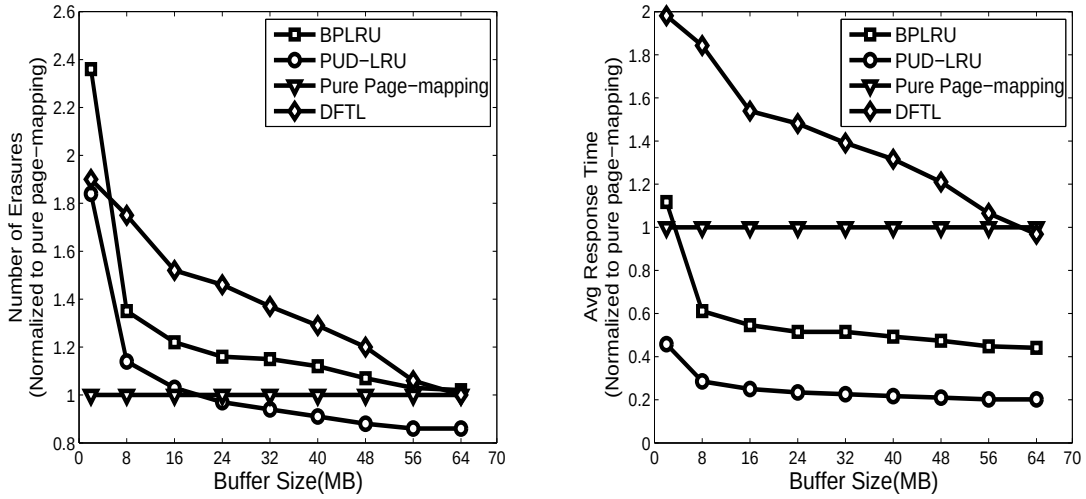
The number of erasures and average response time are used as the endurance and user performance metrics to evaluate the effectiveness of PUD-LRU against BPLRU, DFTL and pure page-mapping FTL.

We fed five traces to the simulator to evaluate and compare the number of erasures and average response time among PUD-LRU, the state-of-the-art write-buffer management algorithm, BPLRU, the sate-of-the-art demand-based page-mapping FTL, DFTL, and the pure page-mapping FTL. The five traces are Financial 1 [39], Financial 2 [39], Microsoft Exchange Server [36], Microsoft Build Server [36], TPC-E benchmark collected at Microsoft [37], whose key I/O characteristics are summarized in Table 2.2. The SSD in our experiment is configured with its key parameters listed in Table 2.4.

2.3.2 Number of erasures and average response time

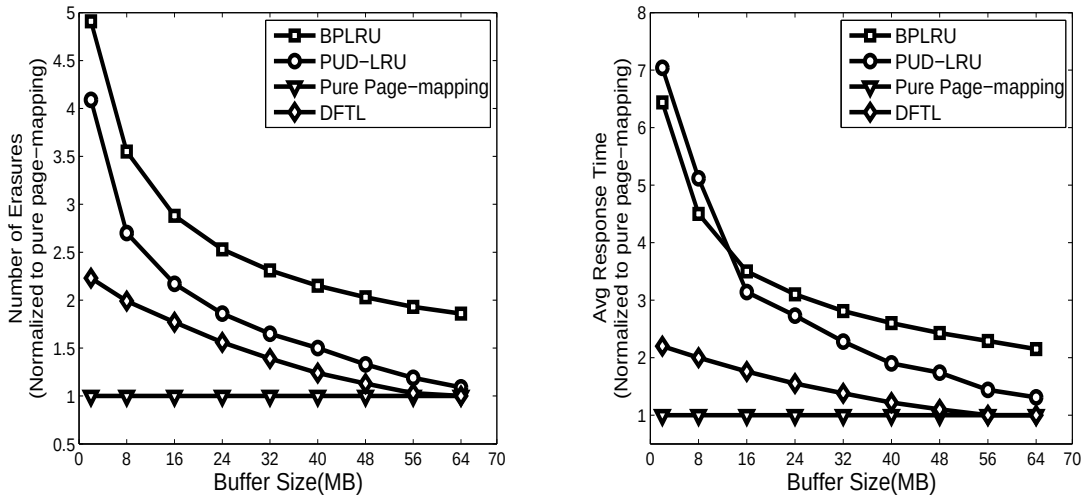
Figure 2.3 through Figure 2.7 plot the number of erasures and average response time respectively as a function of the RAM size. Both PUD-LRU and BPLRU employ the RAM as a write buffer. On the other hand, DFTL employs the RAM as cached mapping table. The pure page-mapping FTL consumes 64MB RAM and is plotted as a straight line. All results are normalized to the pure page-mapping FTL.

For the Financial 1 trace, PUD-LRU reduces the number of erasures and average response time of BPLRU by up to 30% and 56% respectively, and those of DFTL



(a) (b)

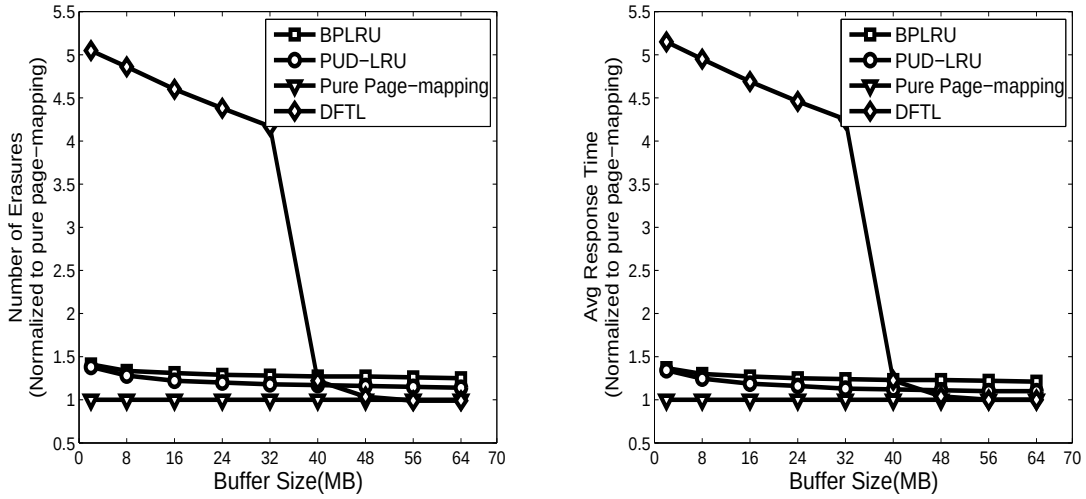
Figure 2.3: Number of erasures and avg response time of Financial 1 workload



(a) (b)

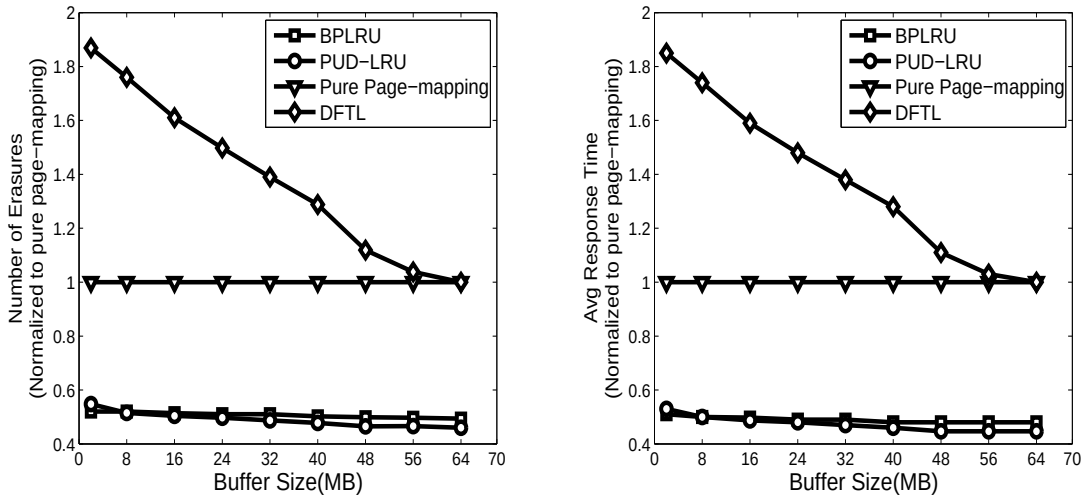
Figure 2.4: Number of erasures and avg response time of Financial 2 workload

by up to 33% and 32% respectively. PUD-LRU outperforms the pure page-mapping FTL in the number of erasures measure when the RAM size is more than 24MB and is consistently better than the pure page-mapping FTL in the average response time measure. The Financial 2 trace has a more pronounced frequent update pattern, thus



(a) (b)

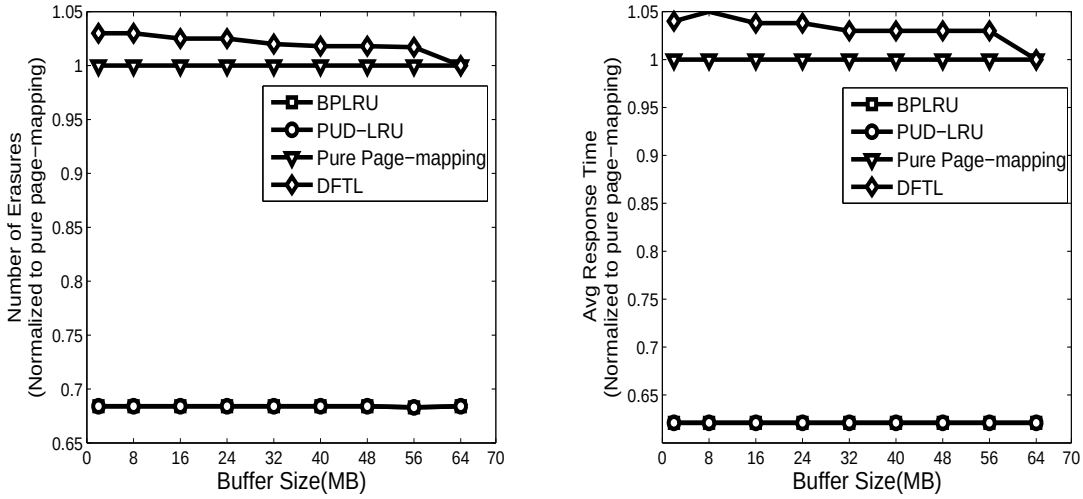
Figure 2.5: Number of erasures and avg response time of Exchange workload



(a) (b)

Figure 2.6: Number of erasures and avg response time of Build workload

exposing more temporal locality to be exploited. As a result, PUD-LRU improves on the number of erasures and average response time over BPLRU by up to 42% and 39% respectively. Meanwhile, DFTL also performs well based on the fact that the Financial 2 trace exhibits strong temporal locality. The hit rate of the cached



(a) (b)

Figure 2.7: Number of erasures and avg response time of TPC-E workload

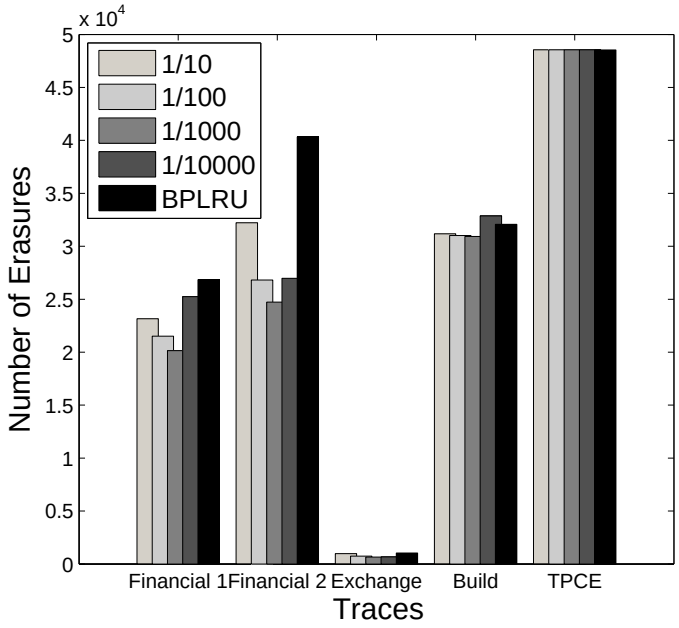
mapping table is as high as 53% for DFTL when the RAM size is 16MB. Although the Exchange trace has strong repeated update pattern, as shown in Table 2.2, the repeated updated logical addresses are sparsely scattered. As a result, PUD-LRUs advantage over BPLRU in these two measures is much lower, with improvements of only 7% and 5% respectively. Moreover, PUD-LRU decreases the number of erasures and average response time of DFTL by up to 73% and 75% respectively when the RAM is smaller than 40MB. However, when the RAM is larger than 40MB, both the number of erasures and average response time drop dramatically and are comparable to BPLRU, PUD-LRU and the pure page-mapping FTL. The reason for this lies in the fact that the working-set of the Exchange trace can be included in the 40MB cached mapping table. The Build trace exhibits much less of a frequent-update pattern than the Financial 2 and Exchange traces, lowering PUD-LRU's advantages over BPLRU in these two measures to just up to 7% and 8% respectively. For the TPC-E trace that offers very little, if any, temporal locality to be exploited by PUD-LRU, however, PUD-LRU performs almost identically to BPLRU as expected, as shown in Figure

2.7. It is noted that for both the Build and TPC-E traces, the pure page-mapping FTL underperforms both PUD-LRU and BPLRU significantly. The Build trace is relatively random, which leads to very low efficiency of the garbage collection of the pure page-mapping FTL because each erased block contains few invalid pages and more valid pages need to be copied to another block. On the other hand, the TPC-E trace is sequential and exhibits little temporal locality, thus leading to frequent switch merges for both BPLRU and PUD-LRU without page-padding. Since each page can only be written once before the entire block is erased, switch merge is optimal for SSD.

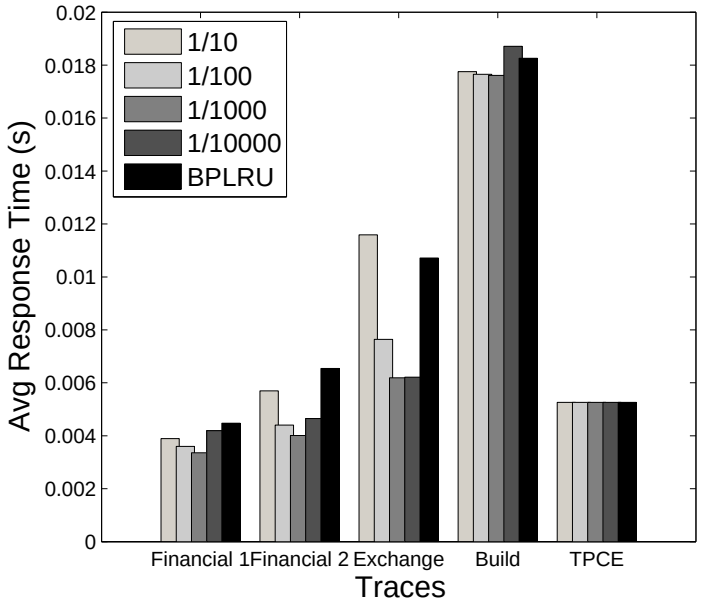
2.3.3 Sensitivity study

In the evaluation of the number of erasures and average response time above, we choose to group blocks whose PUD values are less than 0.1% of the PUD value range into FUG. In other words, the default threshold value for dividing blocks in the buffer into FUG and IUG is 0.1% in our current PUD-LRU implementation. In order to study the sensitivity of the threshold to the performance of PUD-LRU in terms of destaging efficiency, we compare the number of erasures and average response time based on different threshold values. The write-buffer size is set to 16MB.

The experimental results, shown in Figures 2.8(a) and 2.8(b), demonstrate that for the Financial 1 and Financial 2 traces, PUD-LRU consistently outperforms BPLRU for all the threshold values. For the Exchange and Build traces, PUD-LRU shows the best performance when the threshold is set properly. The performance does not change for the TPC-E trace when the threshold value changes because TPC-E trace shows a significant sequential write pattern. The results clearly show that for each workload an optimal threshold value exists that minimizes both the number of



(a) The number of erasures based on different threshold



(b) Average response time based on different threshold

Figure 2.8: Number of erasures and average response time based on different threshold

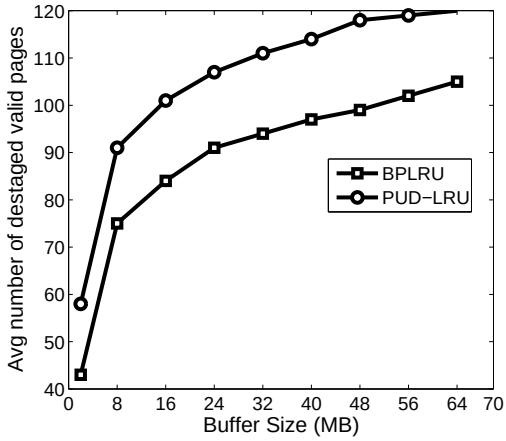
erasures and the average response time. For the Financial 1, Financial 2, Exchange, and Build traces PUD-LRU achieves the best performance for the two measurements when the threshold is $\frac{1}{1000}$. We also observe that different workloads have different sensitivity to the threshold. For example, the performance of the TPC-E trace does not change too much when the threshold changes, implying its insensitivity to the threshold. The reason lies in the fact that the trace does not exhibit much of a frequent-update pattern, as shown in Table 2.2.

From the analysis based on results shown in Figure 2.8, it is clear that there exists an optimal threshold value for different workloads. Furthermore, our experimental results and analysis also indicate that the optimal threshold value is dependent upon the buffer size. Therefore, ideally, this threshold should be tuned dynamically to adapt to different workloads and buffer sizes. In the absence of such a dynamic and adaptive mechanism to tune the threshold, in our current evaluation study we set the threshold to $\frac{1}{1000}$ for all traces and buffer sizes for the sake of simplicity and ease of implementation. Therefore, as a topic of future research on PUD-LRU, we plan to devise the aforementioned dynamic and adaptive tuning mechanism for the threshold.

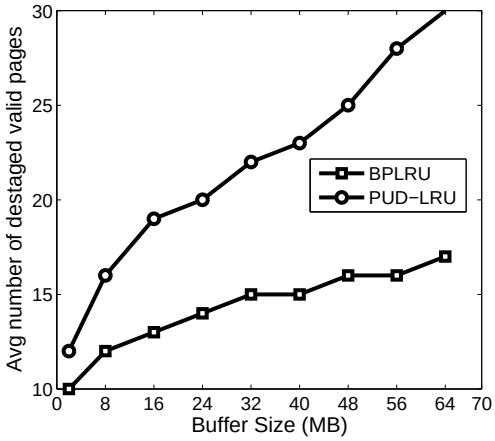
2.3.4 Destaging efficiency

PUD-LRU uses the above threshold to group blocks into FUG and IUG. A block with the most valid pages is chosen as a victim in order to make the destaging operation as efficient as possible. In order to show the destaging efficiency, we compare the average number of valid pages in each destaging operation. The threshold is set to $\frac{1}{1000}$ for all the five traces.

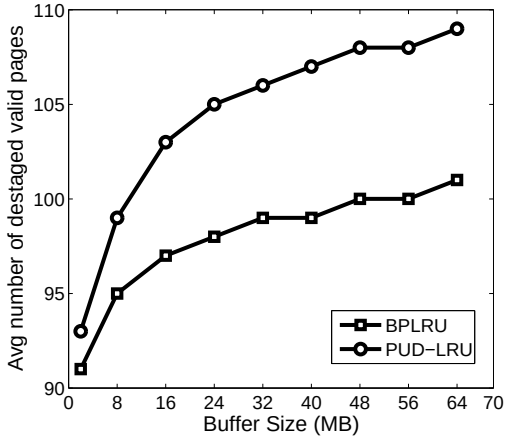
As shown in Figure 2.9 that plots the number of valid pages as a function of the buffer size, the number of valid pages increases monotonously, and more quickly for



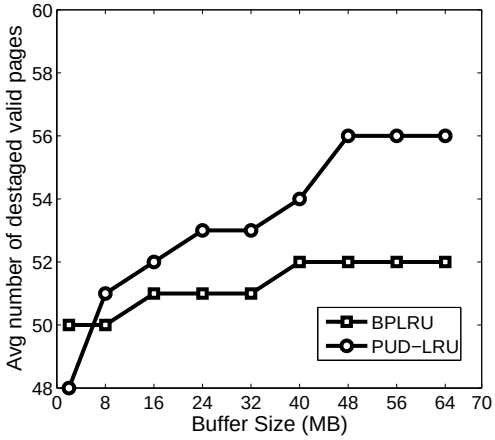
(a) Destaging efficiency of Financial 1



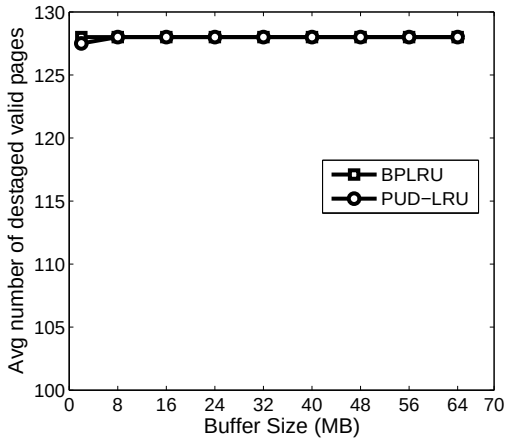
(b) Destaging efficiency of Financial 2



(c) Destaging efficiency of Exchange



(d) Destaging efficiency of Build



(e) Destaging efficiency of TPC-E

Figure 2.9: Destaging efficiency of five workloads

PUD-LRU than for BPLRU under all the traces except for the TPC-E trace. On the other hand, TPC-E exhibits a very sequential write pattern, thus destaging almost 128 pages in each destaging operation. PUD-LRU further increases the number of valid pages destaged every time, by using the number of valid pages, or destaging efficiency, as one of its replacement criteria explicitly, in addition to frequency and recency, while BPLRU does not consider destaging efficiency explicitly in its replacement policy. It is thus possible that in BPLRU the destaged block contains a small number of valid pages, which makes erase operations and page padding very inefficient.

2.3.5 Summary

In this chapter we first demonstrated that typical server and online transaction processing workloads exhibit strong temporal locality based on the observation that a large percentage of the requested addresses are updated repeatedly in the relevant traces. We further showed that buffer management plays an important role in improving the performance of SSDs and cannot be replaced by sophisticated FTLs based on our experimental study. To fully exploit the temporal locality and increase the destaging efficiency, we propose a new erase-efficient write-buffer management algorithm for SSDs, called PUD-LRU, that groups blocks in the buffer into two groups based on a combined measure of frequency and recency, Predicted average Update Distance (PUD). PUD-LRU chooses a block to destage from the group that is considered less frequently and less recently updated (i.e., the group containing high PUD-valued blocks), such that the chosen block has the most valid pages in it. By doing so our scheme maximizes the efficiency of each destaging operation. We evaluated the effectiveness of our PUD-LRU scheme through an extensive trace-driven simulation study that compares PUD-LRU with the state-of-the-art buffer management scheme

BPLRU, the state-of-the-art page-mapping DFTL and the pure page-mapping FTL scheme in terms of the number of erasures and average response time. The results show that PUD-LRU significantly and consistently outperforms BPLRU in both measures. Further, PUD-LRU significantly outperforms DFTL except for the Financial 2 trace. We also found that for workloads that are highly sequential or with low temporal locality, pure page-mapping FTL, though most flexible, underperforms the log-block FTL. This is because the former spends a significant amount of time collecting blocks that contain very few invalid pages and copying valid pages to other available blocks, while the latter initiates frequent switch merges that are optimal based on SSD characteristics. We further compared the destaging efficiency between PUD-LRU and BPLRU, which shows that PUD-LRU has a higher destaging efficiency than BPLRU. In our future study of PUD-LRU, we plan to design and implement a dynamic and adaptive tuning mechanism for obtaining the optimal threshold value that facilitates the grouping of blocks in the write buffer.

Chapter 3

Design and Implementation of Garbage Collection Efficiency-Aware RAM Management

For page-mapping FTL, when the SSD contains too many invalidated pages, some blocks must be erased after valid pages in these blocks are copied to other newly allocated blocks, a process referred to as *garbage collection* (GC). The GC process affects the performance of SSD significantly not only because an erase operation takes much more time than a read (e.g., X100) or write (e.g., X10) [41], but more importantly, because GC induces significant write amplification [45], i.e., copying valid pages in the victim blocks to other available blocks before they are erased. Therefore, if the workload does not contain a sufficient number of repeated writes to generate enough garbage for the GC process to collect, each erase can only free a very limited number of invalid pages while copying a great number of valid pages to

somewhere else, thus exacerbating write amplification and decreasing GC efficiency. Further, the GC process must erase the victim blocks frequently to ensure a reasonable amount of free pages for subsequent incoming write requests. This frequent invocation of GC, unfortunately, can further lower the GC efficiency. In fact, our evaluation results based on two different commercial SSD products show that such a lowered GC efficiency can in turn cause the SSD write bandwidth to drop to as low as 12% of its peak, which indicates the significant performance impact of the low GC efficiency problem. Moreover, given the erase-count limit of today's flash memory, the low GC efficiency problem also reduces the lifetime of SSD significantly because each erase operation of the victim blocks only frees a small number of pages, which forces more frequent erase operations to serve incoming write requests.

In this chapter, we propose *GC-A*ware *R*AM *M*anagement algorithm for SSD, called GC-ARM, which not only improves the GC efficiency to alleviate write amplification, but also cooperates with the FTL to minimize the address translation overhead to reduce the write-back traffic to SSD of mapping entries. Moreover, GC-ARM can also adjust the size ratio between the write buffer and the FTL in RAM dynamically based on the workload characteristics. This work is published in [17].

3.1 Background

3.1.1 Performance impact of GC efficiency in SSD

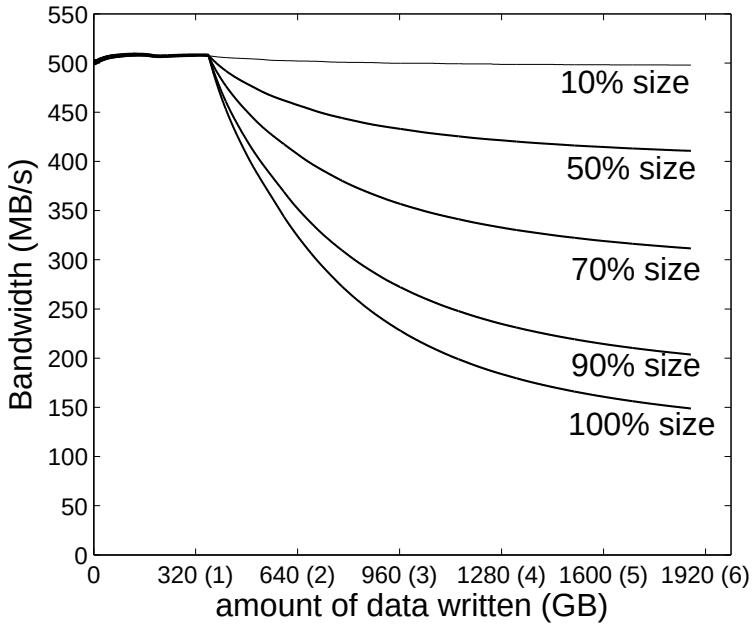
Although previous studies [22, 26, 51] show that as more and more data is written to SSD, the performance of SSD drops sharply because of garbage collection, they failed to analyze the performance impact from the perspective of *garbage collection efficiency* (GC efficiency). In fact, if GC efficiency is high, the performance does not

degrade at all, as we will show shortly.

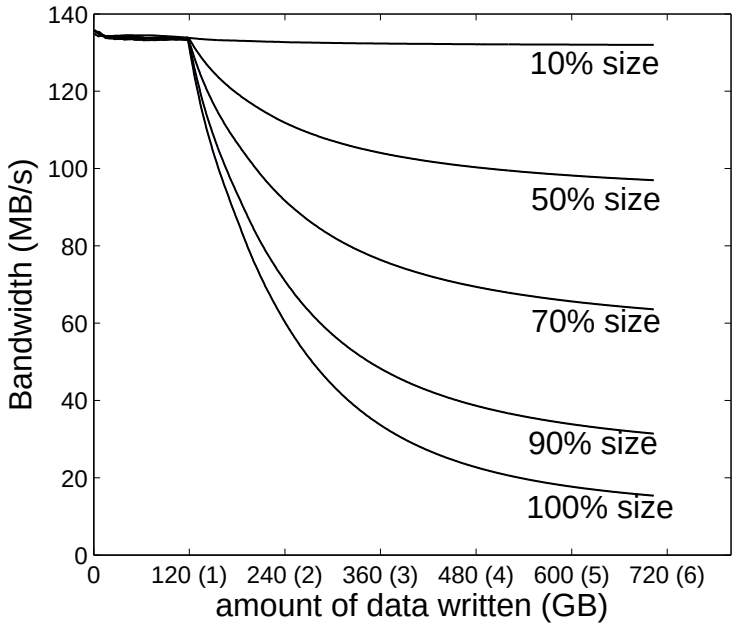
FTL determines how GC works. *Page-mapping FTL* [3] is the most flexible FTL. It maps a write request to any page address in the SSD, but it may suffer from the low GC efficiency problem, as will be demonstrated in Figure 3.1. The design of GC-ARM is based on one of the variations of the page-mapping FTL and thus one of the design goal is to improve the GC efficiency of the page-mapping FTL.

We observe that, for the page-mapping FTL, the more invalid pages there are in a collected victim block, the more useful space (in terms of number of pages) each erase operation will be able to free, and consequently the higher GC efficiency is. However, some workloads do not have enough write requests that update the same addresses (i.e., repeated updates), resulting in very few invalid pages per collected victim block and thus lowering GC efficiency, increasing the GC-induced write traffic. In order to show the wide-spread existence and significant performance impact of this low GC efficiency problem, we evaluate the write bandwidth of a 320GB Fusion IO ioDrive and a 120GB Intel 320 SSD as a function of the amount of data written to them. For each of the SSDs, we customize the workloads by issuing 64KB random write requests with their LPNs (logical page numbers) within 0-10%, 0-50%, 0-70%, 0-90% and 0-100% respectively of the maximal LPN range of the SSDs. The amount of data written is set to be 6 times the entire capacity of the SSD. By customizing the workloads in this way, we can control the amount of invalid pages generated when the SSD is full, that is, writing 0-10% of the LPN range will generate more invalid pages than the 0-100% case when the same amount of data is written.

Figure 3.1 shows the write bandwidth as a function of the amount of data written to the Fusion IO ioDrive and the Intel 320 SSD. The numbers in the parentheses of the x axis indicate the amount of data written divided by the capacity of the SSD. As we can see that, before all the free spaces are consumed by the write requests, that



(a) 320GB Fusion IO ioDrive



(b) 120GB Intel 320 SSD

Figure 3.1: Performance impact of GC efficiency in SSDs

is, the amount of data written is less than 320GB for the Fusion IO ioDrive SSD and 120GB for the Intel 320 SSD, the bandwidth is the same for the five different scenarios for both of the SSDs. This is because the SSDs still have free pages to serve the write requests without triggering the GC process. However, after all the free spaces are consumed by the write requests, the bandwidth drops at different rates because of the different GC efficiencies. On one extreme, when the LPNs of write requests are within 0-100% of the LPN range of the SSD, the bandwidth drops exponentially for both the Fusion IO ioDrive and the Intel 320 SSD. The bandwidth drops to 30% of the peak bandwidth for the former and 12% for the latter and continues its decline. On the other extreme, when the LPNs of write requests are within 0-10% of the LPN range of the SSD, the bandwidth remains flat even with an amount of the data written that is 6 times of the SSD capacity. The bandwidth of 50%, 70% and 90% cases falls in between. The reason is that when the write requests are concentrated in a small LPN range as in the 10% case, after the free pages are all consumed, those small number of LPNs are updated multiple times, resulting in a high percentage of invalidated pages in the SSDs. The GC efficiency is high in this case because the GC process can easily erase a victim block full of invalidated pages, thus making more free pages by one erase and copying very few valid pages to somewhere else to minimize the write amplification. On the contrary, when the write requests spread among 100% of the LPN range, much lower percentage of pages will be invalidated, resulting in very low GC efficiency because the GC process can not find a victim block with many invalidated pages, thus aggravating the write amplification.

3.1.2 GC-efficiency obliviousness of write-buffer management

Previous studies on the write-buffer management of SSD [16, 21, 23] concentrate on converting random writes into sequential ones to improve write performance without explicitly considering the GC efficiency problem (i.e., percentage of invalid pages per block in the SSD). As a result, they are unable to help improve the GC efficiency and reduce the GC-induced write traffic that has a significantly adversary impact on SSD performance and endurance as evidenced in the previous subsection.

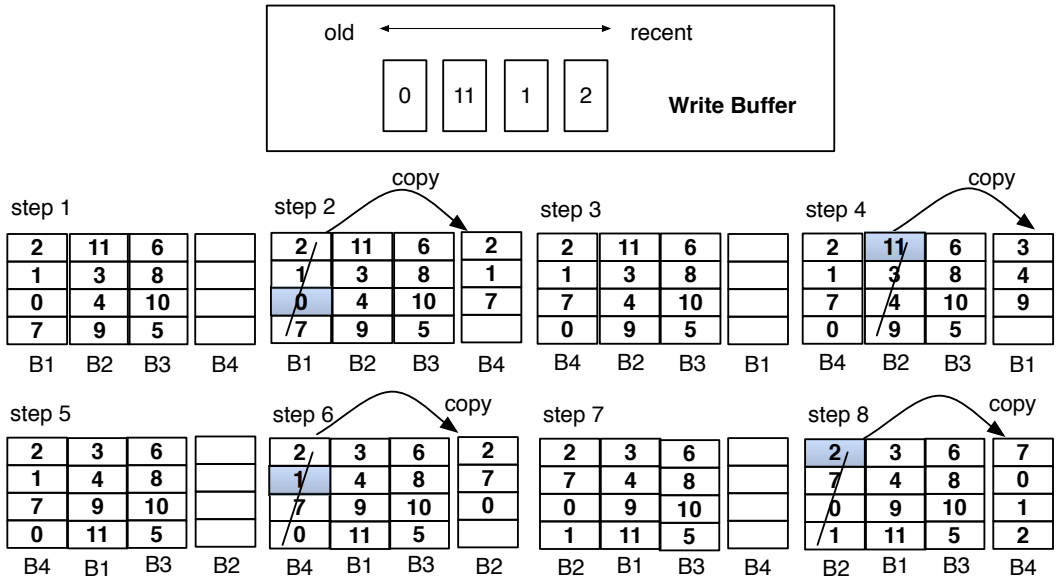
As discussed in the introduction section, BPLRU [23] is a recently proposed SSD-friendly write buffer algorithm that aims to improve the random write performance of SSDs. It converts random write requests to sequential ones by first reading the missing pages from flash memory and writing them back to SSD again to fill the "holes", a process known as *page padding* [24]. However, when the request size of the workloads is small and the LPNs of the write requests are scattered, BPLRU becomes very inefficient because the padding process increases the traffic by reading pages from SSD and writing them back again. More importantly, the increased write-back traffic increases the write amplification because these writes are artificially created by the page padding, not the original workloads.

To address this low GC efficiency problem, the destaging policy of the write buffer must compare the benefit of either destaging a page or a block by considering both the workloads and the current utilization of SSD in terms of invalid pages per block.

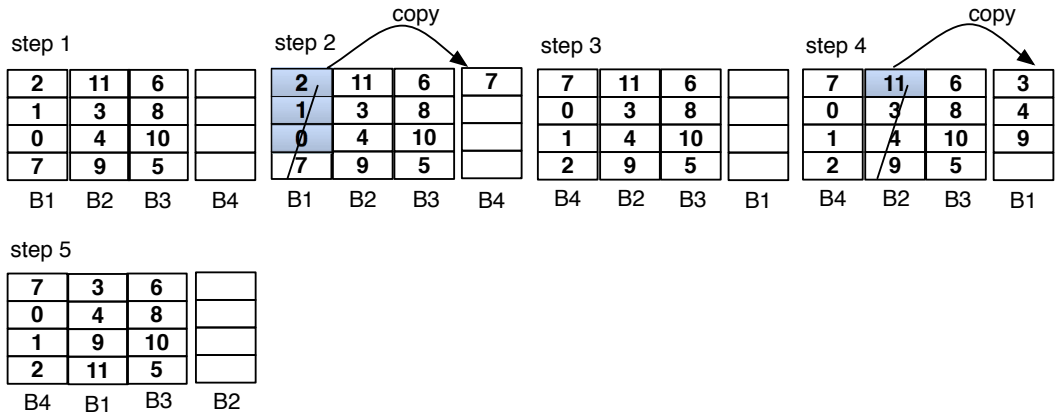
In case of the page-mapping FTL, a single destaged page from the write buffer can trigger the GC process if the flash memory does not contain free pages. Further, if an erase operation can only free very few pages, (e.g., 1 page), this destage operation will result in a very high overhead to the flash memory because storing a single page leads

to 1 erase operation. In this case, it will be more beneficial to destage a block instead of a page from the write buffer if destaging a block incurs less overhead. Figure 3.2(a) illustrates why the GC efficiency can sometimes be so low and how GC induces significant write traffic when destaging a page at a time. In this example, B1, B2 and B3 are three regular data blocks and B4 is a block in a free-block pool reserved for GC use. The pages in the write buffer are stored in an LRU order. Each block has 4 pages and the SSD is currently fully utilized, as shown in step 1 of the figure. The destaging of page 0 entails the 3 valid pages (2, 1, 7) in the same SSD block being copied to B4, along with the new page 0, before B1 is erased and becomes a free block (steps 2 & 3). A similar process is followed for the destaging of pages 11 (steps 4 & 5), 1 (steps 6 & 7), and 2 (step 8) respectively. In this example of destaging 4 pages from the write buffer, the GC process induces 4 erase operations and 12 GC-induced page writes, resulting in a very low GC efficiency. However, if the write buffer is made aware of the SSD's current utilization of blocks in the SSD, the GC efficiency can be significantly improved by destaging pages 0, 1, 2 in a single group, followed by the destaging of page 11, as shown in Figure 3.2(b). This block destaging policy leads to 2 erase operations and 4 GC-induced page writes are executed, where the GC efficiency is significantly improved compared with the previous case.

In this example, the benefit of destaging pages 0, 1, 2 together instead of individually stems from the fact that pages 0, 1 and 2 are stored in the same block in the SSD. Thus, when they are invalidated, these invalidated pages cluster in a single block, enabling GC to free more invalid pages in a single erase operation while copying fewer valid pages. Based on our observation that many workloads exhibit spatial locality, as revealed in many existing studies [16, 20], these invalidated pages tend to cluster in a small number of blocks, an important workload characteristics useful for improving the GC efficiency.



(a) GC efficiency is low when destaging a page at a time



(b) GC-efficiency improvement of destaging a block at a time

Figure 3.2: Comparison between different destaging policies showing the advantage of GC awareness

Contrary to the example shown above, there are other scenarios where destaging a single page at a time can lead to a better GC efficiency. For example, if the same LPN is written multiple times in a short period of time, the invalidated pages will be stored in the same block. In this case, destaging pages, instead of blocks, will not

affect the performance of SSD since the GC efficiency will be high. GC-ARM can dynamically estimate and compare the benefits of destaging either a block or a page based on a benefit value, which will be introduced later.

3.1.3 Partitioning RAM for the write buffer and the mapping table

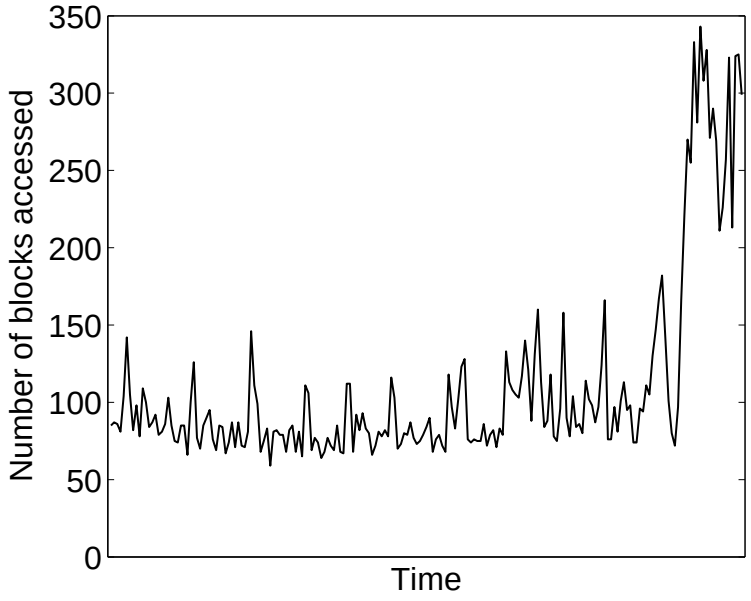
Most of the existing RAM management algorithms of SSD allocate the entire RAM space either as an FTL mapping table or a write buffer and do not consider the workload characteristics. Intuitively, for workloads with high randomness of access, allocating more RAM for the mapping table is more beneficial and thus more important than for the write buffer because the latter can not effectively absorb random write requests. Moreover, since requests of random workloads tend to scatter across much more blocks than those from less random workloads, more mapping entries are required for the cached mapping table. Our analysis of two sample workloads in Figure 3.3 shows that the randomness of workloads varies over time. We approximately associate the randomness of workloads with the number of distinctive blocks accessed in SSD for a fixed number of requests and periodically estimate this randomness value. For example, the more distinctive blocks are accessed for a fixed number of requests in a workload, the more random the workload tends to be. This figure shows the number of distinctive blocks accessed for every 500 requests. We can see from Figure 3.3(a) that the number of accessed blocks fluctuates dramatically for the Financial 1 trace. At the beginning, the workload tends to cluster around a small number of blocks, which, at a later time, has increased noticeably, implying that the randomness of Financial 1 has increased over that period of time. Figure 3.3(b) shows the number of distinctive blocks accessed by the MSN trace, which indicates that the randomness

for this trace is generally high but also fluctuating. Based on these observations, we conclude that it is important to dynamically adjust the ratio of the RAM memory space allocated between the FTL mapping table and the write buffer to make efficient and effective use of the limited RAM space.

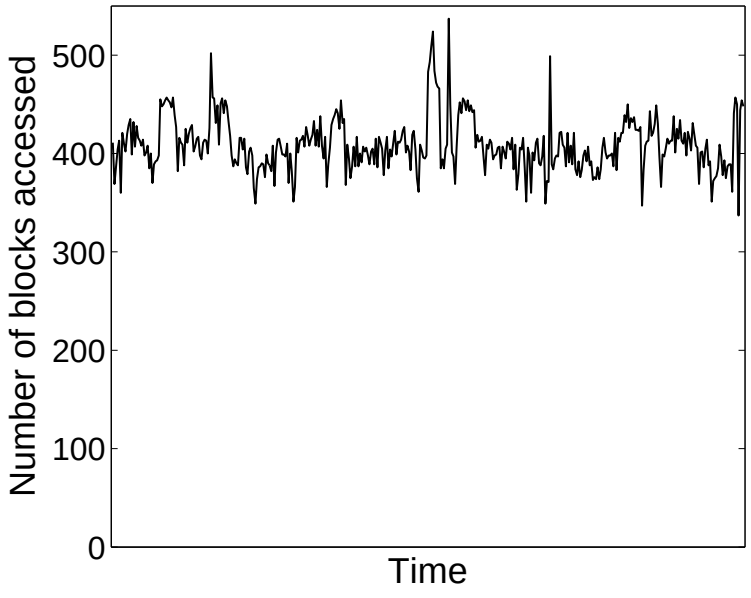
To our best knowledge, the adaptive partitioning scheme [43] for SSD's DRAM-based cache is the only approach that adjusts the size ratio between memory spaces allocated to the write buffer and the mapping table cache. In this scheme, a ghost buffer and a ghost mapping cache collect data or mapping information destaged by the data buffer and the mapping cache to predict whether it is beneficial to increase the size of the data buffer and decrease the size of the mapping cache or vice versa. However, this adaptive partitioning scheme treats the FTL as a black box when adjusting the size ratio, thus ignoring the important impact of the interactions among FTL, write buffer, and workload characteristics on the SSD performance. Moreover, the ghost write buffer and ghost mapping table incur non-negligible and potentially significant overhead in maintaining the metadata destaged from the real write buffer and real mapping table.

3.1.4 Reducing write-back traffic due to mapping entry replacement

For existing approaches that only store part of the LPN-to-PPN mapping entries in RAM and use an on-demand method in an LRU order to write back the least recently used LPN-to-PPN mapping entries in RAM to the SSD, the write-back traffic of the replaced mapping entries can be overwhelming. DFTL [13] is the most recent and represents the state of the art of such approaches. It exploits the temporal locality of workloads by only storing a subset of the mapping entries in RAM while leaving the



(a) Financial 1



(b) MSN

Figure 3.3: Number of distinctive blocks accessed per 500 requests

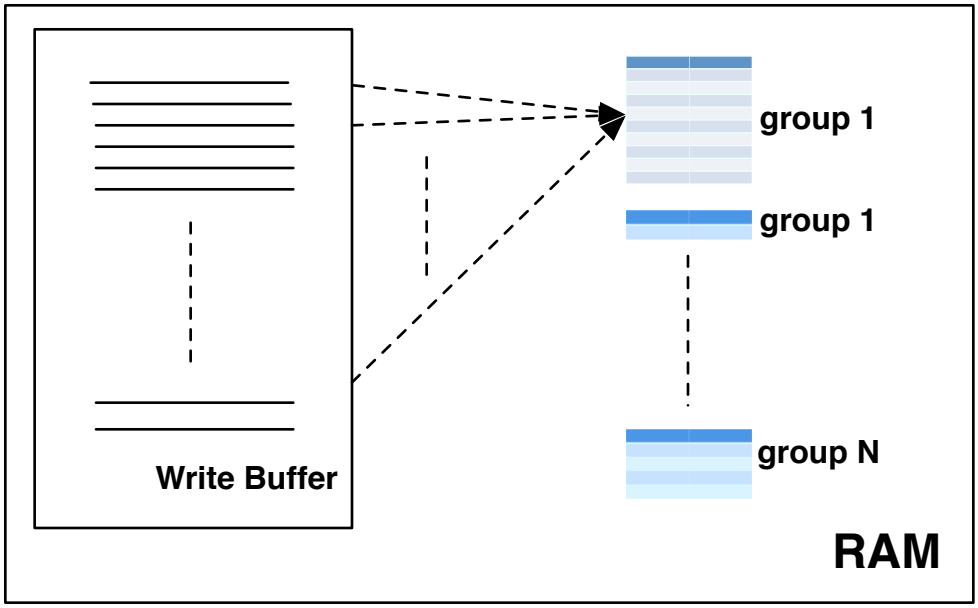


Figure 3.4: Interplay between pages in the write buffer and the mapping entries

rest in the flash memory (i.e., *translation pages*) sequentially stored in the SSD based on their LPNs. Each replacement of the mapping entries in the RAM consumes a translation page. When all the translation pages are consumed, the GC process is invoked to recycle invalid translation pages, which results in a significant translation overhead. Therefore, in order to maximize the number of invalid pages in these erase blocks, LPN-to-PPN mapping entries should be grouped based on their LPN proximity and contiguity while pages should be carefully selected for destaging from the write buffer instead of being destaged simply on an LRU basis. To achieve this, the write buffer can interact with the different mapping groups so that pages belonging to the group that contains the most LPN-to-PPN mapping entries are destaged first, as shown in Figure 3.4.

These observations, combined with our study of the existing flash-aware buffer management algorithms and FTLs, motivate us to propose GC-ARM. GC-ARM im-

proves the GC efficiency based on the dynamic block usage of SSD and the utilization of the write buffer. Based on the spatial locality of workloads, GC-ARM groups logically continuous LPN-to-PPN mapping entries together and maximizes the groups by exploiting the interplay between the write buffer and the mapping table, thus reducing the LPN-to-PPN translation overhead. Further, GC-ARM dynamically adjusts the ratio of the RAM space allocated between the write buffer and the mapping table based on the randomness of the workloads at runtime to optimize performance.

3.2 Design and implementation of GC-ARM

As shown in Figure 3.5, a GC-ARM based SSD consists of two parts, namely, the *GC-ARM* part and the *Flash Memory* part. The former is composed of three functional modules, a *write buffer*, a *cached page-mapping table* and a *monitor*. When the write buffer is full, the monitor helps improve the GC efficiency and reduce the GC-induced write traffic by deciding whether to destage a page or a block from the write buffer based on the *benefit value*, which will be introduced shortly. Further, the monitor reduces the address translation write-back traffic by deciding which page to destage based on the current usage of the write buffer and the mapping entry groups. Moreover, it samples the workloads to dynamically adjust the size ratio of RAM for the write buffer and the cached mapping table. Figure 3.5 shows how a write request is processed in a sequence of steps. The write buffer first determines whether the incoming request from the block interface hits in the write buffer in step (1). If it misses in the write buffer and the write buffer is full (step (2)), the write buffer asks the monitor to determine whether a page or a block should be destaged based on the *benefit value* calculated by the monitor (step (3)). If the monitor decides to destage a block, the cached mapping table is checked to find the mapping entries for pages in

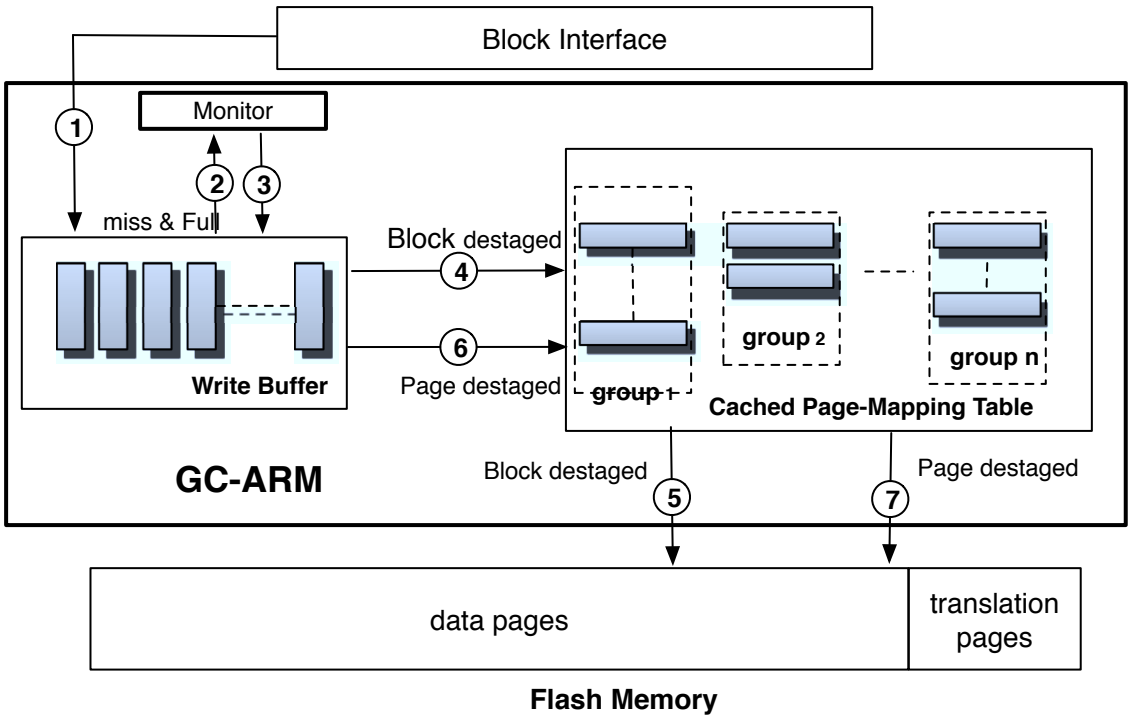


Figure 3.5: The GC-ARM architecture

the block to be destaged (step (4)) before the block is written to the flash memory (step (5)). On the other hand, if the monitor decides to destage a page, the cached mapping table is checked (step (6)) to determine which page should be destaged to minimize the translation overhead by maximizing the LPN-to-PPN group, which will be explained later (step (7)). For a read request, it first checks the write buffer that returns the data if there is a hit. Otherwise, the read request consults the cached mapping table to read the data from SSD.

3.2.1 The design of FTL component

The flash memory reserves a small number of blocks to store the complete set of mapping entries. A 32GB SSD with a page size of 4KB only requires 64MB SSD

space to store all the mapping entries if each mapping entry occupies 8 bytes, which is only 0.2% of the SSD capacity. Without loss of generality, we assume 4KB-page and 8-Byte mapping entry in the discussion below. The page-mapping FTL only keeps a small subset of all mapping entries in RAM to reduce the RAM consumption based on the assumption that the working sets of workloads vary over time and from workload to workload. As shown in Figure 3.5, unlike DFTL that organizes RAM entries in an LRU queue, the GC-ARM FTL component groups the mapping entries in RAM according to their LPNs so that mapping entries belonging to the same translation page are put in one group. For example, LPNs 0 through 511 are in one translation page and thus all RAM entries with LPNs in this range are put in one group. When a read or write request arrives and it misses in the cached mapping table, it tries to find victim mapping entry by first scanning all the cached mapping entries to see whether there are clean entries or not. If there are, it simply removes a clean entry from the cached mapping entries as the victim for replacement, reads the requested PBN (physical block number) from the reserved SSD translation blocks and stores it in the corresponding group. Otherwise, it employs a greedy scheme that destages *an entire group* that has *the most number of mapping entries* as the replacement victim, which can significantly increase the efficiency of updating a page in the reserved SSD translation block. The reason for choosing a group with the most number of mapping entries to destage lies in the fact that no matter how much data is updated in a page, an entire translation page is consumed. This greedy scheme corresponds to step (7) in Figure 3.5 and it will further reduce the translation overhead when it interplays with the write buffer, as we will show shortly.

3.2.2 GC-aware destaging

We have shown in Figure 3.1 that workloads with fewer write requests updating the same LPNs will generate fewer invalid pages in a block. The much-lowered GC efficiency in turn degrades the performance significantly. GC-ARM is designed in part to prevent the performance from being degraded under such workloads.

GC-ARM achieves this goal by dynamically destaging either a single page or a sequential block based on the knowledge of current status of the write buffer and the amount of invalid pages per block in the SSD. Specifically, GC-ARM evaluates the benefits of either destaging a single page or a sequential block and then destages based on whichever benefit is bigger.

We define the benefit value differently for the two destaging policies. But essentially they are based on the same criteria, that is, how many free pages an erase operation can provide. Therefore, we can compare the two different benefit values to make decisions. The benefit value for the page destaging policy is defined as the maximal number of pages an erase operation can free because GC-ARM employs an on-demand greedy policy to select the victim block in the SSD to erase, as shown in Equation 3.1. In this equation, $NVB_{invalid}$ is the number of invalid pages in a victim block to be erased.

$$B_{page-destaging} = NVB_{invalid} \tag{3.1}$$

On the other hand, GC-ARM improves GC efficiency by destaging a sequential block because many workloads exhibit spatial locality [16, 20]. Therefore, when a block is destaged, the invalidated pages in SSD resulting from this destaged block will likely cluster in a small number of blocks, thereby enabling the GC process to free more pages in an erase operation and thus improving the GC efficiency. Each such

destaging of a sequential block consumes a block in the SSD and will be erased later. Thus, each erase can accommodate NDB_{valid} pages, where NDB_{valid} indicates the number of valid pages in the destaged block in the write buffer. Further, because GC-ARM employs the page-padding scheme, the number of invalid pages in the destaged block in the write buffer, $NDB_{invalid}$, should be subtracted from NDB_{valid} because it is the overhead of the padding scheme. Therefore, the benefit value of the block destaging policy is defined in Equation 3.2.

$$B_{block-destaging} = NDB_{valid} - NDB_{invalid} \tag{3.2}$$

When destaging is needed, the monitor calculates the benefit values for both the page-destaging scheme and the block-destaging scheme, based on Equations 3.1 and 3.2 respectively, and chooses the scheme with the larger benefit value to optimize GC efficiency.

Therefore, if the workload has many write requests that update the same addresses, more invalid pages in SSD will be generated. In this case, based on the equations above, the page-destaging scheme is chosen more frequently because the benefit value of page-destaging tends to be larger, where an erase operation can free more pages and cause fewer valid pages to be copied. On the other hand, when the workload no longer has many write requests updating the same addresses, meaning that there are not many invalid pages in SSD, continuing page destaging to a full SSD will force the GC process to be triggered frequently, with each GC process freeing only a small number of pages but copying a large number of valid pages. This results in very low GC efficiency and high GC-induced write traffic. GC-ARM can easily detect this situation by the discovery of a larger benefit value of the block-destaging scheme than that of the page-destaging scheme and switch to the block-destaging scheme to

increase the GC efficiency and improve the performance.

3.2.3 Adaptive adjustment of RAM space partitioning

Another key functionality of GC-ARM is to dynamically adjust the RAM size ratio between the write buffer and the mapping table, adapting to the changing randomness of workloads. For random workloads, the write buffer can become ineffective and of little benefit while it will be beneficial to cache more mapping entries in RAM because random workloads will likely lead to a high miss rate in the cached mapping table. The monitor calculates the randomness of workloads based on the number of distinctive blocks accessed by workloads during a sampling window with a fixed number of requests. We set the sampling window to be 500 requests. In determining the randomness of a workload in the current implementation of GC-ARM, if over half of the 500 requests access distinctive blocks, GC-ARM considers it random because on average every two consecutive requests access two different blocks. Thus, we choose 250 distinctive blocks accessed in a 500-request window as the threshold at or beyond which the write buffer size will be decreased and the mapping table size will be increased. Moreover, for the design of GC-ARM, since it is possible for the randomness of workloads to change dramatically over time, we choose only to decrease the write buffer by a 16KB decrement and increase the mapping table by a 2048-entry increment at a time because 1KB memory can accommodate 128 mapping entries. By doing so, we can avoid the situation in which, as soon as the write buffer size is substantially decreased, the workload begins to exhibit strong temporal locality, inducing unnecessary destaging due to the significant reduction in the write buffer size. On the other hand, if the number of distinctive blocks accessed is less than 250 and half of the requests are write requests, meaning that the write requests have high

temporal locality and the write buffer can become more beneficial, then the monitor will increase the write buffer by 16KB and decrease the mapping table by 2048 entries. By adjusting the size of memory allocated to the write buffer and the mapping table, GC-ARM adapts dynamically to the characteristics of workloads.

3.2.4 Interplay between write buffer and cached mapping table

The write buffer can further reduce the address translation overhead by interacting with the FTL component. The write buffer can select a page that minimizes the translation overhead to destage whenever the page-destaging scheme is selected. Based on the status of the FTL component, when there are clean mapping entries in the cached mapping table, the least recently written page is chosen to be destaged. Otherwise, the page that falls into the *largest* mapping-entry group is chosen to be destaged so that this maximal mapping-entry group can further grow in size. This interplay between the write buffer and the cached mapping table helps to reduce the write-back traffic of translation entries since the largest group will be written back when necessary.

3.2.5 Data integrity concern

Similar to what we have discussed in Section 2.2.4, data in the write buffer and mapping entries in RAM may be lost if there is a sudden power failure, giving rise the data integrity concern. Several existing technologies can be leveraged to ensure data integrity. For example, SandForce SF1500 [1], which is an enterprise-level SSD product, employs a super-capacitor as an insurance policy to guarantee data integrity. Similar data-integrity protection technologies, such as battery-backed RAM, continuous data

protection (CDP), etc., may also be employed.

3.3 Performance evaluation

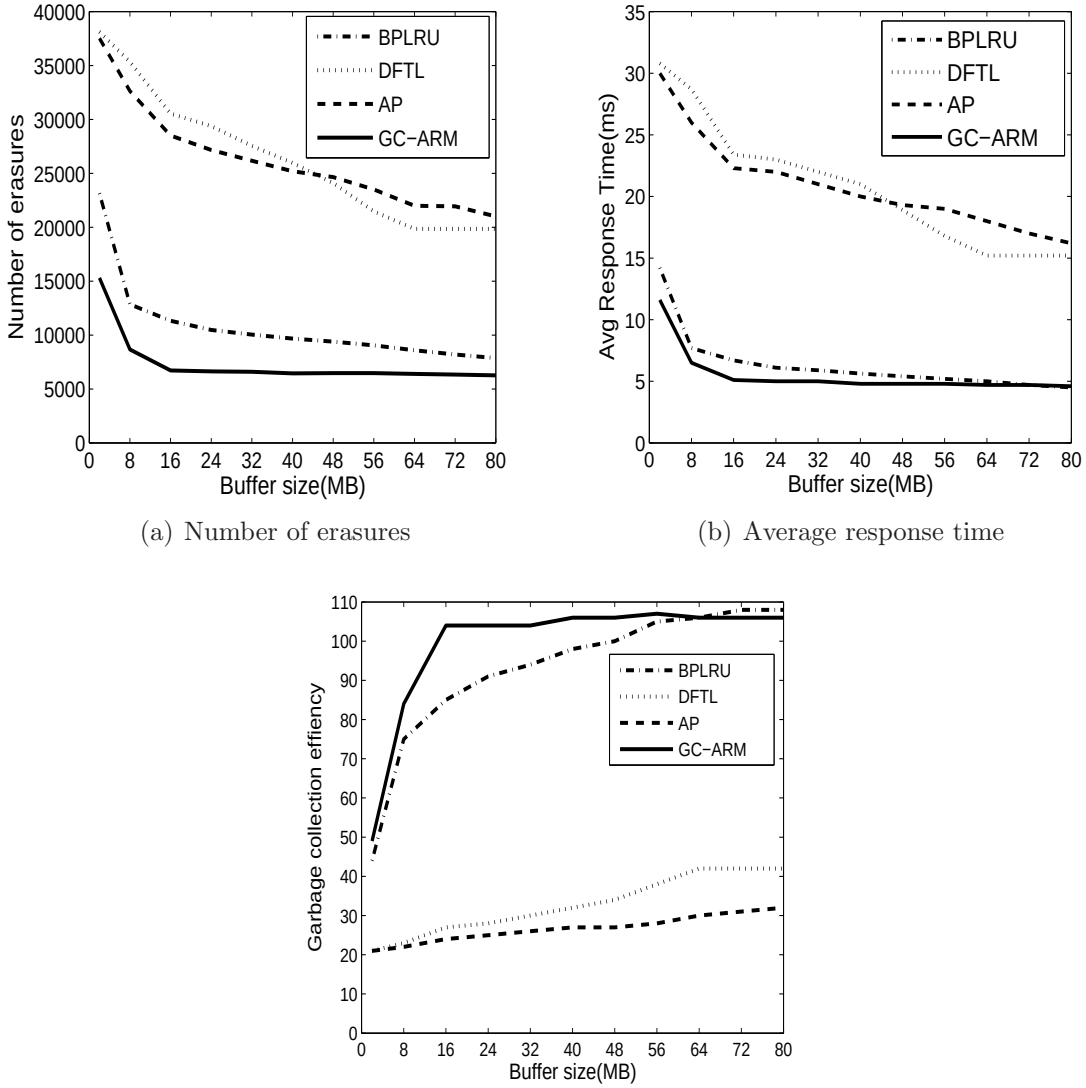
3.3.1 Experimental setup

In our trace-driven simulation study of GC-ARM, we simulate a 32GB SSD, with a page size of 4KB and a block size of 512KB, based on FlashSim [53]. We implemented GC-ARM and a recently proposed adaptive RAM-space partitioning scheme [43], which we call *AP* for brevity, that treats FTL as a black box. we generates number of erasures, average response time, the average number of pages freed by an erase operation and the amount of write traffic reduction after the simulation. These measures are used as the endurance, GC efficiency, and user performance metrics to evaluate the effectiveness of GC-ARM against the state-of-the-art schemes BPLRU, DFTL and AP.

We fed five workloads to the simulator to evaluate and compare the aforementioned metrics among GC-ARM, BPLRU, DFTL, and AP. The five traces are Financial 1 [39], Financial 2 [39], Microsoft Exchange Server [36], Microsoft MSN File Server [38], and TPC-E benchmark collected at Microsoft [37], whose key I/O characteristics are summarized in Table 2.2. The SSD in our experiment is configured with its key parameters listed in Table 2.4.

3.3.2 Performance and GC efficiency

Figure 3.6 through Figure 3.10 plot the number of erasures, average response time and garbage collection efficiency respectively as a function of the RAM size. BPLRU employs the RAM only as a write buffer while DFTL employs the RAM only as a



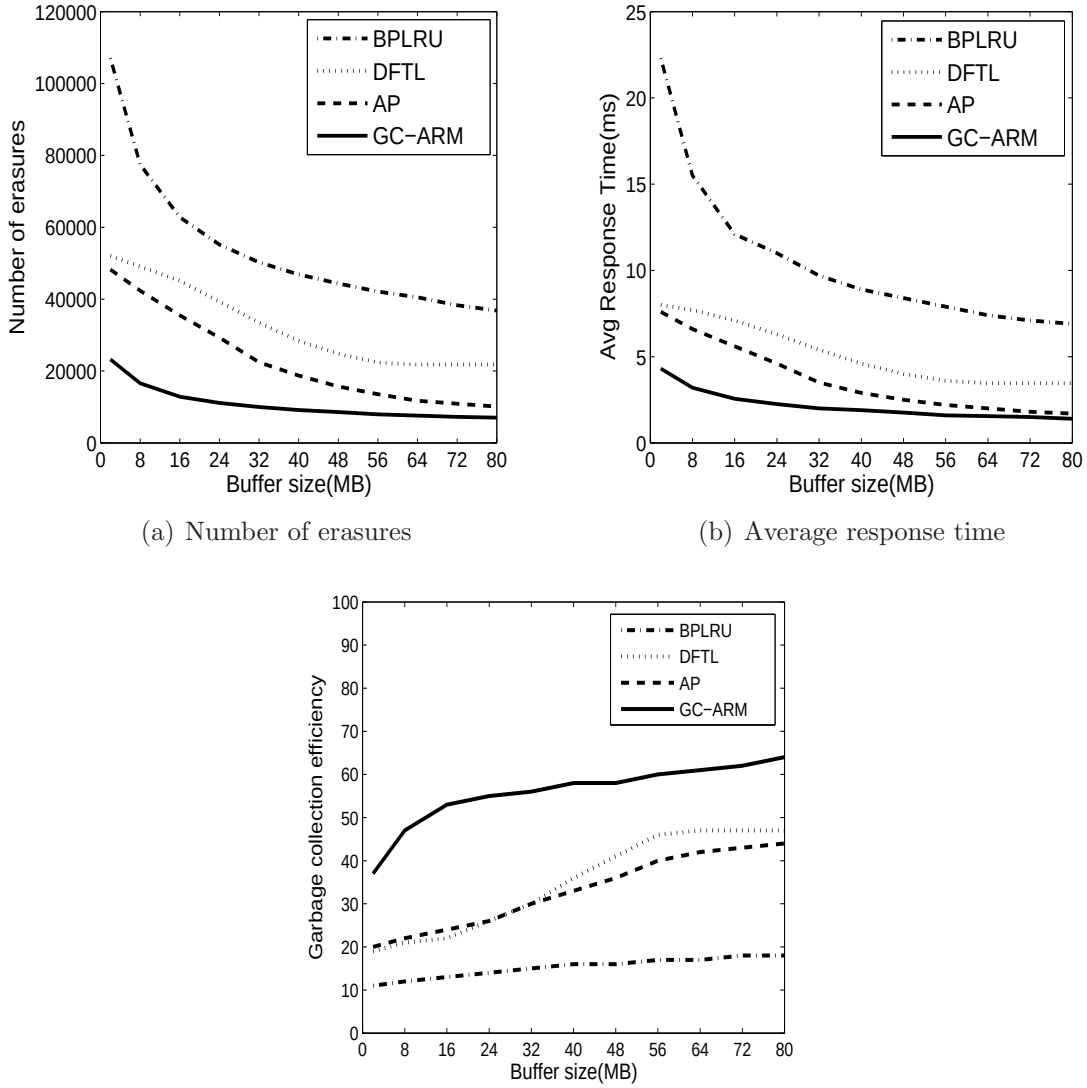
(a) Number of erasures

(b) Average response time

(c) Garbage collection efficiency

Figure 3.6: GC efficiency evaluation of Financial 1 workload

cached mapping table. Both GC-ARM and the AP scheme dynamically adjust the size ratio between the write buffer and the mapping table. It is worth noting that, because the simulated SSD requires 64MB RAM to store the mapping table in its entirety, when the RAM size is equal to or more than 64MB, the performance of DFTL will be identical to that of the pure page-mapping FTL scheme in all these



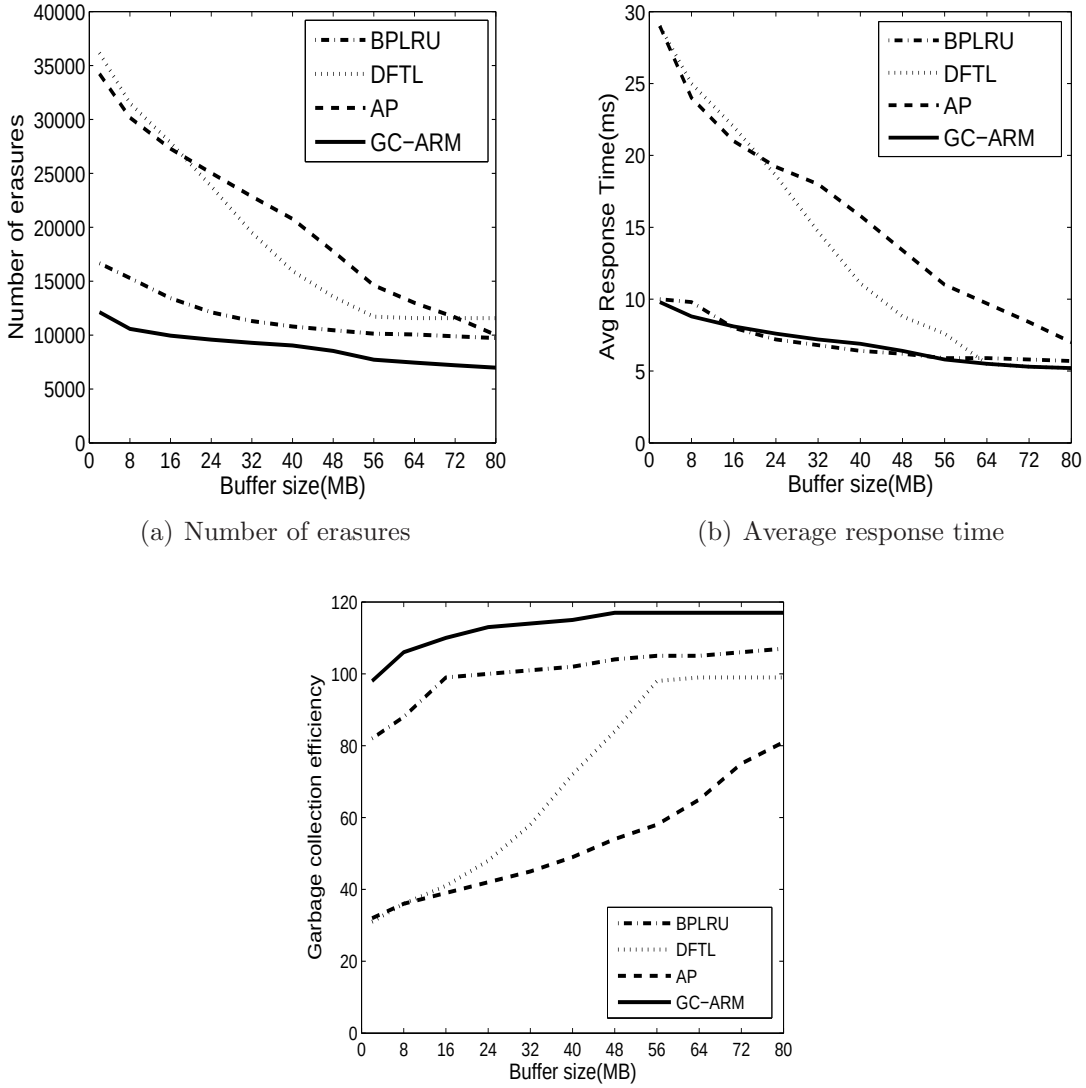
(a) Number of erasures

(b) Average response time

(c) Garbage collection efficiency

Figure 3.7: GC efficiency evaluation of Financial 2 workload

figures because the former is able to store the entire mapping table in RAM, reducing itself to the latter. For the GC efficiency evaluation, DFTL, GC-ARM and AP employ page-mapping FTLs, so we use the average number of invalid pages in a victim block collected by GC as a metric because this is the average effective number of pages freed by an erase operation to serve future requests. On the other hand, BPLRU employs



(a) Number of erasures

(b) Average response time

(c) Garbage collection efficiency

Figure 3.8: GC efficiency evaluation of Exchange workload

page-padding, which first reads some valid pages from the SSD to convert random write requests into a sequential block in the write buffer and then destages it in to the SSD. The log-block FTL switches the fully sequentially written log block to the data area and switches back the old data block to the log area and erase it, which is called "switch merge". The erase operation that erases the old data block enables the valid

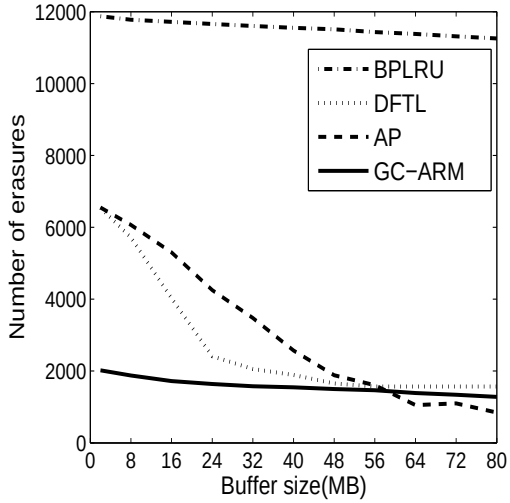
pages in the log block to be stored, which is equivalent to the page-mapping based schemes that free invalid pages in a block by the erase operation. Hence, we use the average number of valid pages destaged in each destage operation, which is also the average number of valid pages in the log block when a switch-merge operation occurs, to evaluate the GC efficiency of BPLRU.

For the Financial 1 trace, GC-ARM is slightly better than BPLRU while both GC-ARM and BPLRU significantly outperform the two page-mapping based FTL schemes, DFTL and AP, for both performance measures, as shown in Figure 3.6(a) and Figure 3.6(b). This significant performance difference lies in the fact that the log-block-FTL-based BPLRU employs page-padding to destage a block each time and convert all full merges to the more efficient switch merges while GC-ARM dynamically decides whether to destage a block or a page at a time from the write buffer based on the workload characteristics. This observation is also confirmed by the GC efficiency results, shown in Figure 3.6(c), where the GC efficiencies of BPLRU and GC-ARM are seen to be much higher than those of DFTL and AP. The average number of valid pages destaged by BPLRU is very high, which indicates that the Financial 1 trace has very strong spatial locality. GC-ARM can dynamically adapt to this workload by destaging blocks instead of pages to improve GC efficiency. Based on these results, we conclude that workloads similar to the Financial 1 trace, where the request size is large and have stronger spatial locality, are well suitable for block-based schemes such as BPLRU or adaptive destaging schemes such as GC-ARM.

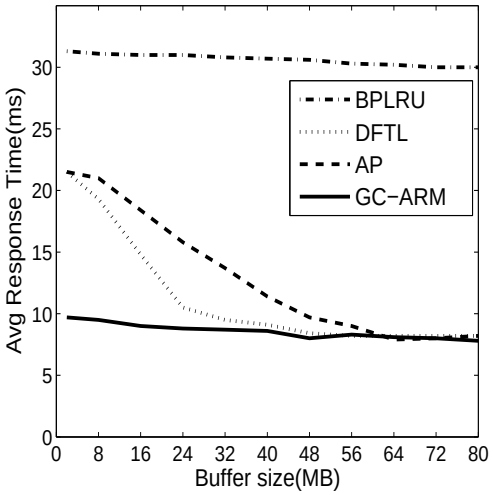
On the other hand, the Financial 2 trace shows a very different performance under different schemes than the Financial 1 trace, as evidenced in Figure 3.7(a) and Figure 3.7(b). GC-ARM, AP and DFTL outperform BPLRU significantly while GC-ARM consistently outperforms DFTL and AP for both the number-of-erasure and the average-response-time measures. The main reason for BPLRU's inferiority

to all the other three schemes is because the Financial 2 trace has much smaller request sizes than the Financial 1 trace, forcing BPLRU to spend most of its time reading pages from the SSD to pad a relatively large number of invalid pages to form a whole block to destage. In this case, since each time only a relatively small number of valid pages are destaged, the GC efficiency is significantly reduced for BPLRU. This is confirmed by Figure 3.7(c), which shows that all the three page-mapping based schemes outperform BPLRU, with GC-ARM being the best among these three page-mapping based schemes. While the other two page-mapping FTL based schemes spend a lot of time on address translation, GC-ARM, with the help of the write buffer, can detect the status of the mapping entries, thus minimizing the translation overhead. The reason for GC-ARM's performance advantage over the other two page-mapping based schemes, DFTL and AP, especially when the RAM size is small, is because the write buffer of GC-ARM not only absorbs the requests but also judiciously interacts with the FTL component to minimize the translation overhead. The cached mapping table in GC-ARM removes the mapping-entry group with the most mapping entries based on a greedy policy. When the RAM size is small, the benefits of the interplay between the write buffer and the FTL component and the greedy destaging of the mapping entries become more pronounced.

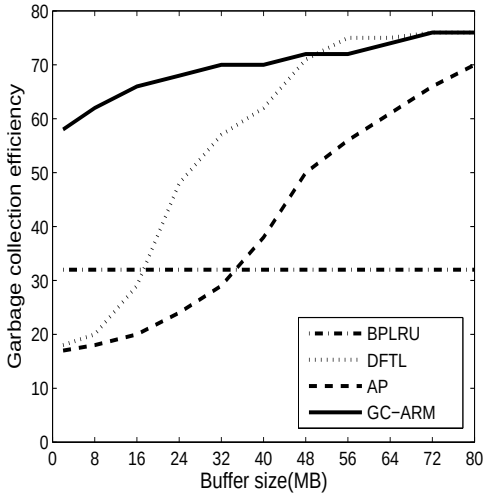
Although the Exchange trace has a strong repeatedly updating pattern, as shown in Table 2.2, the average request size of this trace is relatively large. As a result, both GC-ARM and BPLRU outperform DFTL and AP in both measures and GC-ARM is slightly better than BPLRU for the number-of-erasure measure and almost identical to BPLRU for the average-response-time measure, as shown in Figure 3.8(a) and Figure 3.8(b). For the same reason, the GC efficiency of both GC-ARM and BPLRU is higher than the other two page-mapping based schemes, as shown in Figure 3.8(c). Since GC-ARM can dynamically adjust itself to destage a block at a time in most



(a) Number of erasures



(b) Average response time

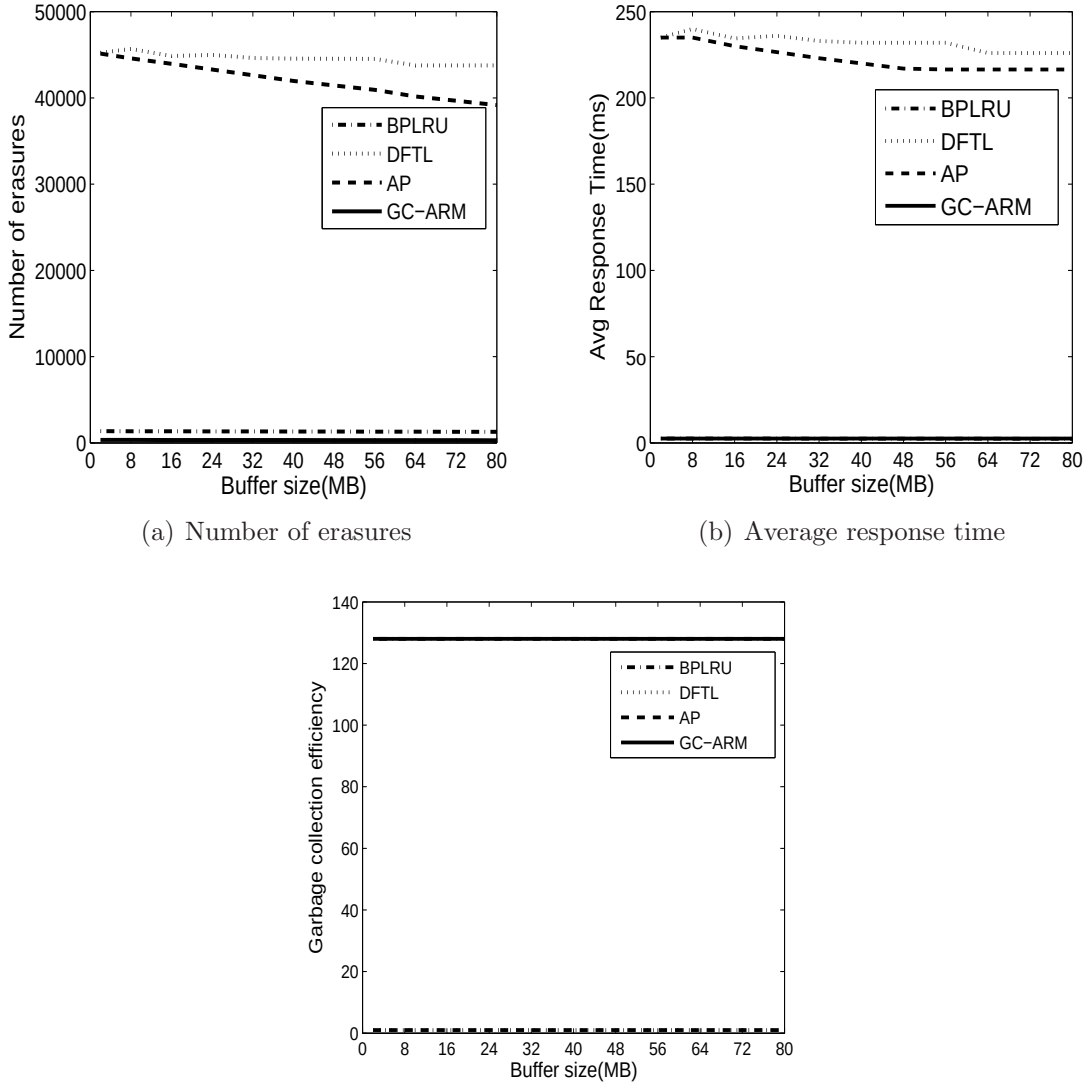


(c) Garbage collection efficiency

Figure 3.9: GC efficiency evaluation of MSN workload

cases, it achieves a comparable performance to BPLRU while outperforming the other two page-mapping-based schemes, DFTL and AP.

While the MSN trace has a relatively large average request size, its requests are sparsely scattered in the address space. This is why BPLRU is much worse than the other three schemes in both measures, as seen in Figure 3.9(a) and Figure 3.9(b).



(a) Number of erasures

(b) Average response time

(c) Garbage collection efficiency

Figure 3.10: GC efficiency evaluation of TPC-E workload

This is further evidenced by the GC efficiency results in Figure 3.9(c). Based on the random nature of the MSN trace, the GC efficiency of BPLRU remains unchanged. When the RAM size is small, both DFTL and AP suffer from a heavy address translation overhead, even underperforming BPLRU. As the RAM size increases, their GC efficiency begins to increase and eventually all the three page-mapping based

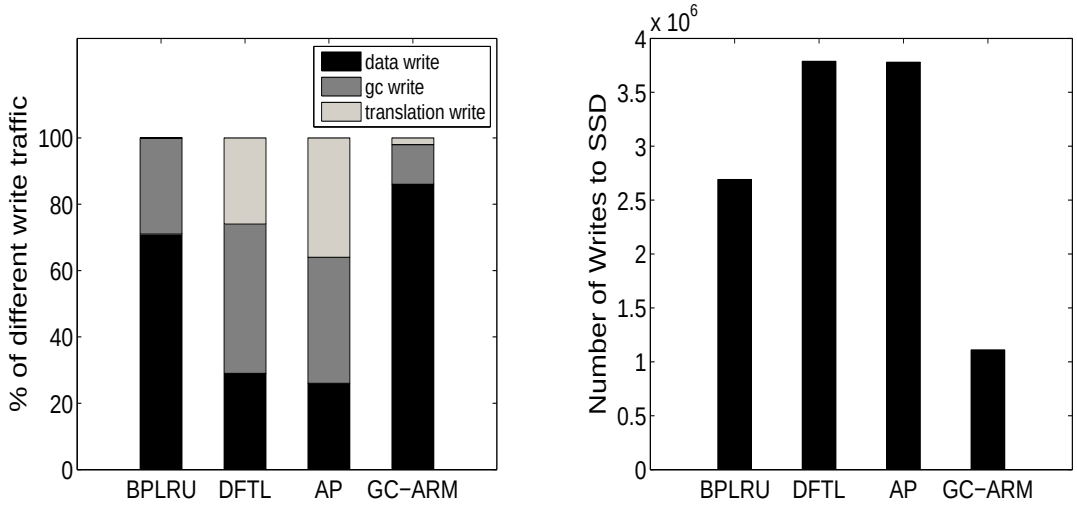
schemes outperform BPLRU. This is because page-mapping FTL can deal with random workloads much more effectively than block-mapping FTL as the former can map a request to anywhere inside SSD. GC-ARM can adaptively switch itself between the page-destaging and block-destaging schemes, thus outperforming DFTL and AP when the RAM size is less than 48MB while performing almost identically when the RAM is more than 64MB when the entire mapping table can be stored in RAM.

The TPC-E trace is very unique and pathologic in that almost no two or more write requests share the same logical address and the write requests update a block randomly until the block is almost full, then the requests go to the next block and continue updating the block until it is full, and so on. Hence, BPLRU performs much better than the other two page-mapping based schemes, DFTL and AP, because each destaged block from the write buffer contains almost entirely valid pages, maximizing the GC efficiency. For this trace, GC-ARM adapts to the workload and dynamically switches itself to the block-destaging scheme, destaging a block at a time as in BPLRU. So GC-ARM performs much better than DFTL and AP, as shown in Figure 3.10(a) and Figure 3.10(b). Furthermore, for BPLRU, only log blocks can accept write requests, while GC-ARM, employing a page-mapping FTL, can map a destaged block to any block inside SSD, thus it outperforms BPLRU. Another very important reason why both DFTL and AP perform so poorly is that in the TPC-E trace almost no two or more write requests have the same logical block address, thus generating very few invalid pages. Therefore, the GC efficiency of the pure page-mapping-based schemes is very low and the GC-induced write traffic is very high, as indicated in Figure 3.10(c), where both BPLRU and GC-ARM achieve 128 (i.e., maximal GC efficiency) while DFTL and AP only achieve a very low GC efficiency.

As mentioned before, when the RAM size is more than 64MB, the DFTL scheme's

performance is equal to that of the pure page-mapping FTL. It is worth noting that GC-ARM outperforms DFTL for all workloads when the RAM size is more than 64MB, except for the MSN trace under which GC-ARM and DFTL perform equally, which indicates that GC-ARM is superior to the most flexible page-mapping FTL. *Therefore, simply keeping all mapping entries in RAM, as the pure page-mapping FTL does, does not guarantee the best performance.* This implies that, even if RAM space and cost are not a major concern, blindly throwing in RAM capacity to SSD will not always buy one the desired performance!

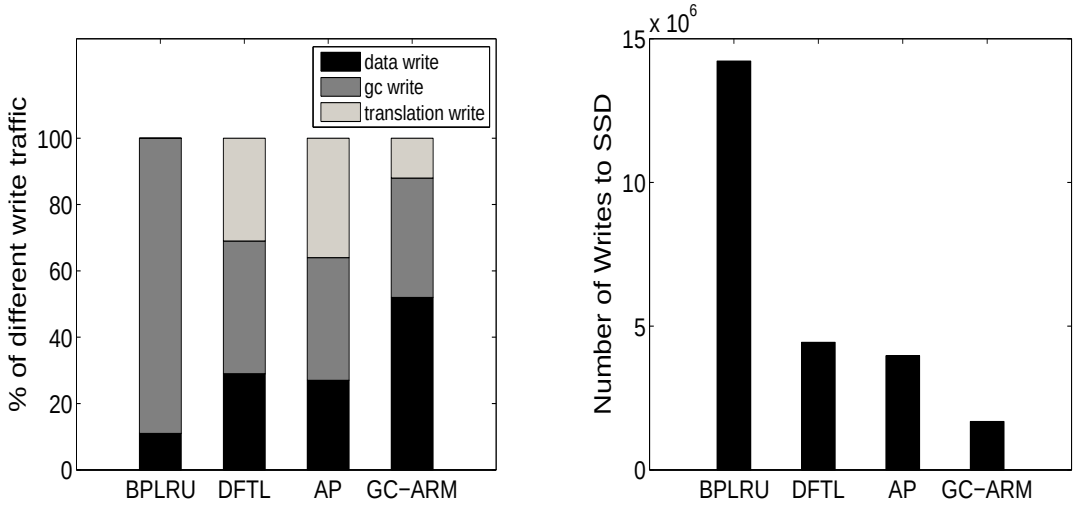
3.3.3 Write traffic reduction



(a) Financial 1 (b) Financial 1

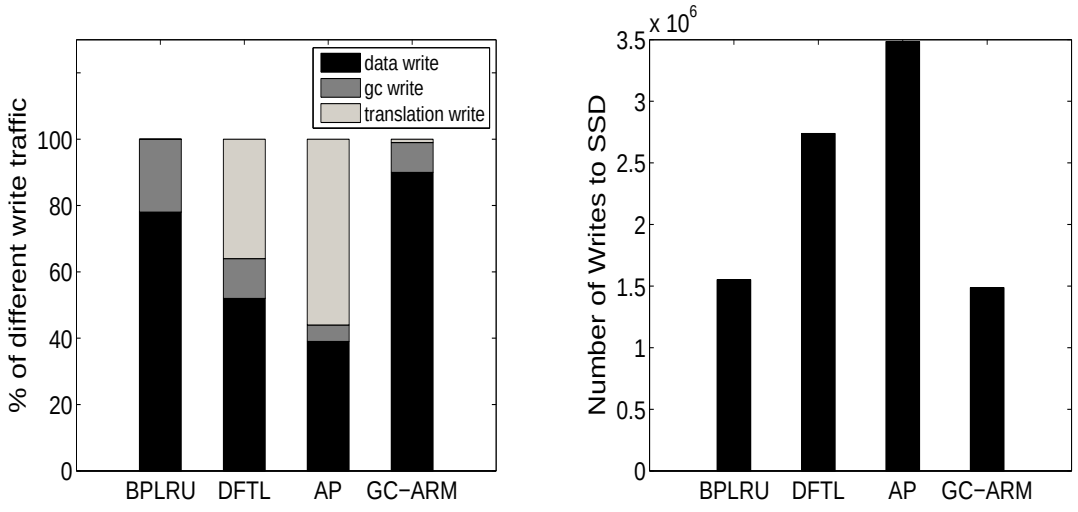
Figure 3.11: Write traffic distribution and the total number of writes of Financial 1

GC-ARM aims to reduce both the GC-induced write traffic by improving the GC efficiency and the address translation write-back traffic, which in turn increases the percentage of data write traffic. Figure 3.11 to Figure 3.15 show both the distribution of different kinds of write traffic and the total number of writes to SSD for the four



(a) Financial 2 (b) Financial 2

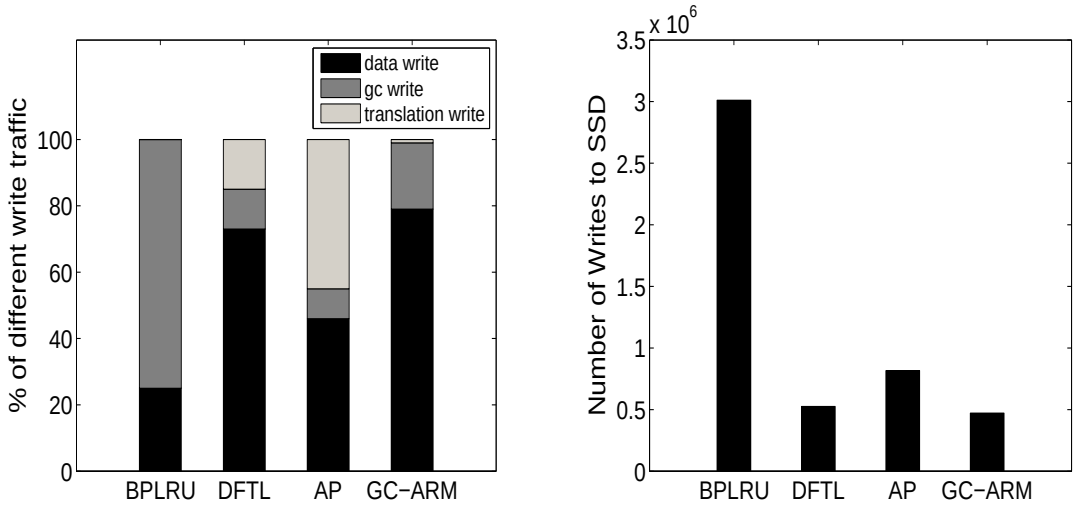
Figure 3.12: Write traffic distribution and the total number of writes of Financial 2



(a) Exchange (b) Exchange

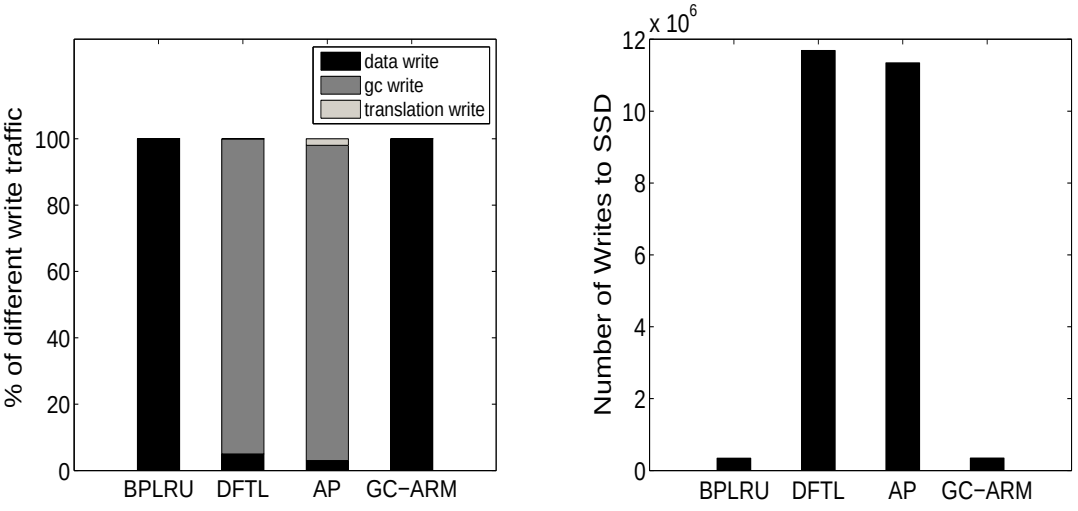
Figure 3.13: Write traffic distribution and the total number of writes of Exchange

approaches under the five workloads. Among these figures, the left subfigures show the percentages of the data write traffic, GC-induced write traffic and address translation write-back traffic, which are labeled as "data write", "gc write" and "translation write", respectively. The right subfigures show the total number of writes to SSD.



(a) MSN (b) MSN

Figure 3.14: Write traffic distribution and the total number of writes of MSN



(a) TPC-E (b) TPC-E

Figure 3.15: Write traffic distribution and the total number of writes of TPC-E

BPLRU is built on top of the log-block FTL, so there is no translation write associated with it because all mapping entries can be stored in RAM. For Financial 1 trace, the data write traffic of GC-ARM is 90% of all write traffic, much higher than the other three approaches evaluated. GC-ARM reduces much of the translation write compared with DFTL and AP because of the interplay between the write buffer and the mapping table. Moreover, GC-ARM also reduces the GC-induced write traffic because of its ability to improve the GC efficiency and cluster invalid pages before erasing, which is also confirmed by Figure 3.6(c). On the other hand, the Financial 2 trace shows a very different performance. We can see that the GC-induced write traffic in GC-ARM is significantly reduced compared with that in BPLRU. This is because the request size of Financial 2 is small, which causes BPLRU to read a lot of pages to pad to a full block that is written back later and increases the number of writes. GC-ARM also outperforms DFTL and AP under this workload because it reduces the translation write-back traffic. Due to the large request size of the Exchange trace, both GC-ARM and BPLRU have a higher data write percentage. GC-ARM also outperforms DFTL and AP because GC-ARM can maximize the mapping groups based on the strong spatial locality of this workload. The MSN trace is relatively random because fewer addresses are updated more than once, as shown in Table 2.2. Thus, the percentages of translation write-back traffic for both DFTL and AP are relatively high. However, with the help of the interplay between write buffer and the mapping table, this percentage is greatly reduced in GC-ARM. For the pathologic TPC-E trace (as explained in Section 3.3.2), BPLRU can achieve almost 100% data write because no page-padding is needed. GC-ARM performs equally well because it chooses block-destaging all the time based on the *benefit value*. On the other hand, the percentages of GC-induced write traffic of DFTL and AP are extremely high, almost 99%, because this workload rarely has two or more write requests updating

the same addresses, and thus causing DFTL and AP to generate very few invalid pages and copy a large number of valid pages. This is consistent with results shown in Figure 3.10(c).

GC-ARM also consistently reduces the total number of writes to SSD, as shown in the right subfigures in from Figure 3.11 to Figure 3.15, because of its ability to effectively reduce the GC-induced and address-translation-induced traffic as explained before.

3.4 Summary

In this chapter, we proposed GC-ARM, a garbage-collection aware RAM management scheme for SSDs. GC-ARM is motivated by a number of important observations. First, we observed the significant impact of the low GC efficiency problem to SSDs by evaluating two commercial SSDs. On the other hand, the existing write buffer management schemes are by and large oblivious of GC efficiency, thus failing to help improve performance for workloads that tend to induce low GC efficiency. GC-ARM can detect this situation using a *benefit value* to decide whether to destage a page or a block at a time. Second, based on the observed high write-back traffic resulting from address translation, GC-ARM groups the LPN-to-PPN mapping entries based on their LPN. Finally, the observation of the randomness of workloads varying over time and from workload to workload motivated the GC-ARM design that dynamically adjusts the size ratio between the write buffer and the cached mapping table. Extensive trace-driven evaluation results show that GC-ARM consistently outperforms BPLRU, DFTL and AP in terms of the number of erasures, average response time, GC efficiency and write traffic reduction.

Chapter 4

Identifying Performance Anomalies in Enterprise Environment

Recently, an increasing number of enterprise-level applications have employed SSD as a cache device between HDD (Hard Drive Disk) and main memory of the host machine to meet mission-critical requirements because of SSD's high random-access performance [10,19,32,47]. The unique workload characteristics of higher write intensity and more random accesses in enterprise applications have exposed anomalies and problems with SSD that rarely surface in the consumer-level application environment.

In this chapter, we conduct intensive experiments on a number of commercial SSD products from high-end PCI-E SSD to low-end SATA SSD. Based on the anomalies we identify, we obtain useful insights into SSD performance under enterprise-level environment where workloads tend to be more random and intensive. Based on the findings of the experimental evaluation study, we then provide insightful suggestions to designers and developers of enterprise-level applications to make the best use of SSDs and achieve and sustain a near-peak performance while avoiding some of the intrinsic SSD problems. Specifically, we identify and analyze the following problems

and anomalies.

4.1 Background

4.1.1 Low GC efficiency

Recent studies [11, 22, 26, 51] have revealed that the performance of an SSD falls precipitously as more data is being written to it, as a result of the slow erase operations and write amplification caused by garbage collection. This problem is far more pronounced in enterprise applications where workloads are more random, more intensive and have less temporal locality. Under a workload with little temporal locality, fewer addresses are updated, resulting in fewer pages being invalidated and thus less garbage being generated. On the other hand, highly random workloads will likely scatter the updated addresses sparsely all over the SSD. This means that most blocks in the SSD will contain far more valid pages than invalid ones by the time the GC process is triggered by a lack of free pages. Therefore, when the GC process starts the read-write-erase sequence, it spends most of its time erasing blocks with a high valid-to-invalid page ratio and copying valid pages in these blocks to somewhere else, significantly reducing the performance and GC efficiency. Moreover, the low GC efficiency leads to more frequent invocations of GC, thus reducing the lifetime of the SSD as well. In this chapter, we will show that with judiciously allocated space, it is possible to avoid this performance anomaly and achieve a sustained write bandwidth.

4.1.2 Predicting the residual lifetime of SSDs

The lifespan of a block in SSD is measured by the maximum number of erasures it can endure before failing to operate properly. This maximum number is 100,000 and 5,000

to 10,000 respectively for the SLC flash memory and the MLC flash memory [18]. It was reported that SSDs based on the MLC flash in an enterprise environment could easily wear out in 23 days of use [42], which is far too short to be useful. However, the lifespan of SSD blocks is convolutedly related to several key internal SSD strategies, such as wear-leveling, LBN-PBN mapping policy and GC strategy, that are unfortunately proprietary to SSD vendors. This makes it extremely difficult, if not impossible, for the users of an SSD to determine the SSD's residual lifetime. Therefore, in order to effectively deploy and use SSDs in enterprise applications, it is necessary and important to be able to predict – without knowing the vendor's SSD strategies – the lifetime left of an SSD to avoid the destructive loss of data and performance. In this chapter, based on the evaluation results, we develop an analytical model to predict the residual lifetime of an SSD.

4.1.3 High random read performance

The random-read performance of an SSD is much higher than that of an HDD. Therefore, SDDs have been employed as a cache device by some enterprise-level software products to improve their performance [10, 19, 32, 47]. However, the aforementioned GC-efficiency problem of SSD can significantly reduce the SSD performance, and both random-read and write performance are affected. Under write-intensive workloads where GC is called frequently due to low GC efficiency, the random-read performance of an SSD will severely deviate from that specified on the SSD's data sheets. In this chapter, we analyze the impact of the GC and its efficiency to the random read performance.

4.1.4 Achieving high bandwidth and low average response time simultaneously

Achieving a high bandwidth as specified on the data sheets of an SSD is one of the design goals and evaluation criteria of an SSD-based server used for an enterprise application. On the other hand, application users will also desire a low average response time. Thus, low average response time should be another important metric for both the client and the server. Although an SSD offers rich internal parallelism [8] that can serve multiple concurrent requests to achieve a high bandwidth as specified on the data sheets, maximizing the parallelism does not necessarily guarantee a minimal average response time. If the request size is too big or the queue length is too long, the average response time will be kept high no matter how high the bandwidth is. In this chapter, we find out a tradeoff between the bandwidth, average response time and the queue length.

4.2 Experiment

4.2.1 Experiment setup

Given the experimental nature of this study, this section first introduces the experimental platform used in this study. We then present the evaluation methodology that takes the unique features of SSD into consideration. We also briefly describe the evaluation tools used and the commercial SSD products evaluated in this study.

4.2.1.1 Experimental evaluation environment

Our experimental platform is a 2U Dell PowerEdge R710 Rack Server® equipped with an Intel® Xeon® 2.4GHZ processor and 16GB RAM. The operating system is

Linux RedHat enterprise server 5 with the 2.6.38 kernel.

4.2.1.2 SSD-specific considerations and evaluation methodology

The unique physical characteristics and more dynamic unpredictable behaviors of SSD make it substantially different from HDD when it comes to experimental evaluation. In particular, the following SSD-specific factors must be carefully considered in an experimental study of SSD.

Unlike HDD, the performance of SSD is *stateful* in that the evaluation results of one experiment run on an SSD are affected by the state of the SSD after the completion of the previous experiment run on it. This is because of the various internal physical characteristics and control strategies of SSD. Thus, it is necessary that each experiment starts from the same state of the SSD in order to make the evaluation repeatable and results meaningful. In this paper, we follow the Solid State Storage Test Specification devised by SNIA [44]. Specifically, in all our experiments, we issue a secure erase command to the SSDs before all the tests to make the SSDs start from the empty state.

Unlike HDD, the performance of SSD drops substantially after a period of sustained write operations due to garbage collection. Thus, the performance of an SSD is not only a function of the applied workload, but also a function of the amount of data written to it, suggesting that the evaluation must consider the timing and impact of GC. For our experiments, if the write or read performance is being evaluated, we only run the SSD for 15 seconds to obtain the performance results with a minimum GC effect.

SSD has an FTL module that maps an LPN to a PPN, which does not exist in HDD. The mapping entries of FTL are gradually populated as SSD starts to process write requests. For read requests, on the other hand, the LPNs of the requests are first

looked up in FTL to find their corresponding PPNs before the requests are sent to the device. Therefore, when evaluating the read performance of SSD, it is necessary to populate the mapping entries of the FTL module first to avoid inaccurate evaluation results. Otherwise, the read requests will never actually go to the flash memory if their mapping entries are not found. Therefore, in our experimental evaluation, we first populate all the mapping entries by sequentially writing to the SSD if read performance is to be evaluated.

SSDs are equipped with small on-board RAM to buffer either user data or LPN-to-PPN mapping entries. For all our experiment, we do not turn off the on-board RAM because we want to show the performance anomalies of the SSD instead of the NAND flash memory. The working sets (LPN range of the requests) of all the experiment are large and random enough so that the on-board RAM effect is minimized.

4.2.1.3 Evaluation tools

In this paper, we use the Intel Open Storage Toolkit® from the Intel labs. It contains an iSCSI initiator and target, a SCSI RAM disk, a block-level micro-benchmark, a performance monitor, and I/O tracing and replay. We use the block-level micro-benchmark to issue various workloads to the SSDs. The O_DIRECT flag is turned on to bypass the page cache in the operating system.

The Intel Open Storage Toolkit® only reports the overall performance results. In order to analyze various performance metrics such as response time of each request, real-time bandwidth, etc., we use the blktrace tool to intercept all the requests and events from the Intel Open Storage Toolkit and then process the intercepted requests and events using the blkparse tool.

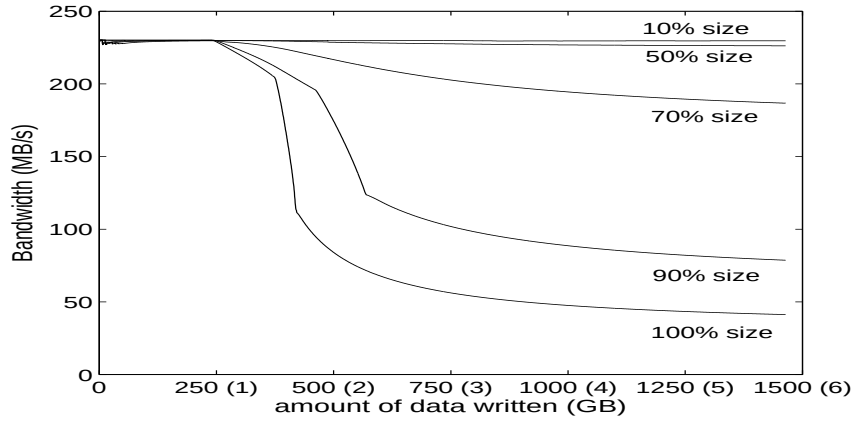
Table 4.1: Specifications of SSDs

Parameters	Intel 320	Samsung 470	OCZ Vertex 3	Fusion IO ioDrive
Chip	MLC	MLC	MLC	MLC
Capacity(GB)	120	250	120	320
Read bandwidth(MB/s)	270	250	280	735
Write bandwidth(MB/s)	220	220	260	510
Read latency(μs)	75	N/A	50	10
Write latency(μs)	90	N/A	16	7

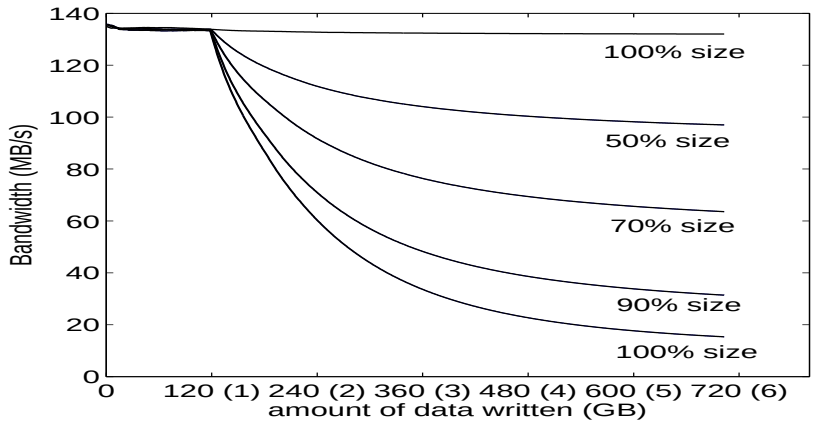
4.2.1.4 Commercial SSD products evaluated

Many commercial SSD products are available on the market and most of them are SATA based. In this paper, we choose 3 SATA based SSDs (Intel 320, Samsung 470 and OCZ Vertex 3) and one PCI-E based SSD (Fusion IO ioDrive). For the SATA based SSDs, all their functional modules run in the firmware. The PCI-E based SSD we chose keeps its various functional modules running in the host. The parameters of the evaluated SSD products are listed in Table 4.1. Note for OCZ Vertex 3 and Fusion IO ioDrive, we calculate the response time of read and write based on the IOPS they provide.

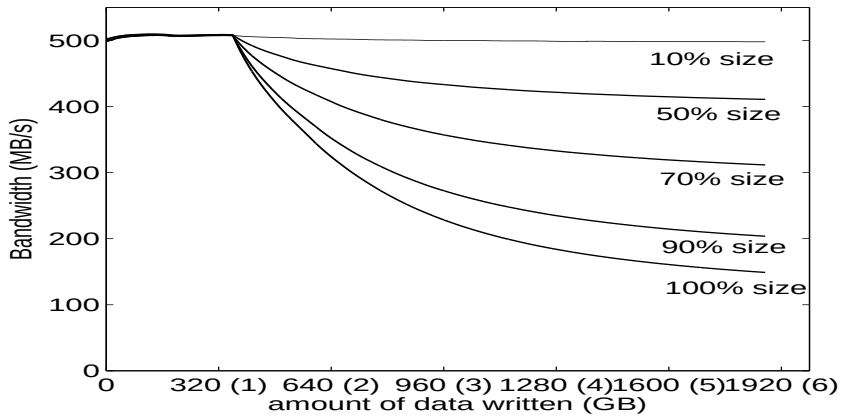
In this section, we first discuss how GC affects the performance at different levels of GC efficiency by evaluating three SSDs. We then show that different mapping policies of FTL have different responses as more data is being written, which can help conjecture the mapping policy of a given SSD. Further, based on the conclusion drawn from the impact of GC efficiency, we are able to develop an analytical model that predicts the residual lifetime of an SSD as a function of SSD capacity, the reserved LPN range, the amount of data written, and the maximal erase cycles of the flash memory. We then show that the response time of random reads is highly unpredictable, which is contrary to the common belief. Finally, we strike an optimal balance between the queue length and the request size to achieve the maximal bandwidth while minimizing the average response time.



(a) 250GB Samsung 470 series SSD



(b) 120GB Intel 320 SSD



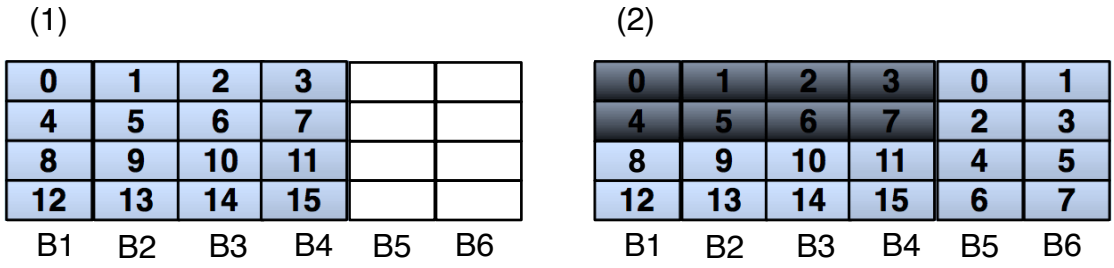
(c) 320GB Fusion IO ioDrive

Figure 4.1: Performance impact of GC efficiency in SSDs – bandwidth as a function of the amount of data written and the percentage of reserved LPN range

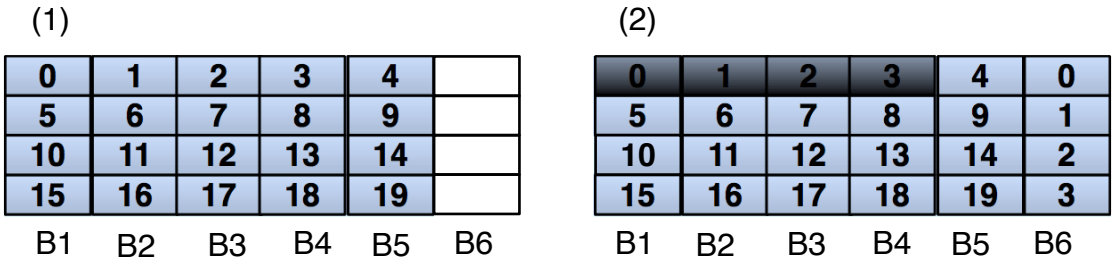
4.2.2 Performance impact of GC efficiency

It is widely accepted that as more data is being written to SSD, the performance drops sharply because of the GC process [22, 51]. In the enterprise applications, where workloads are highly write intensive, this problem is believed to be more pronounced. However, we will show that this is not necessarily true. By manipulating the space allocation of SSD, it is possible to avoid or control this significant performance degradation to some extent.

Figure 4.1 shows the bandwidth as a function of the amount of data written to Samsung 470 series SSD, Intel 320 SSD and Fusion IO ioDrive, respectively. For each of the SSDs, we customize the workloads by issuing 64KB random write requests with their LPNs within 0-10%, 0-50%, 0-70%, 0-90% and 0-100% respectively of the maximal LPN range of the SSDs. The amount of data written is set to be 6 times the full capacity of the SSD. The numbers in the parentheses of the x axis indicate the amount of data written divided by the full capacity of the SSD. By customizing the workloads this way, we can control the amount of invalidated pages generated when the SSD is full. For example, when the request addresses are set within 0-10% of the LPN range, more pages are invalidated when the SSD is full than in the 0-100% case when the same amount of data is written. This is because pages are more frequently updated in the 0-10% case. This is more clearly illustrated in the simple example of Figure 4.2. The SSD has 6 blocks, each of which consists of 4 pages. The invalidated pages are shaded in dark grey. In Figure 4.2(a), LPNs 0-15 are reserved for use among the full LPN range of 0-23, so 0-66% LPN range is used. The reserved range is shaded in light grey for clarity, although it must be noted that in reality the reserved range is not fixed physically since it represents a portion of the logical address space, not the physical one. In (1) of Figure 4.2(a), B1, B2, B3, B4 are randomly written with pages



(a) 0-66% LPN reserved



(b) 0-83% LPN reserved

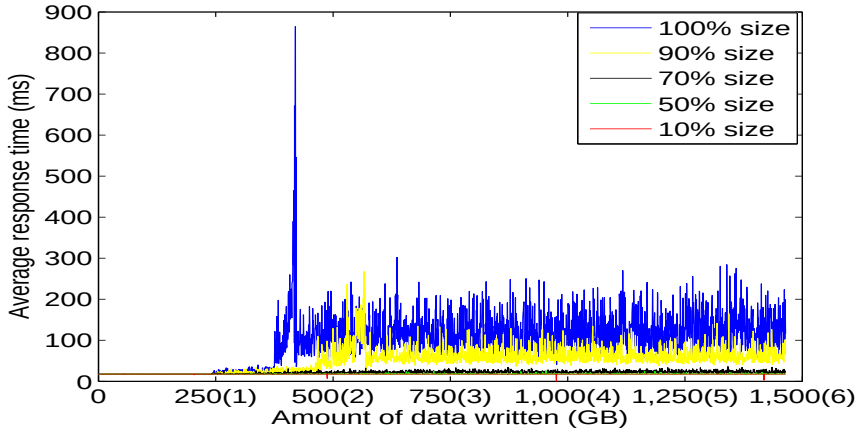
Figure 4.2: Examples illustrating GC efficiency when different LPN range is reserved

within the reserved LPN range, LBNs 0-15. In (2) of Figure 4.2(a), pages within the reserved LPN range, LBNs 0-7, are written again (updated) in B5 and B6 because of out-out-place update, invalidating their previous versions, darkly shaded in the figure, contained in blocks B1 through B4. The SSD is now full and the GC process is triggered, where each victim block contains 2 invalid pages and each erase operation frees two pages. On the other hand, with a higher percentage of reserved LPN range, say, 0-83%, as shown in Figure 4.2(b) where LPNs 0-19 are reserved among the full LPN range of 0-23, when GC is triggered, each erase can only free 1 page, decreasing the GC efficiency from 50% in the case of 4.2(a) to 25%. Here we consider GC to be 100% efficient if each erase operation frees 100% of pages contained in the erased block (i.e., the block contains 100% invalid pages before erase).

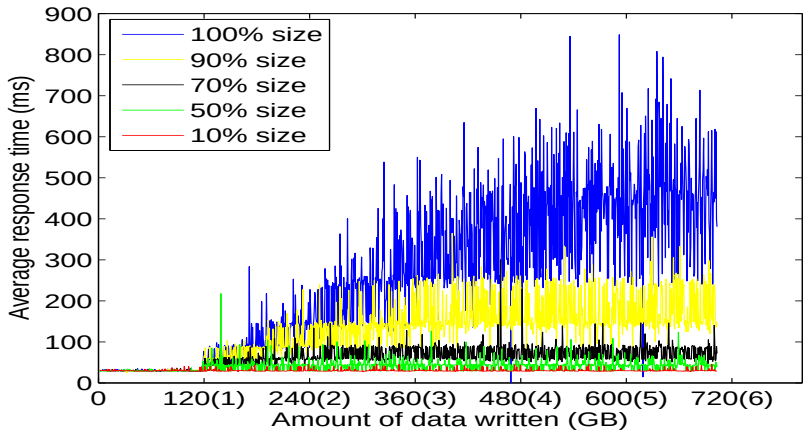
As shown in Figure 4.1, for all the three SSDs, before they are full the bandwidth remains unchanged and is independent of the percentage of reserved LPN range. When they are full all free pages are consumed by the write requests. This is because before an SSD is full, it can process the write requests immediately by using its available free pages no matter what the requests' addresses are.

However, after the three SSDs are full, that is, the amount of data written is more than 250GB for Samsung 470 series, 120GB for Intel 320 and 320GB for Fusion IO ioDrive, the bandwidth drops at very different rates depending on the GC efficiency. The rate is the highest when the LPNs of write requests are within 0-100% of the LPN range. For the Intel 320 SSD, the bandwidth can drop to as low as 12% of its peak value. Even for the best case, Fusion IO ioDrive, the bandwidth drops to 30% of its peak. The reason lies in the fact that LPNs of the write requests scatter all over the address space of SSD, making it less likely for the same LPN to be updated and thus generating fewer invalidated pages. Therefore, when GC starts the read-write-erase sequence, it has to copy many valid pages but only free a very limited number of invalid pages when victim blocks are erased. This is the case where the GC efficiency is low and the write amplification is high. To make things worse, in this case, the slow GC process will also be triggered very frequently to free enough pages. Similarly, we also show the performance impact of GC efficiency to the write response time. As shown in Figure 4.3(a) and Figure 4.3(a), when the LPNs of write requests are within 0-100% of the LPN range, most of the write requests have very high response time because GC is always in process and GC efficiency is low. Fusion IO ioDrive is a very high-end SSD. Therefore, even if the LPN reserved is 0-100%, the response time of most write requests is still low. However, we can also see very high spikes of the write response time when 0-100% LPN is reserved, as shown in Figure 4.3(c).

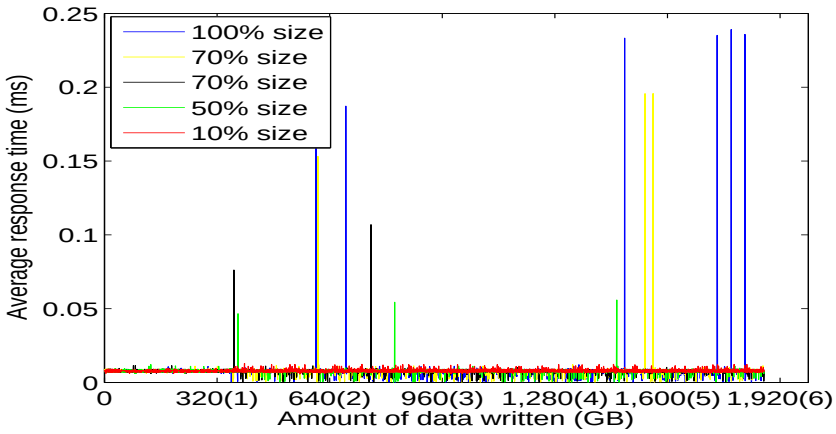
On the other hand, when the LPNs of write requests are within 0-10% of the LPN



(a) 250GB Samsung 470 series SSD



(b) 120GB Intel 320 SSD



(c) 320GB Fusion IO ioDrive

Figure 4.3: Performance impact of GC efficiency in SSDs – write response time as a function of the amount of data written and the percentage of reserved LPN range

range, the write bandwidth of all the three SSDs hardly decreases as more data is written to them. This is because the LPNs of the write requests concentrate within a small percentage of the LPN range. In this case, when all the free pages are consumed, most pages are likely to have been updated once or multiple times. Therefore, when the GC starts the read-write-erase sequence, it can easily find victim blocks that have a high percentage of invalidated pages, increasing the GC efficiency and reducing write amplification. Moreover, GC will be triggered less frequently because each GC frees much more pages in this case than in the 0-100% case. The performances of the 0-50%, 0-70%, 0-90% cases fall in between the two extremes. The 0-50%, 0-70%, 0-90% and 0-100% cases also indicate that the working sets of the customized workloads are large and random enough to void the on-board RAM effect because the bandwidth of the four cases start to drop at the same time when the SSDs are full. Otherwise, the bandwidth of 0-50% case should start to drop later than the 0-70%, 0-90%, and 0-100% cases because the on-board RAM can absorb some write requests. Similarly, for the write response time, when the LPN range reserved is 0-90%, 0-70%, 0-50% and 0-10%, the response time of the write requests drop significantly because the GC efficiency is higher than the 0-100% case.

Our experimental evaluation results show that the widely accepted belief that the performance of SSD drops significantly as more data is written to it is not always true. On the contrary, in some cases where the workloads keep writing to a small percentage of the LPN range, the bandwidth and response time do not drop at all. In the enterprise-level application environment, however, the high write-intensity and access randomness of the workloads make them behave much more like the 0-100% case where writes are intensive and update requests are much fewer, leading to a very low GC efficiency and high write amplification. To avoid or alleviate the precipitous bandwidth and response time drop regardless of how much data is written to the

SSD, it is desirable to judiciously limit the writable LPN range of the SSD to strike a sensible tradeoff between performance and addressable space efficiency. For example, we can use 0-50% of the LPN range of a 100GB SSD to achieve a good write bandwidth and response time by improving the GC efficiency, but at the cost of sacrificing 50GB of the SSD's addressable storage space. In enterprise applications where SSDs are used as a cache device, this is a feasible and reasonable solution to achieve and sustain the peak performance.

4.2.3 Performance impact of the mapping policy of FTL

Different LPN-to-PPN mapping policies tend to generate different performance curves as more data is written to the SSD. Among the many FTLs proposed [11] in the past, the most flexible *page-mapping FTL* [3] keeps all the LPN-to-PPN mapping entries of the SSD address space in RAM, requiring a large RAM space. GC is periodically invoked to erase victim blocks, move valid pages and collect invalidated pages to make free pages to accommodate new requests. Though there are several variants for page-mapping FTLs [11], they all suffer from the intrinsic and unavoidable problem. That is, the performance is reduced significantly when free pages are used up and the GC efficiency is low.

The most inflexible *block-mapping FTL* [4] reduces RAM consumption significantly by mapping a request to a fixed offset of a block, thus requiring an expensive read-write-erase sequence each time a page is updated (recall Section 2.1).

Several *log-block based hybrid FTLs* [25, 29, 30] have been proposed to reduce the high RAM consumption of the page-mapping FTL and high-overhead of the block-mapping FTL. The basic idea of log-block based FTLs is to forward all write requests to several reserved log blocks and when log blocks are full, it combines valid pages

in both the log block and its corresponding data block, an operation call *full merge*. Because the number of log blocks is very limited [23, 48], the process of full merge is always in progress.

Figure 4.4 shows the bandwidth as a function of the amount of data written to the OCZ Vertex 3 SSD. The configuration is the same as those in Figure 4.1 except we only write 3 times the capacity of the device because we find that the bandwidth remains unchanged as more data is written, independent of the percentage of the reserved LPN range, which is completely different from what observed in Figure 4.1 and discussed in Section 4.1. Based on the results, we speculate that the mapping policy employed in the OCZ vertex 3 SSD is most likely log-block mapped, where the continuous full-merge process creates a steady flow of free pages and thus stabilizes the bandwidth. On the other hand, we speculate that the Samsung 470 series, Intel 320 and Fusion IO ioDrive SSDs are most likely all page mapped because, as shown in Figure 4.1, their bandwidth drops sharply after they are full due to low GC efficiency. When the GC efficiency is low, the bandwidth continues its downward spiral, a process that is likely irreversible.

Our experimental evaluation results of Figure 4.1 and Figure 4.4 clearly illustrate the dynamics of four different SSDs, which in turn helps us predict the mapping policies employed in them. It should be noted that the mapping policies in commercial products are more complicated because they are often affected by other internal operations such as wear leveling, buffering/caching, and prefetching. Nevertheless, the insight gained from our experimental study can still be helpful in speculating what the most likely mapping policy a given SSD employs.

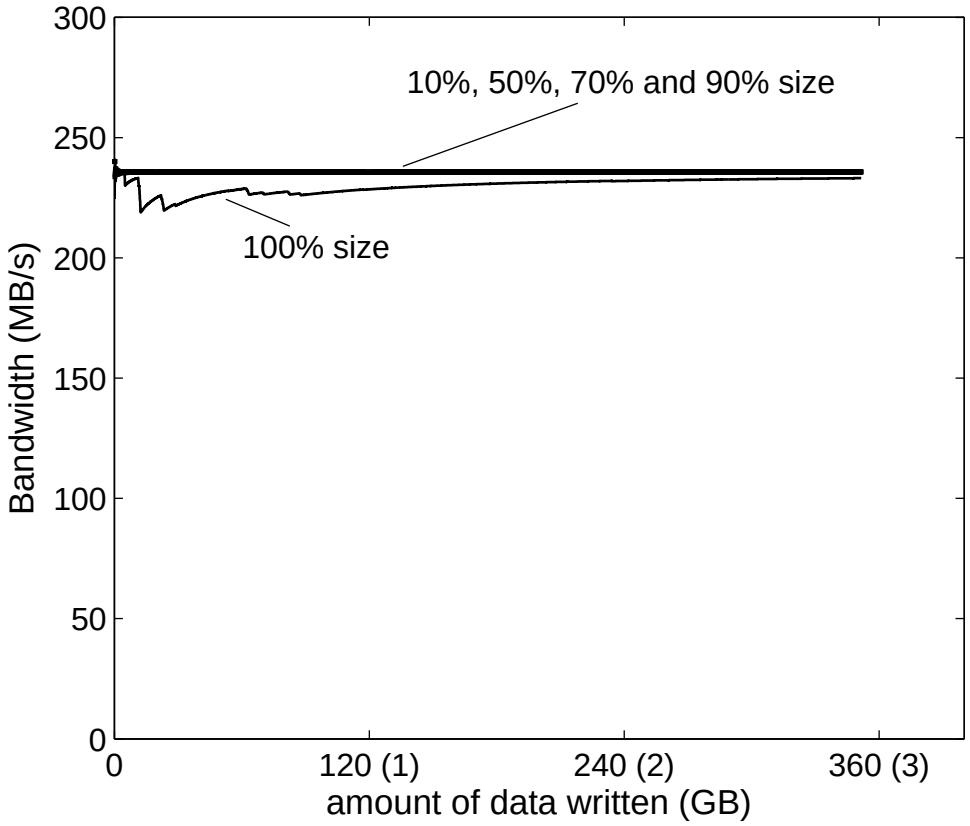


Figure 4.4: Impact of mapping policies – bandwidth of 120GB OCZ Vertex 3 SSD as a function of the amount of data written and the percentage of reserved LPN range

4.2.4 Predicting the residual lifetime of SSD based on a predefined space allocation scheme

The importance of data integrity and reliability in the enterprise applications implies that it is critical to predict the residual lifetime of the SSDs, especially when they are used as cache devices that lie in the critical path of I/O operations and are not protected by other data protection methods such as RAID [35]. With some knowledge of the residual lifetime of an SSD, it is possible to take appropriate proactive or preemptive measure to protect the integrity and reliability of data in it. Based on

the space allocation scheme (i.e., the percentage of reserved LPN range) that can be controlled and the mapping policy that can be inferred (recall Section 4.2), we develop an analytical model to predict the residual lifetime as a function of the SSD capacity, the reserved LPN range, the amount of data written, and the maximal erase cycles of the flash memory. The analytical model, in its current form, only applies when the mapping policy of the SSD is page mapped. The variables used in Equation 4.1 are defined below.

- S: SSD capacity
- P_{size}: page size
- $N_{page} = \frac{S}{P_{size}}$: total number of physical pages in SSD
- P: the percentage of the reserved LPN range of the SSD. e.g., 0.9 if 0-90% of the full LPN range is used ($0 < P < 1$)
- A: Amount of data written to SSD in GB
- M: Maximal erase cycles, e.g., $10,000$ for MLC flash memory and $100,000$ for SLC flash memory

$$ResidualLifetime = 1 - \frac{A}{S \cdot [P + (1 - P) \cdot M]} \tag{4.1}$$

The model is derived as follows. We first make the following simplifying but justifiable assumptions. First, we assume that the SSD is new (i.e., no data has been written to it) to begin with and the only background operation is GC because GC arguably dominates the background/internal operations in SSD. Second, we assume that the write requests are random within the LPN range reserved for use, i.e., 0 to P of the full LPN range because of the high access randomness of enterprise-level

applications. Therefore, the invalid pages are uniformly distributed among all blocks and the average number of invalid pages in each block is the same. We further assume that, for simplicity, when a victim block is erased, the valid pages in it are first copied somewhere else (e.g., RAM) and, after the block is erased, copied back to the same block that now contains no invalid pages.

Write requests first propagate from the 0-to-P portion of the full PPN range of the SSD and, once this portion is full, then continue to propagate the remaining $1 - P$ portion of the SSD PPN space. At this time, all the PPNs are used and the percentage of invalid pages in each block is $(1 - P) \cdot 100$ because the latter requests are all updates (recall Figure 4.2). Then GC starts to collect these $(1 - P) \cdot N_{page}$ invalid pages and the blocks containing these invalid pages are erased once as a result. After this point, every time $(1 - P) \cdot N_{page}$ pages are written, the same number of invalid pages are generated, causing GC to be triggered to collect these invalid pages by erasing the corresponding blocks once. This process repeats until all the blocks are erased M times. Therefore, the maximal amount of data that can be written to the SSD is $N_{page} \cdot P_{size} \times P + N_{page} \cdot P_{size} \times (1-P) \times M = S \cdot P + S \cdot (1-P) \times M$. The residual lifetime of SSD is then calculated based on how much data is written to the SSD, as shown in Equation 4.1, in terms of the percentage of lifespan. Therefore, we can see that the residual lifetime is not only a function of the endurance of the type of the chip (MLC or SLC), but also a function the LPN range reserved for use. The smaller the range reserved, the longer the SSD can be used because of more efficient GC – at the cost of reduced addressable space utilization.

Figure 4.5 illustrates the predicted residual lifetime of a 100GB MLC SSD as a function of the reserved LPN range and the amount of data written based on the analytical model. We can see when the reserved LPN range is small, say, less than 60%, the residual lifetime does not decrease significantly when more data is being written.

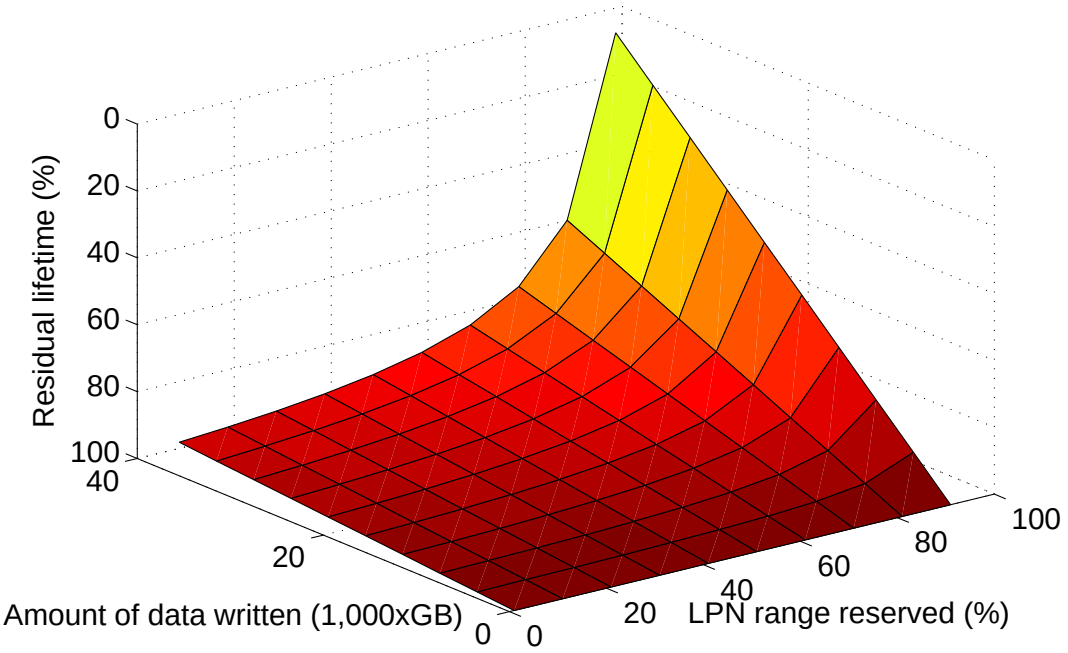
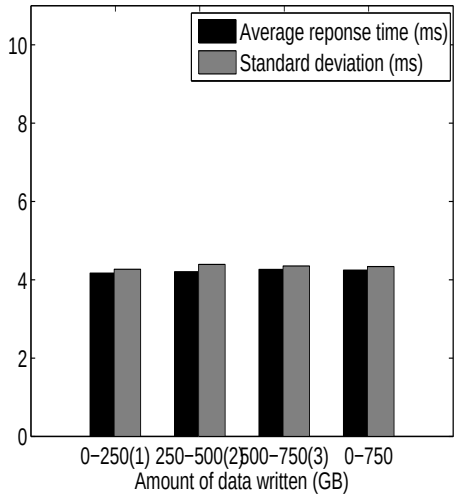


Figure 4.5: Residual lifetime as a function of reserved LPN range and the amount of data written

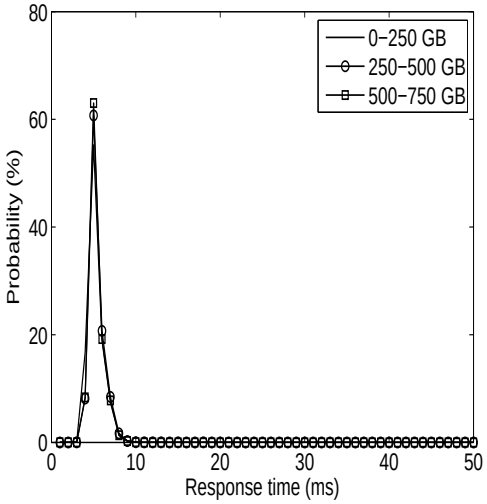
However, when the reserved LPN range is more than 80%, the residual lifetime is reduced more quickly as more data is being written. Therefore, the reserved LPN range is a key factor in keeping the SSD running longer. This conclusion, combined with the evaluated impact of the GC efficiency on the bandwidth, suggests that enterprise application developers can choose a medium reserved LPN range to prevent the bandwidth from dropping precipitously while ensuring a longer SSD endurance.

4.2.5 Unpredictability of random read performance

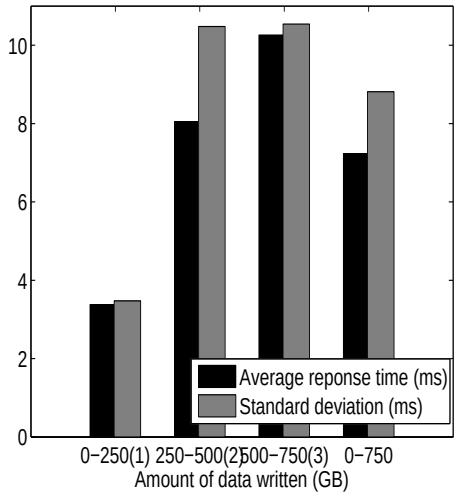
SSD has much higher random-read performance than HDD. This is in part why SSD is used as a cache device under random and intensive workloads in the enterprise environment. Most of the data sheets of SSDs only provide a single average value



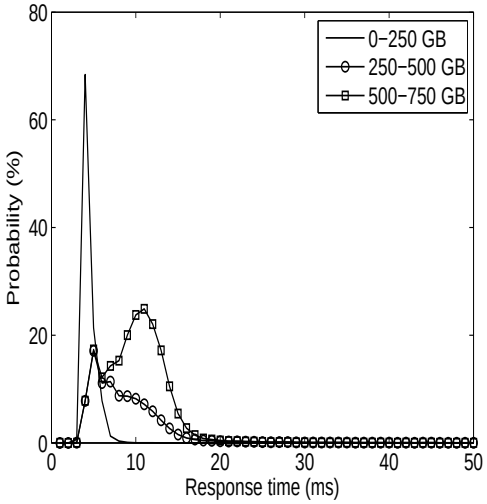
(a) 0-10% of the LPN range



(b) pdf of response time when 0-10% LPN is reserved



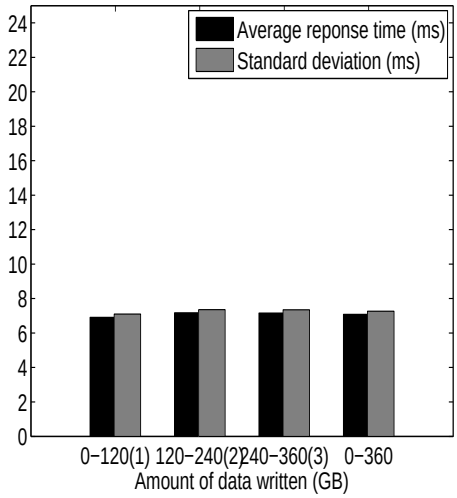
(c) 0-100% of the LPN range



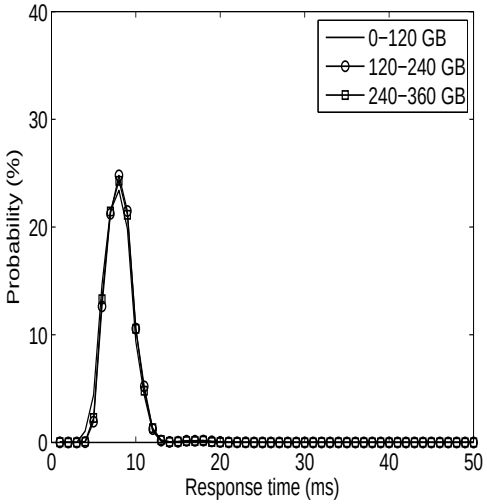
(d) pdf of response time when 0-100% LPN is reserved

Figure 4.6: Average response time, standard deviation and distribution of response time of the Samsung 470 Series SSD

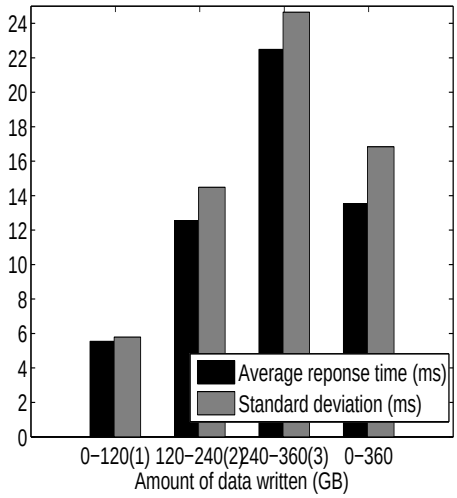
on response time of read requests. This value, however, turns out to be inaccurate, as revealed by our experimental evaluation of the commercial SSD products. In fact, the read response time of an SSD depends highly on the scheme and efficiency of its GC.



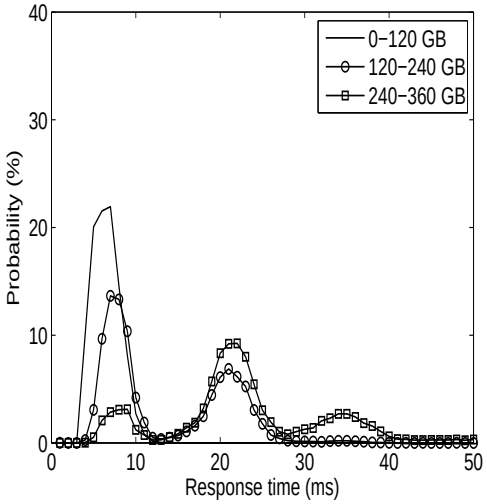
(a) 0-10% of the LPN range



(b) pdf of response time when 0-10% LPN is reserved



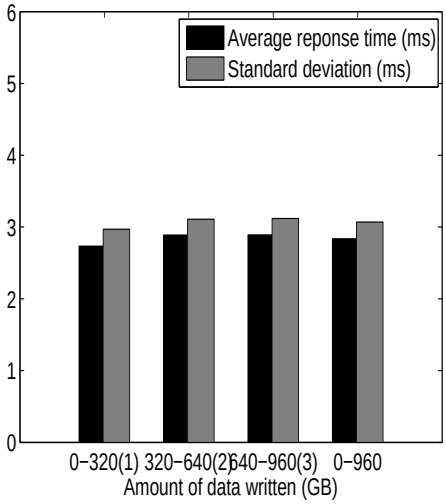
(c) 0-100% of the LPN range



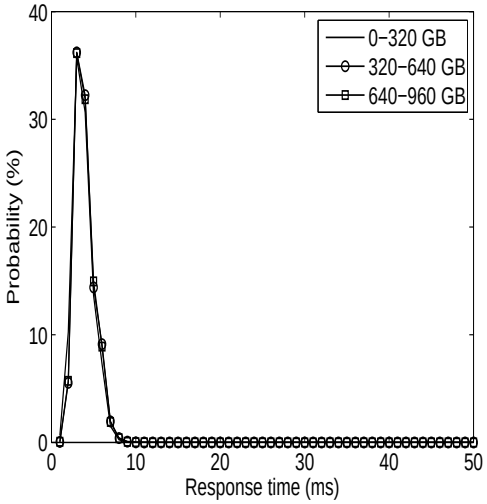
(d) pdf of response time when 0-100% LPN is reserved

Figure 4.7: Average response time, standard deviation and distribution of response time of the Intel 320 SSD

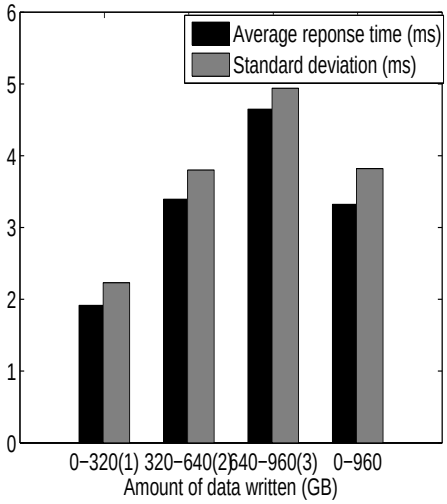
In order to evaluate the read response time at different stages of GC, we mix the read and write requests and set the read and write requests ratio to 1:1. Both the read and write requests are random and their request size is 64KB. Further, in order to evaluate the read response time at different levels of GC efficiency, we customize



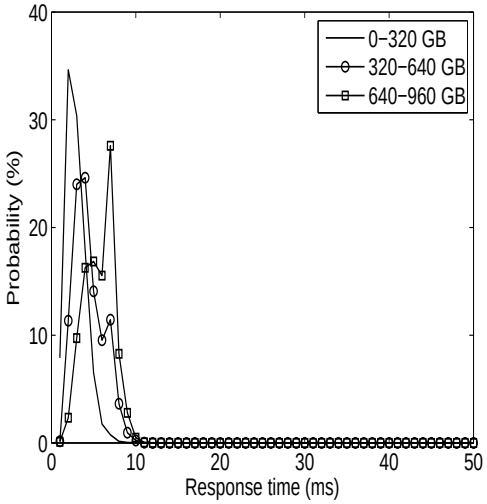
(a) 0-10% of the LPN range



(b) pdf of response time when 0-10% LPN is reserved



(c) 0-100% of the LPN range



(d) pdf of response time when 0-100% LPN is reserved

Figure 4.8: Average response time, standard deviation and distribution of response time of the Fusion IO ioDrive

the workloads by issuing requests with their LPNs confined to the ranges of 0-10%, 0-50%, 0-70%, 0-90% and 0-100% respectively of the maximal LPN range of the SSDs, that is, the same percentages of reserved LPN range as defined previously. We write 3 times the full capacity of each of the SSDs. Because the results of the cases of 0-50%,

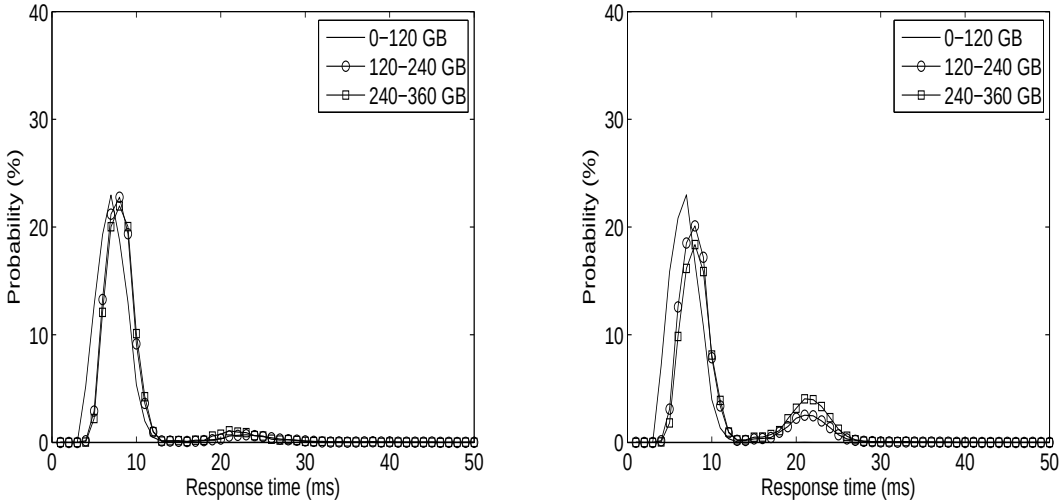
0-70% and 0-90% fall in between those of the two extreme cases of 0-10% and 0-100%, we only show the latter cases. We evaluate the average response time, standard deviation and the probability density function (pdf) of read requests as a function of the amount of data written, that is, less than 1 times the full capacity, between 1 and 2 times the full capacity, and between 2 and 3 times of the full capacity, respectively. The following two main observations drawn from the experimental results show how significant an impact GC has on the performance of read requests.

(1) *GC and its efficiency significantly affect the average response time of read requests.* Figure 4.6 through Figure 4.8 show the average response time of read requests as a function of the amount of data written to the SSDs and the percentage of the reserved LPN range of the requests. For all the SSDs, when the LPNs of requests are within 0-10% of the maximal LPN ranges of the SSDs, the average response time of read requests is almost completely independent of the amount of data written, as shown in Figure 4.6(a), Figure 4.7(a) and Figure 4.8(a), which indicates that as long as GC efficiency is high, the average response time of read requests is more predictable and can be kept steady no matter how much data is written to the SSDs. On the other hand, when the LPNs of requests are within 0-100% of the capacity of the SSDs, as shown in Figure 4.6(c), Figure 4.7(c) and Figure 4.8(c),the average response time of read requests increases sharply for all the three SSDs. In the worst case, which occurs in the Intel 320 SSD, the average response time during the period when the amount of data written is between 2 and 3 times of the SSD capacity is 5 times higher than during the period when the amount of data written is less than the SSD capacity. The reason is similar to the explanations of GC efficiency, that is, the GC process of the read-write-erase sequence is triggered frequently at a very low GC efficiency. The GC process frees very few pages each time and slows down the processing of the read requests after the amount of data written exceeds the SSD capacity (i.e., 250GB for

the Samsung 470 series SSD, 120GB for the Intel 320 SSD and 320GB for the Fusion IO ioDrive).

(2) *GC and its efficiency significantly affect the standard deviation of read requests.* When GC efficiency is high, as shown in Figure 4.6(a), Figure 4.7(a) and Figure 4.8(a), the standard deviation remains stable as data is written, which indicates that the response time of each read request is steady as long as the GC efficiency is high. This is more clearly shown in Figure 4.6(b), Figure 4.7(b) and Figure 4.8(b). The response times of read requests during the period when the amount of data written is less than the capacity, between 1 and 2 times of the capacity and between 2 and 3 times of the capacity all follow in the same normal distribution, which indicates that the standard deviation does not change as more data is being written and the response time of read requests is easily predictable. On the contrary, when GC efficiency is low, as shown in Figure 4.6(c), Figure 4.7(c) and Figure 4.8(c), the standard deviation increases sharply as more data is written, which indicates that low GC efficiency makes the response time of read requests very unstable and fluctuate greatly. This is again more clearly shown in Figure 4.6(d), Figure 4.7(d) and Figure 4.8(d). For all three SSDs, except for the period when the amount of data written is less than the capacity of the SSDs, the height of the curves decreases and the curves move to the right, which indicates that the response time of read requests not only increases, but also fluctuates dramatically as the amount data written grows beyond the capacity of the SSDs.

The GC efficiency at which the response time of read requests starts to become unpredictable varies among different SSDs. In this paper, we only show the results of Intel 320 SSD. Figure 4.9 shows the response time of read requests of Intel 320 SSD when the reserved LPN range is 0-50% and 0-70%. We can see that, in the former case shown in Figure 4.9(a), there are two spikes during the period when the amount of data written is between 1 and 2 times of the SSD capacity and 2 and 3 times



(a) pdf of response time when 0-50% LPN is reserved

(b) pdf of response time when 0-70% LPN is reserved

Figure 4.9: Average response time of read requests of Intel 320 SSD at different GC efficiency

of the SSD capacity, which indicates that the response time of read requests starts to become unpredictable. When the reserved range increases to 70%, as shown in Figure 4.9(b), the spikes are much higher, which indicates the response time of reads is already very unpredictable.

4.2.6 Achieving high bandwidth and low average response time simultaneously

Recent studies have indicated that the rich internal parallelism of SSD can be exploited to achieve the peak bandwidth (i.e., data processing throughput). Parallelism is maximized by issuing as many requests as possible and increasing the request size [8]. However, from an individual client/user's point of view, low average response time is more preferred. Our study attempts to determine an optimal combination of the request queue length and request size for achieving a high bandwidth while

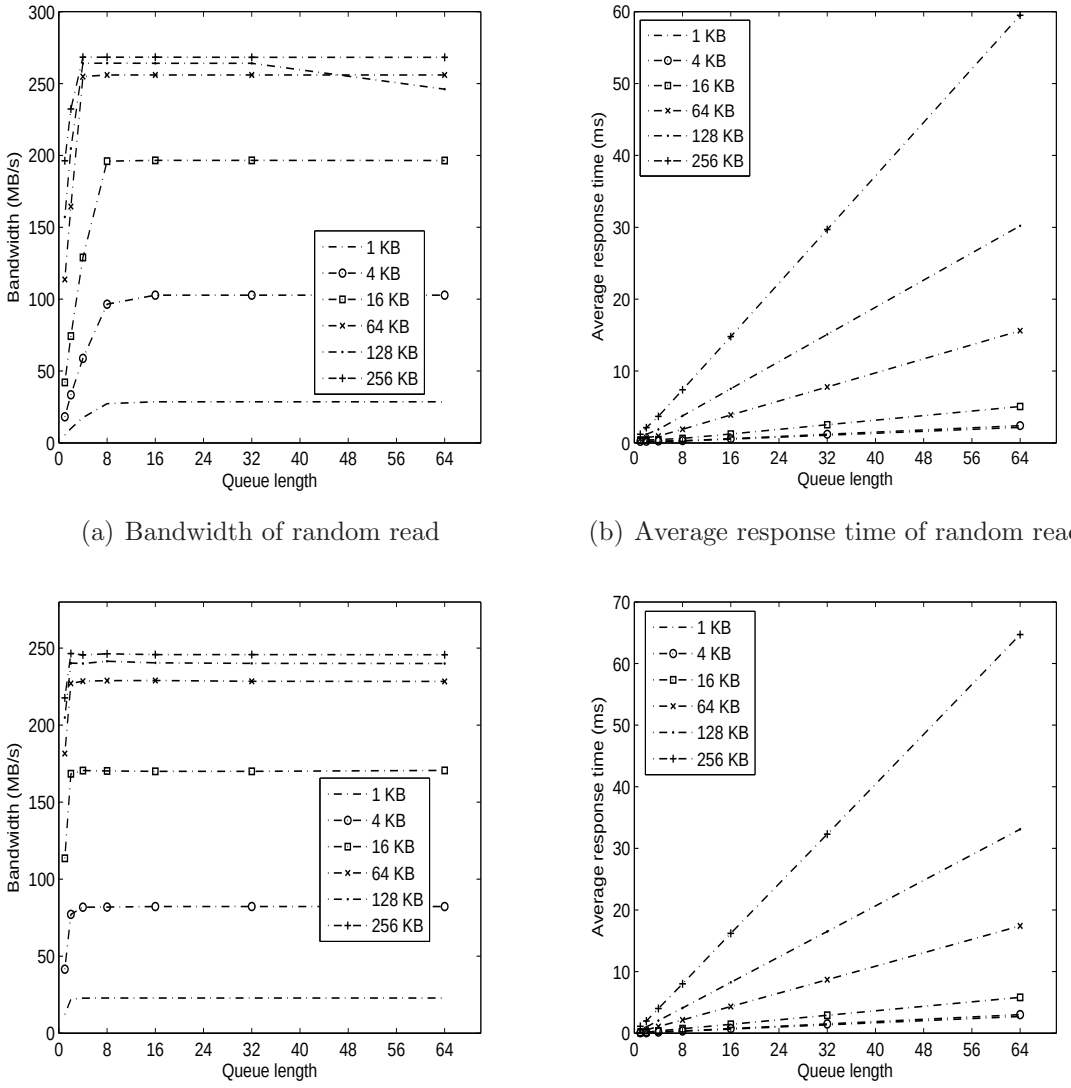
minimizing the average response time for a given SSD.

We evaluate both the bandwidth and the average response time as a function of queue length of the requests, with the request size being set to 1KB, 4KB, 16KB, 64KB, 128KB and 256KB respectively. Because the evaluation results for sequential reads and sequential writes followed the same pattern as random reads and random writes, we only present the evaluation results of the latter.

As expected, for all the workload patterns, the bandwidth increases with queue length increases for all the SSDs. This conclusion is consistent with previous studies [8]. More importantly, we find that the maximal bandwidth that can be achieved is determined by the request size, not the queue length. For all different request sizes, the maximal bandwidth determined by the request size is quickly reached after the queue length grows to merely 8 or more. This indicates that the request size is the main factor determining the maximal bandwidth. Moreover, for both random reads and random writes, the peak bandwidth can be reached when the request size is more than 64KB. This runs counter to the common belief that the request size should be increased as much as possible.

In summary, the average-response-time evaluation counters the common intuition that the queue length and request size should be increased as much as possible to improve the performance. Further, these increases actually hurt the average-response-time performance. As we can see from the average-response-time evaluation results, the average response time increases linearly as the queue length increases. Further, the bigger the request size is, the faster the average response time increases, as illustrated by the much steeper slopes of the curves of larger request sizes than those of smaller ones.

Based on the bandwidth and average-response-time evaluation results, a request size of 64KB and queue length of 8 seem to be an optimal combination for achiev-



(a) Bandwidth of random read

(b) Average response time of random read

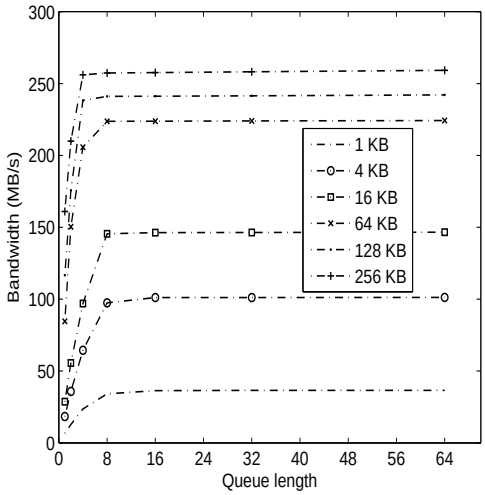
(c) Bandwidth of random write

(d) Average response time of random write

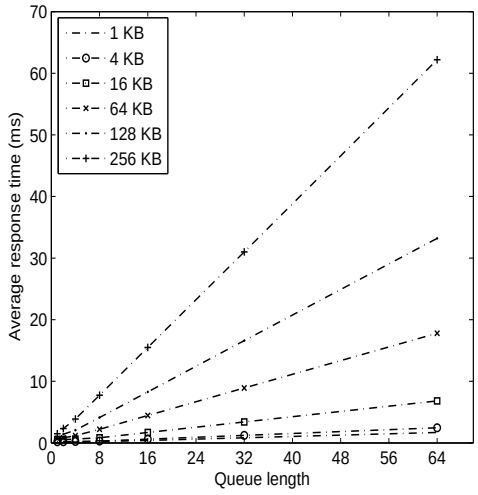
Figure 4.10: Tradeoff between bandwidth and average response time of Samsung 470 SSD

ing the peak bandwidth without significantly increasing the average response time. Further increasing the queue length and request size will only increase the average response time without any further benefit to the bandwidth.

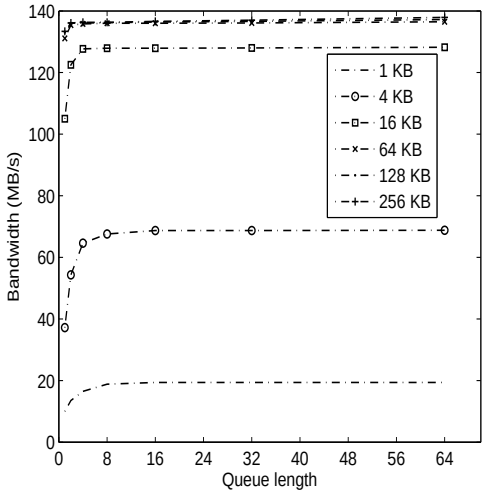
The reason behind this particular combination is that these SSDs have limited resources to process requests larger than 64KB. In other words, they can directly



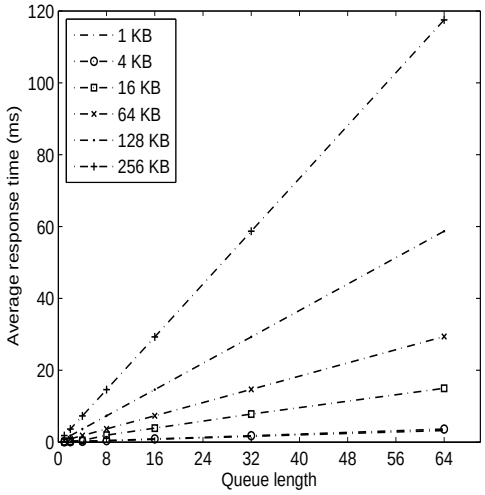
(a) Bandwidth of random read



(b) Average response time of random read



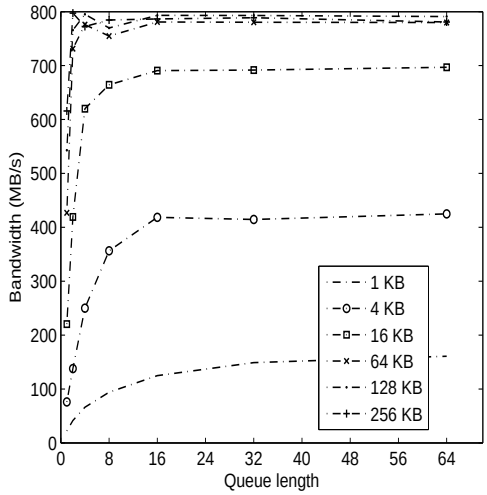
(c) Bandwidth of random write



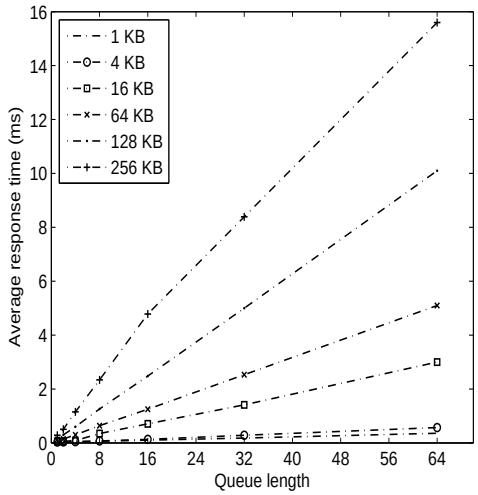
(d) Average response time of random write

Figure 4.11: Tradeoff between bandwidth and average response time of Intel 320 SSD

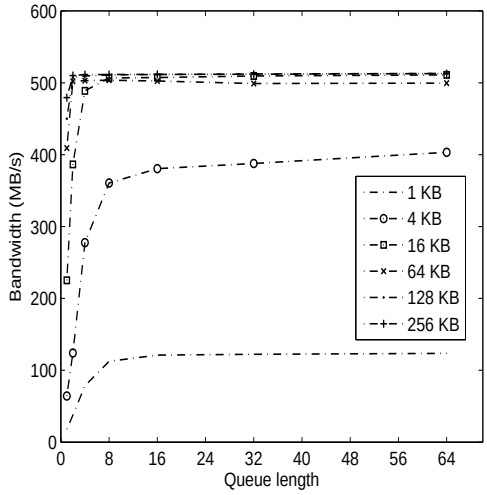
break requests smaller than or equal to 64KB into pages to process in parallel, but not for requests larger than 64KB. This is why we see that the bandwidth increases with the request size until the latter reaches 64KB.



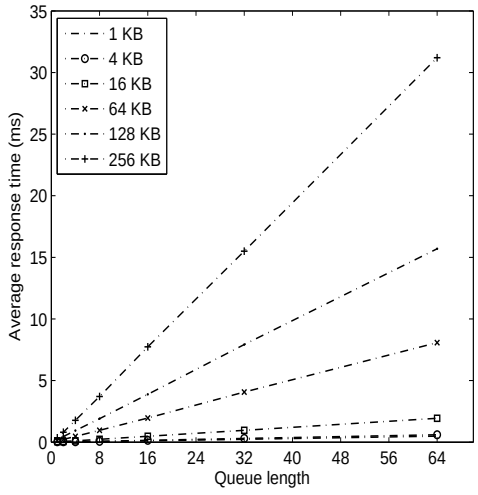
(a) Bandwidth of random read



(b) Average response time of random read



(c) Bandwidth of random write



(d) Average response time of random write

Figure 4.12: Tradeoff between bandwidth and average response time of Fusion IO ioDrive

4.3 Guidelines and suggestions

Drawing on insights gained from our experimental study, in this section, we further provide some useful suggestions to enterprise application developers and SSD vendors for possible performance improvement and optimization.

Improving performance by allocating a reserved fractional LPN range: GC efficiency has a significant impact on the performance of SSD and can be controlled by allocating a reserved fractional LPN range of the full LPN range for use. Smaller reserved range tends to improve the GC efficiency, which in turn prevents the bandwidth from dropping to some extent. In the case where SSD is used as a cache device, system developers can dynamically tune the reserved LPN range based on the temporal locality of workloads. When the temporal locality of workloads is high, meaning that the same LPNs tend to be updated repeatedly, the number of invalid pages will be high, resulting in a high GC efficiency and relatively stable bandwidth. In this case, system developers can reserve a larger LPN range. On the other hand, if the temporal locality of workloads is low, the number of invalid pages will be low, giving rise to a low GC efficiency and sharp bandwidth drops. For this case, system developers can reserve a smaller LPN range to improve the low GC efficiency and prevent the bandwidth from dropping at the cost of losing some addressable space. The performance and the addressable space can be traded off one way or another, depending on which metric is more important for the overall system performance. SSD vendors do not have the knowlege of the temporal locality of workloads. Therefore, for SSD vendors, more aggressive resource provisioning is recommended in order to keep a sustained bandwith even when the workloads do not have any temporal locality.

Making judicious use of SSDs with different mapping policies: Different LPN-PPN mapping policies of FTL behave very differently. For system developers, it is important to know the SSD's mapping policy and its behaviors in advance to make the best use of SSDs. For example, if the mapping policy of an SSD is log-block-mapped, system developers do not need to consider the reservation of fractional LPN range or the temporal locality of the workloads because the bandwidth does not drop even if the temporal locality of the workloads is low. On the other hand, if the

mapping policy of an SSD is page-mapped, system developers must consider these two factors to maintain a good bandwidth and make the space usage efficient.

Striking a sensible tradeoff among performance, endurance and space: We have shown that low GC efficiency reduces both the performance and endurance of the SSD. By reserving a LPN range that is a fraction of the full capacity of the SSD, the GC efficiency can be improved significantly, which in turn improves the performance and endurance at the cost of losing some addressable space. Thus, system designers can determine and choose which factor is the main design goal of the system, based in part on the analytical model we developed (Equation 4.1) and in part on the evaluation results of the performance impact of the SSD GC efficiency. For systems that aim to achieve the peak performance of the SSD, for which cost is no object, e.g., military applications, reserving a fraction of the full LPN range can achieve the performance goal. On the contrary, when space efficiency is more important, e.g., backup and archiving systems, system designers can reserve most of the LPN range of the SSD. The performance degradation in the latter case can be alleviated by other design options, e.g., monitoring the workload temporal locality as discussed above.

Achieving stable and predictable response time of random reads: The GC and GC efficiency have a significant impact on the response time of random reads, which can make the random-read performance highly unpredictable. This conclusion, drawn from our experimental study, is very different from both the common belief and the specifications of commercial SSD products about SSD's random-read performance. For both application developers and SSD vendors, simply relying on the SSD to improve the random-read performance is not reliable because of the GC effect. To solve this problem, the vendors of SSD can incorporate control mechanisms in SSDs to postpone the GC until as many read requests are serviced as possible when the GC efficiency is low. Further, for both SSD vendors and system developers, increasing

the size of on-board RAM of the read cache inside the SSD or RAM of read cache of the host machine can alleviate the read-request pressure when low-efficiency GC is in progress.

Finding an optimal combination of the queue length and the request size: Achieving the maximal bandwidth and obtaining the minimal response time are conflicting goals. This calls for a tradeoff between the two. We have demonstrated that the request size is the main factor determining the achievable bandwidth and how fast the response time increases. Increasing the request size beyond 64KB will only increase response time without improving bandwidth. Therefore, for systems that aim to achieve the peak bandwidth, e.g., backup servers, it is beneficial to coalesce requests to 64KB groups before they are sent to the SSD. On the other hand, if the minimal response time is the main design goal of the system, sending as many small requests as possible is recommended because the response time remains almost unchanged while small requests are issued concurrently.

4.4 Summary

In this chapter, we conduct intensive experiments on a number of commercial SSD products from high-end PCI-E SSD to low-end SATA SSD. We expose performance anomalies of SSDs in enterprise-level application environment by customizing workloads in several different ways. In particular, we have made several insightful observations and provided useful suggestions for system developers and designers to take full advantage of SSDs' superior features while avoiding their limitations in the enterprise-level application environment. We have shown that, while GC efficiency has a significant impact on the performance of SSD, it is possible to avoid this problem with a judicious reservation of the LPN range at the cost of losing some addressable

space. Further, we have found that mapping policies of the FTL in SSD affect the dynamics of SSD. Thus system developers and designers can make use of this knowledge to decide how to best deploy SSDs with different mapping policies. We develop an analytical model based on our experimental findings in this study to estimate the residual lifetime of an SSD as a function of the LPN reservation range, the chip type and capacity of the SSD to help system developers and designers to anticipate and plan for the possible failure of SSD. We have demonstrated that random reads, which are commonly believed to be fast for SSDs, suffer noticeably from the problem of low GC efficiency and their performance can be highly unpredictable. We also find, experimentally, the optimal combination of queue length and request size that maximizes the bandwidth while minimizing response time for an SSD.

Chapter 5

SSD-Friendly IO Scheduling

As more and more SSDs have been deployed either as cache devices between the main memory and the hard drive [10, 19, 32, 47], or as the main storage device at the same level of HDD [27, 28] in the storage, the unique characteristics of SSD has exposed some problems of the traditional IO schedulers that fail to consider.

Conventional IO schedulers are largely designed to mitigate the high seek and rotational costs of mechanical disks by elevator-style IO request ordering and anticipation. In other words, the IO scheduler's target is to increase the sequentiality as much as possible to achieve the peak bandwidth. Further more, SSD has rich parallelism which traditional HDD does not have [8]. The current IO schedulers in the operating systems ignore the rich parallelism of SSD. Therefore, with more and more SSDs being deployed, the conventional IO schedulers in the current operating systems can no longer treat the device as a black box because the significant difference between SSD and HDD.

In this chapter, based on the parameters we find out in Chapter 4 and the fact that SSD exhibits strong parallelism, we design and implement an SSD-friendly IO scheduling scheme. This scheduling scheme considers queue length, read/write mix

and request size. These three factors are not considered in the conventional schedulers. The evaluation results show that this scheduling scheme is effective and improves both the bandwidth and average response time.

5.1 Background

5.1.1 IO schedulers and its ineffectiveness to SSD

Although block device drivers are able to transfer a single sector at a time, the block IO layer does not perform an individual IO operation for each sector to be accessed on disk because it will lead to frequent slow disk seek. Instead, the OS kernel can cluster several sectors and handle them as a whole, thus reducing the average number of head movements.

When a kernel component wishes to read or write some disk data, it actually creates a block device request. That request essentially describes the requested sectors and the kind of operation to be performed on them. The IO scheduler saves this IO request, which will be performed at a later time. This artificial delay is the crucial mechanism for boosting the performance of block devices. When a new block data transfer is requested, the kernel checks whether it can be satisfied by slightly enlarging a previous request that is still waiting. Because disks tend to be accessed sequentially, this simple mechanism is very effective.

In the Linux kernel, this IO scheduling is implemented by a single, robust, general purpose IO elevator. Moreover, this general purpose IO elevator can support multiple scheduling algorithms. Linux kernel has four scheduling algorithms, namely, *Noop*, *CFQ*, *Deadline* and *Anticipatory*. One of the pointers in the general purpose elevator points to the implementation one of the four scheduling algorithms. The user of the

operating system can configure the scheduling algorithms when it is started.

***Noop* elevator**

This is the simplest IO scheduling algorithm. There is no ordered queue: new requests are always added either at the front or at the tail of the dispatch queue depending on whether it is possible to merge at the front or at the back. The next request to be processed is always the first request in the queue.

***CFQ* elevator**

The Complete Fairness Queuing (CFQ) elevator fairly allocation of the disk IO bandwidth among all the processes that trigger the IO requests. To achieve this result, CFQ makes use of a large number of sorted queues which store the requests coming from the different processes. When a request is handed to the elevator, the kernel invokes a hash function that insert the request into the index of a queue; then, the elevator inserts the new request at the tail of this queue. Therefore, requests coming from the same process are always inserted in the same queue. By maintaining a different queue for each process, CFQ can achieve the fairness.

***Deadline* elevator**

The Deadline elevator makes use of four queues. Two of them are sorted queues including the read and write requests, respectively. These requests are ordered according to their initial sector numbers. The other two queues include the same read and write requests sorted according to their deadlines. A request deadline is essentially an expire timer that starts ticking when the request is passed to the elevator. The deadline ensures that the scheduler looks at a request if it's been waiting a long time, even if it is low in the sort.

The elevator checks the deadline queue of either read or write. If the deadline of the first request in the queue is passed, the elevator moves that request to the tail of the dispatch queue, ready to be served.

If no request is expired, the elevator dispatches a batch of requests starting with the request following the last one taken from the sorted queue.

***Anticipatory* elevator**

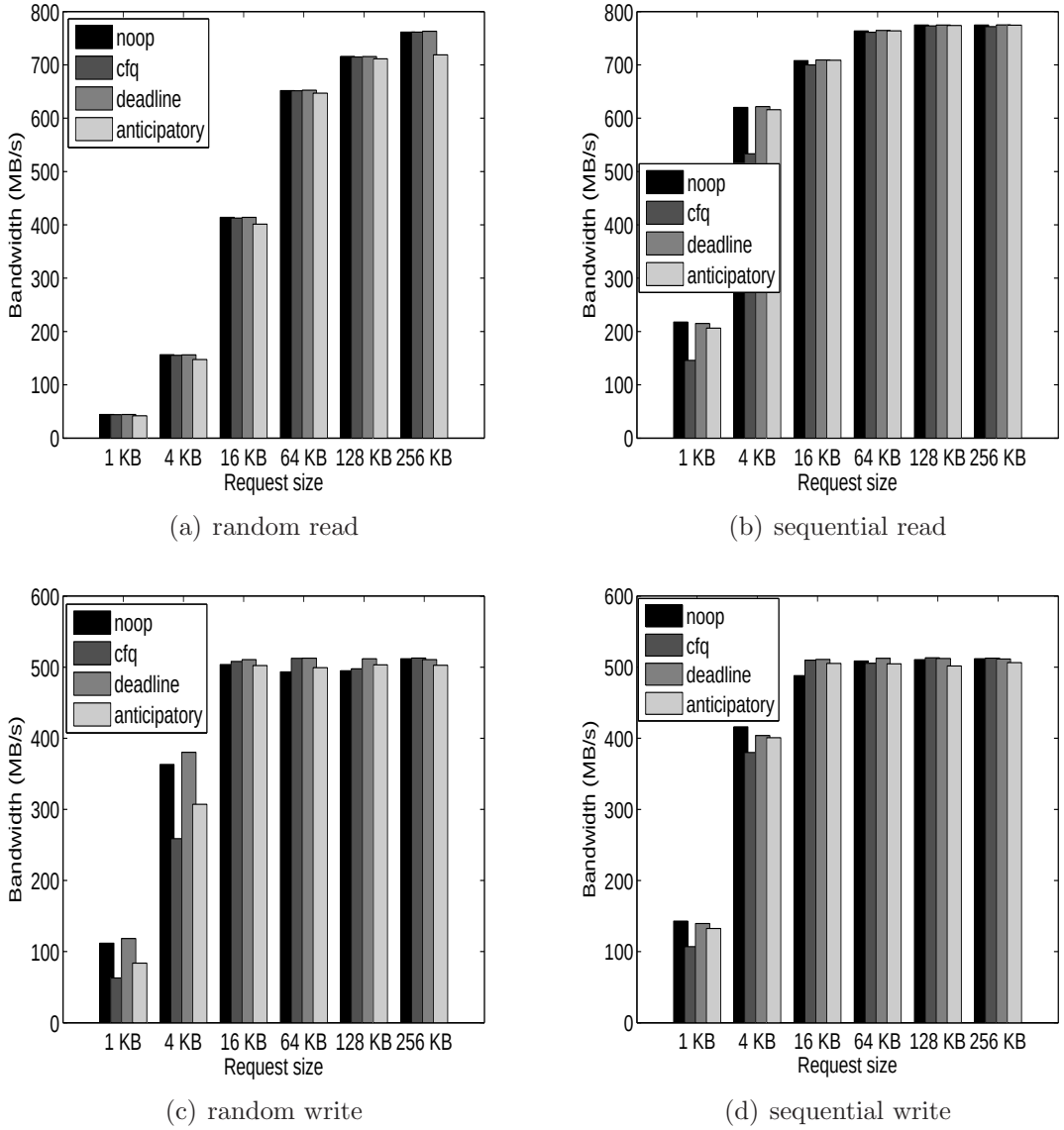
The Anticipatory elevator is similar to the Deadline elevator. There are two deadline queues and two sorted queues; the IO scheduler keeps scanning the sorted queues, alternating between read and write requests, but giving preference to the read ones. The scanning is basically sequential, unless a request expires.

Moreover, the elevator chooses a request behind the current position in the sorted queue, thus forcing a backward seek of the disk head. This happens, typically, when the seek distance for the request behind is less than half the seek distance of the request after the current position in the sorted queue. The scheduler will stall for a short period of time to wait for the next read request, which is the meaning of "anticipation".

From the discussion of the current schedulers in Linux system, we can see that these schedulers are designed for hard disk. Except for Noop scheduler which does nothing, the other three schedulers sort the requests based on their logical address to reduce seek. Moreover, some scheduler also waits for a while to anticipate the next read request. However, if the storage device is SSD, these schemes designed for increasing sequentiality and avoiding seek become useless. Moreover, it will also increase the storage stack overhead [52].

5.1.2 Ineffectiveness of conventional IO schedulers to SSD

In order to show the effect of the four IO schedulers to the performance of SSD, we evaluate the bandwidth of Fusion IO ioDrive under random read, sequential read, random write and sequential write workloads with varied request sizes, as shown in



(a) random read

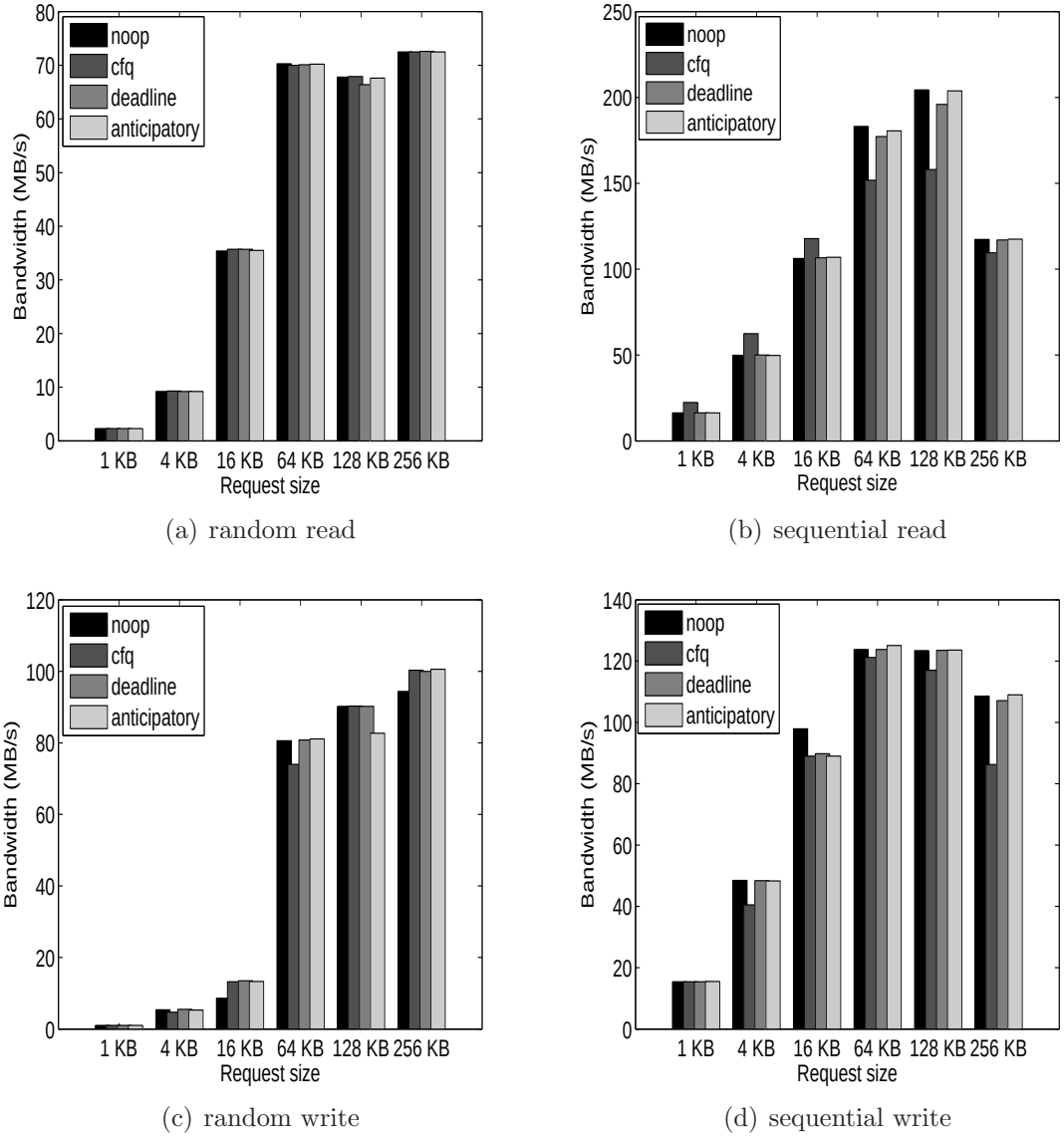
(b) sequential read

(c) random write

(d) sequential write

Figure 5.1: Bandwidth as a function of different schedulers and workloads of Fusion IO ioDrive

Figure 5.1 and Figure 5.2. We can see for different schedulers, the bandwidth is almost the same under different workloads with different request sizes for both SSDs. On the other hand, the request size is the deterministic factor of the performance. Therefore, we can conclude that the conventional IO schedulers in Linux system do not consider the SSD characteristics at all and they are not effective on SSD.



(a) random read

(b) sequential read

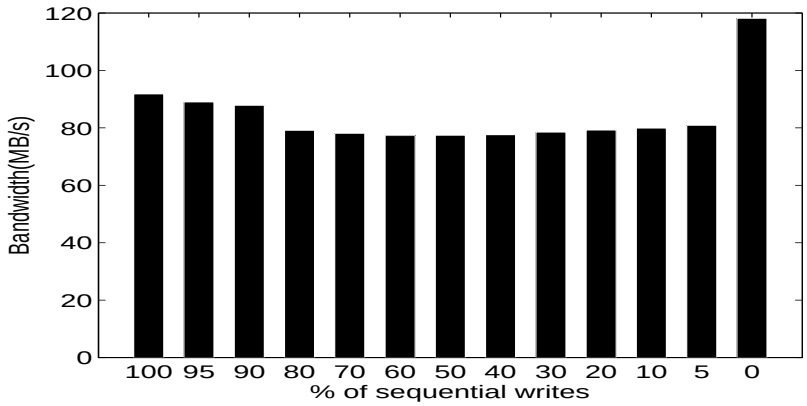
(c) random write

(d) sequential write

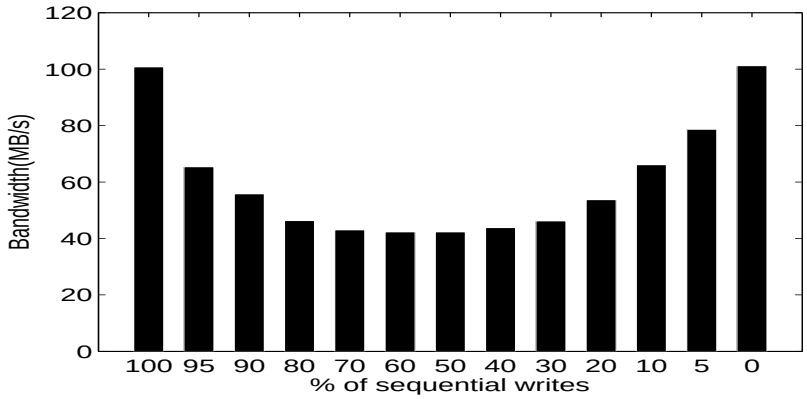
Figure 5.2: Bandwidth as a function of different schedulers and workloads of Sandisk SSD

5.1.3 Impact of mixing read and write requests

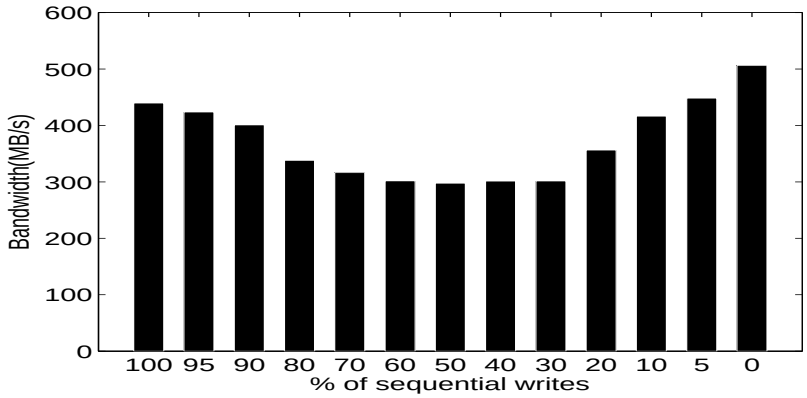
Previous studies have shown that mixing read and write requests will significantly increase the average response time [7]. Our own research based on the SSDs listed in Table 4.1. Also show that mixing read and write request will reduce the bandwidth



(a) Samsung with 4 KB request size and 64 outstanding IO



(b) Intel 320 with 4 KB request size and 64 outstanding IO



(c) Fusion IO ioDrive with 4 KB request size and 64 outstanding IO

Figure 5.3: Impact of mixing read and write requests – bandwidth as a function of the percentage of sequential write requests

significantly. Figure 5.3(a) to Figure 5.3(c) show the bandwidth as a function of the percentage of 4KB sequential write requests. We gradually inject 4KB random read requests into the workloads to find the impact to the bandwidth.

We can see for all the three SSDs, the highest bandwidth is when either the write requests and read requests accounts for 100% of the requests. On the other hand, when the sequential writes are mixed with 50% read requests, the bandwidth drops to the minimal value. For Intel 320 SSD, the bandwidth almost drops as high as 60%. Therefore, in order to achieve a better bandwidth, it is reasonable, if possible, to avoid mixing read and write requests as much as possible.

5.1.4 Key factors determining the performance of SSD

As shown in Figure 4.10, Figure 4.11 and Figure 4.12 in the previous section, the maximal bandwidth that can be achieved is determined by the request size, not the queue length. For all different request sizes, the maximal bandwidth determined by the request size is quickly reached after the queue length grows to merely 8 or more. This indicates that the request size is the main factor determining the maximal bandwidth. Moreover, for both random reads and random writes, the peak bandwidth can be reached when the request size is more than 64KB. This runs counter to the common belief that the request size should be increased as much as possible.

Therefore, based on results shown in Figure 4.10, Figure 4.11 and Figure 4.12, a request size of 64KB and queue length of 8 seem to be an optimal combination for achieving the peak bandwidth without significantly increasing the average response time. Further increasing the queue length and request size will only increase the average response time without any further benefit to the bandwidth.

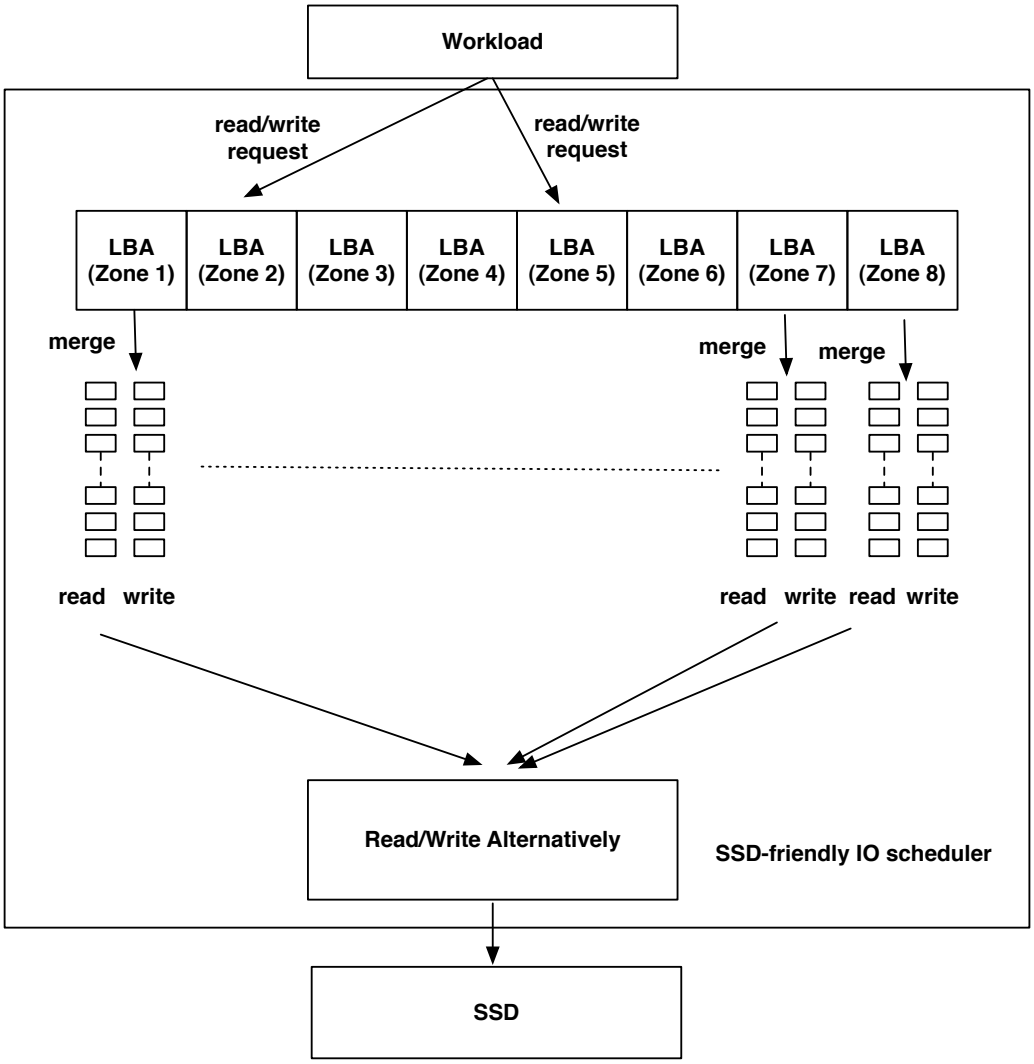


Figure 5.4: Overall architecture of SSD-friendly IO scheduler

5.2 Design and implement of SSD-friendly IO scheduler

As we discussed in the Introduction section, the studies of SSD are categorized into white-box approaches and black-box approaches. The design and implement of SSD-friendly IO scheduler falls into the black-box approach category. The design of SSD-

friendly IO scheduler is based on the findings of Chapter 4 and previous studies.

5.2.1 Dividing (logical block address) LBA into zones to maximize parallelism

Figure 5.4 shows the overall architecture of the SSD-friendly IO scheduler. Based on the size of the SSD, the scheduler divides the logical block address space into 8 zones, each of which represents a consecutive logical block address (LBA) sequence. That is, if the SSD has N logical block address, then each zone has $N/8$ LBA. And zone 1 stores LBA 0-$N/8-1$, zone 2 stores $N/8$-$N/4-1$, etc. The reason behind this design is that, as we found out in Chapter 4, queue length of 8 is an optimal value for the SSD to achieve the maximal bandwidth without reducing the response time. In other words, SSD can process 8 requests at the same time because these requests access different area of the chips of the SSD. Logically, these zones are controlled in different chips and work independently to generate very high performance. Therefore, we logically divide the SSD into 8 zones based on the LBA range to maximize the performance by exploiting the rich parallelism of the SSD.

For some workloads, the LBA may cluster in a limited range. So it is possible that some zones have many requests to be served while other zones are empty. This kind of workload will make the dividing of LBA useless. However, if the SSD is a dedicated device for a given application, which is aware of the maximal LBA of the requests it may generate, we can divide the zone evenly based on the maximal LBA the application will generate. Therefore, each of the zone will have an evenly distributed number of requests waiting to be serviced. In the evaluation, we assume the work sets of the workloads are known.

5.2.2 Splitting read and write requests

We have already shown that mixing read and write requests degrades the performance significantly. Therefore, for each zone, we create a read request queue and a write request queue in FIFO order. The scheduler then chooses read requests from all the read queues or write requests from all the write queues of each zone to send to the SSD. The scheduler alternatively selects read or write requests to synchronizing all the zones. By designing this way, we can make sure that the SSD does not process read and write requests at the same time.

Further, the scheduler can adjust the frequency to serve read or write requests. Intuitively, when the workload is read intensive, the scheduler should select more read requests from the queue than write requests. On the other hand, when the workload is write intensive, the scheduler should select more write requests from the queue than read requests. Therefore, the overall bandwidth can be improved.

5.2.3 Merging to increase the request size

We have shown in Chapter 4 that the request size determines the maximal bandwidth an SSD can achieve. Therefore, for each zone, after it accepts a request, the zone first checks the read queue or write queue based on whether the request is read or write. Then, the scheduler finds a request that can either back merge or front merge with the incoming request. In order to locate the request that can merge with the incoming request fast, we maintain a hash table for each of the queues with the key set as the ending address of each request. Therefore, we can use the start address of the incoming request to search the hash table, if it is a hit, it means the scheduler can back merge with the hit request. Otherwise, the scheduler insert the incoming request into both the FIFO request queue and hash table using its ending address for

future possible back merge. It is noted that the front merge is very rare, therefore, we ignore the front merge and set the key of the hash table as the ending address of each request.

As discussed in Chapter 4, the key factor that determines the maximal bandwidth an SSD can achieve is the average request size. Based on the fact, the scheduler merges the requests of each zone as much as possible. However, for some workloads, the merging of requests is very rare because the workloads are random. The request size of these workloads are small and not sequential. Therefore, for these kinds of workloads, the scheduler is not effective, as will be shown in the evaluation section.

5.2.4 Flowchart of SSD-friendly IO scheduler

Figure 5.5 is the flowchart of the scheduler. It first decides whether to accept a request or to send a request to SSD. If the scheduler accepts a request, it first determines which zone the request should send to based on the logical block address. After that, it searches the hash table by the start logical block address of the incoming request. If it hits, meaning the incoming request can be merged with an existing request in the queue of the given zone, the scheduler extends the key of hash table by adding it with the length of the incoming request. Otherwise, if it misses, meaning the incoming request can not be merged with any request in the queue of the given zone, the scheduler insert the incoming request in both the FIFO queue and the hash table. Meanwhile, the scheduler fetches all read requests or all write requests from the FIFO queues of all the zones to split the read and write requests and removes the corresponding items in the hash table. After that, it sends the requests to the SSD.

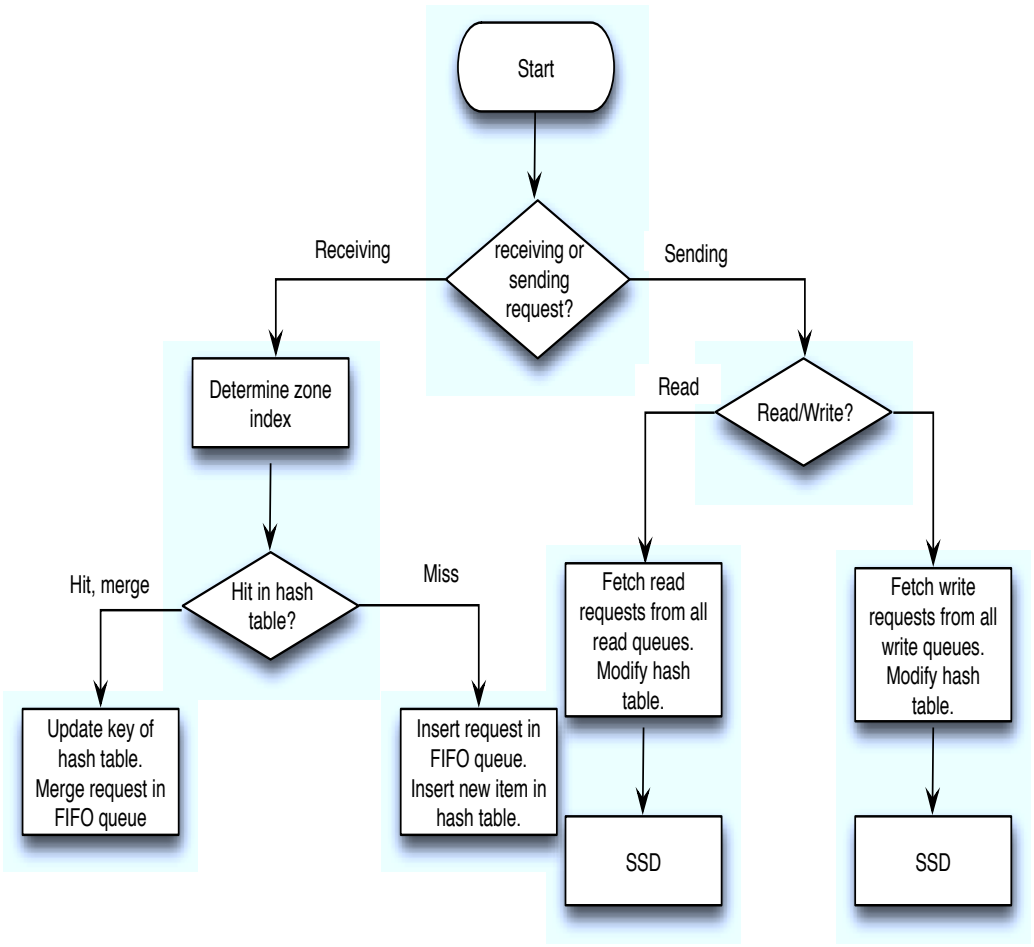


Figure 5.5: Flowchart of the SSD-friendly IO scheduler

5.3 Performance evaluation

5.3.1 Experiment setup

In our trace-driven simulation study of the SSD-friendly scheduler, we use the Intel 320 SSD whose characteristics are summarized in Table 4.1. We implement the scheduler in user space. It essentially intercepts the requests of the workload, inserts into the read or write queue of each zone, modifies hash table and sends the read or write requests synchronously among all the zones. We generate bandwidth and average

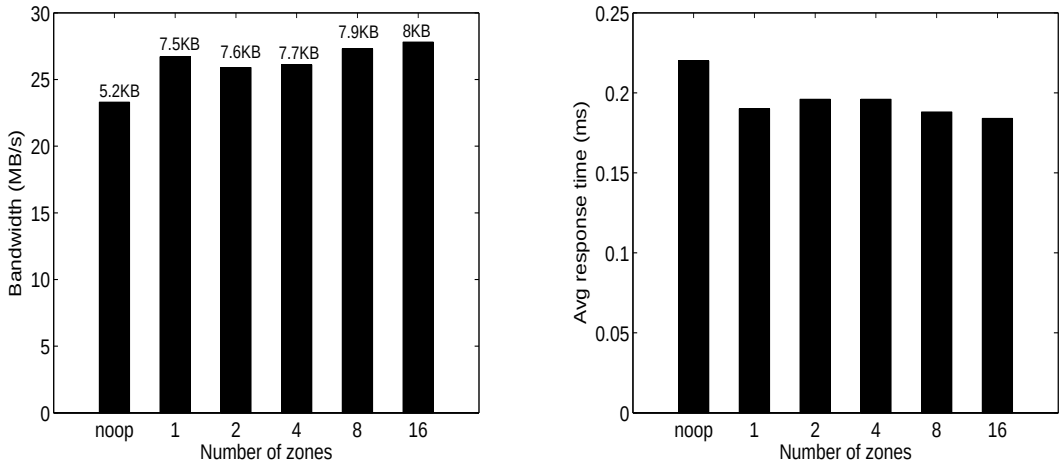
response time as output to evaluate the effectiveness of the scheduler. For all the evaluations, the IO scheduler of the Linux kernel is set to Noop to bypass any effect of the conventional schedulers. Moreover, in order to evaluate the sensitivity of zones to the performance, we evaluate our scheduler under different number of zones. We then compare our scheduler with Noop scheduler of Linux by comparing the bandwidth and average response time.

We fed five workloads to the SSD-friendly IO scheduler. The five traces are Financial 1 [39], Financial 2 [39], Microsoft Exchange Server [36], Microsoft MSN File Server [38], and Microsoft Builder server [36], whose key IO characteristics are summarized in Table 2.2.

5.3.2 Result analysis

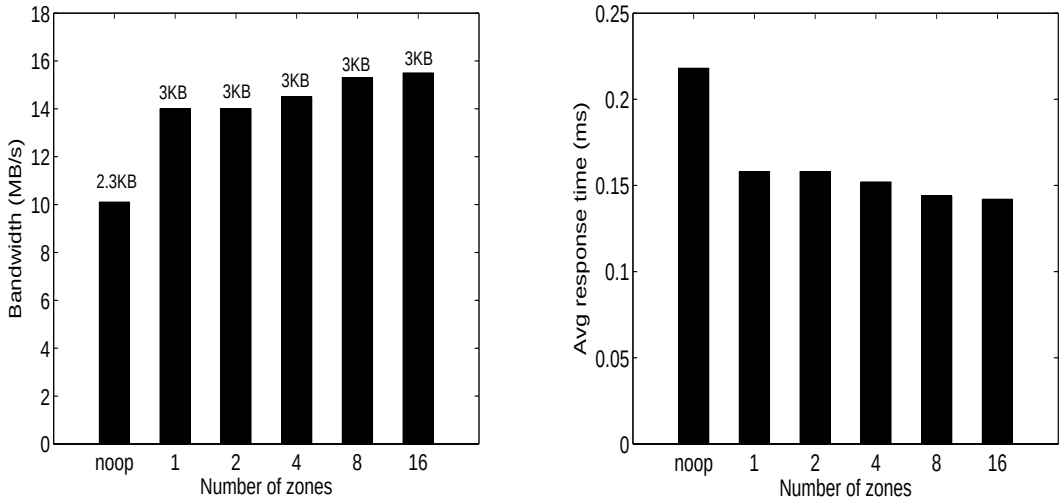
Figure 5.6 to Figure 5.10 show the evaluation results of our scheduler. The numbers on top of the bars are the average request sizes. The leftmost bar is the performance of the Noop scheduler coming with Linux. The right five bars are the performance of our scheduler configured with different number of zones.

The results can be categorized into two classes. Figure 5.6 and Figure 5.7 show the results of Financial 1 and Financial 2 workloads, respectively. These two workloads are sensitive to our scheduler. We can see for Financial 1 workload, the bandwidth increases up to 20% after we apply our scheduler. Moreover, the average response time decreases up to 20% when our scheduler is working. The reason lies in the fact that the request size determines the performance of SSD. The more the scheduler can increase the request size by merging, the more potentials of the SSD can be exploited. We can see the request size is increased by up to 54% when the number of zones is 16, as shown in Figure 5.6(a). Therefore, we can see the increase of bandwidth



(a) Bandwidth (b) Average response time

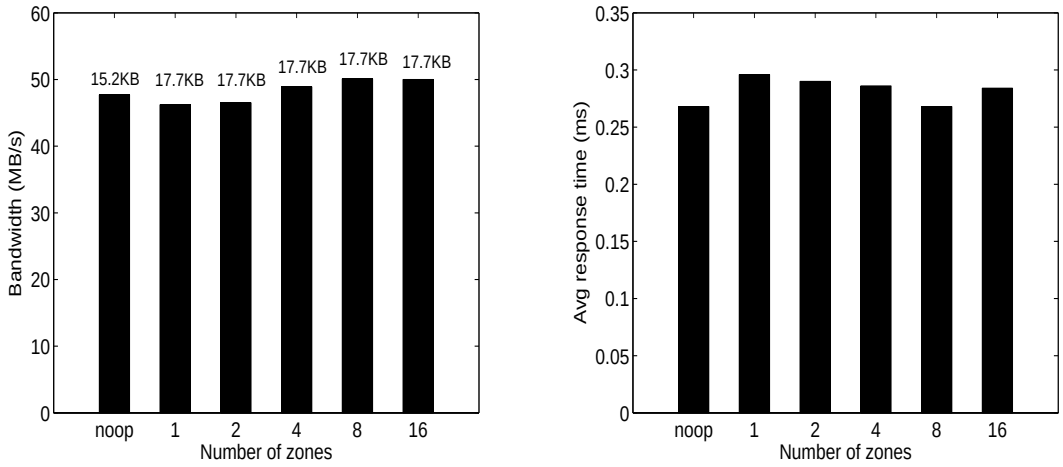
Figure 5.6: Bandwidth and average response time as a function of number of zones and schedulers for Financial 1 workload



(a) Bandwidth (b) Average response time

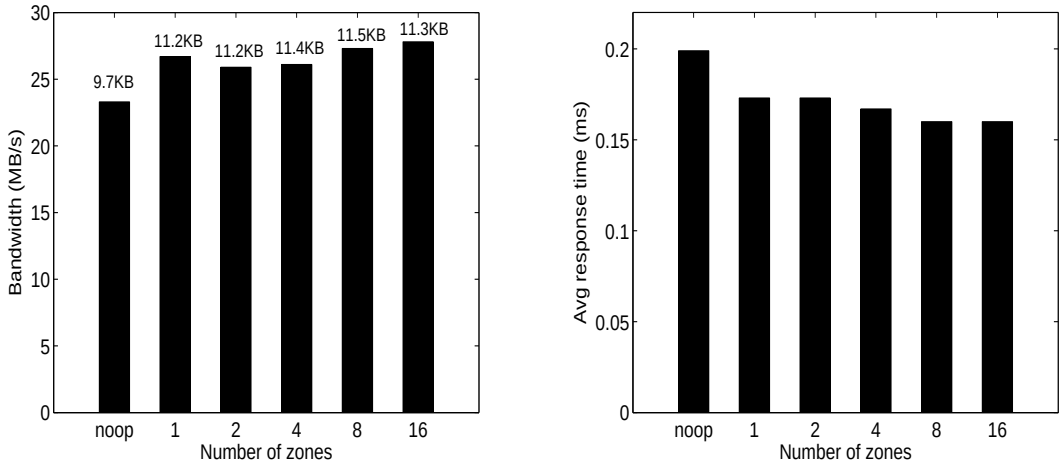
Figure 5.7: Bandwidth and average response time as a function of number of zones and schedulers for Financial 2 workload

and the decrease of average response time. Similarly, Financial 2 workload exhibits an even more improvement. The bandwidth is increased by up to 50%, shown in Figure 5.7(a) and the average response time is decreased by up to 30%, shown in



(a) Bandwidth (b) Average response time

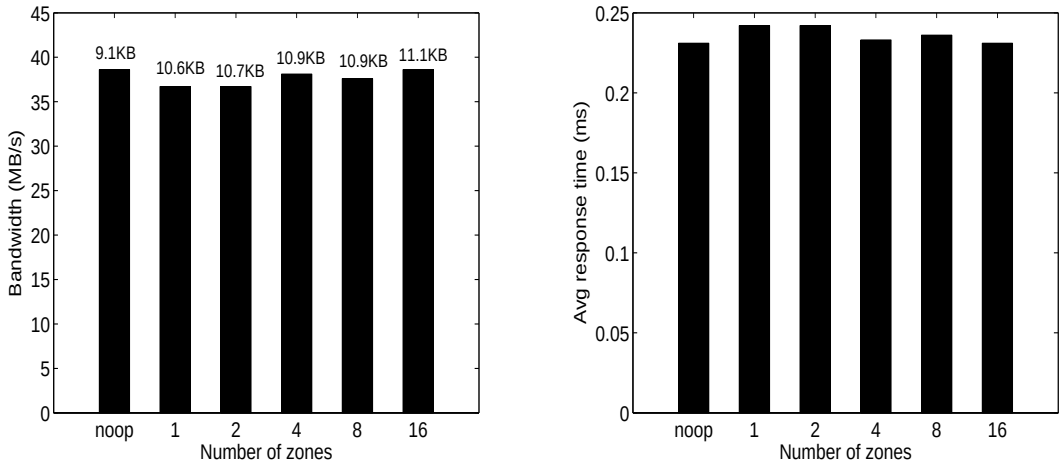
Figure 5.8: Bandwidth and average response time as a function of number of zones and schedulers for Exchange workload



(a) Bandwidth (b) Average response time

Figure 5.9: Bandwidth and average response time as a function of number of zones and schedulers for MSN workload

Figure 5.7(b). The reason is the same as for the Financial 1 workload. We can see the request size is increased by 30%. For both Financial 1 and Financial 2 workloads, the number of zones does not affect the performance very much. As the number of zones increases, the main trend of the bandwidth increases and the average response



(a) Bandwidth

(b) Average response time

Figure 5.10: Bandwidth and average response time as a function of number of zones and schedulers for Build workload

time decreases. This is because the FTL will map the LPN to any PPN in the SSD. Therefore, although we divide the SSD into logical zones, the FTL may have its own strategy to maximize parallelism.

Based on the evaluation of Financial 1 and Financial 2, we can also see that for workloads with smaller request size but much bigger request size after being merged, our scheduler can improve both the bandwidth and average response time significantly.

The other category of results include Exchange, MSN and Build workload. Different from Financial 1 and Financial 2, these workloads have much bigger request size, which is not increased by our scheduler. Therefore, we do not see improvement for these workloads, as shown in Figure 5.8, Figure 5.9 and Figure 5.10. These workloads are sequential workloads with big request sizes. Further, the requests do not arrive consecutively to allow our scheduler to merge.

The evaluation results prove our own research in Chapter 4 that the request size is the most important factor to determine the performance. Because the FTL of

SSD can map an LPN to any PPN in the SSD, the dividing design of our scheduler may not be effective because it treats the SSD as a black box. The SSD has its own internal queue to parallel the requests. However, the dividing approach still artificially randomizes the workloads, which helps FTL parallel requests to achieve a higher performance later on.

5.4 Summary

In this chapter, we proposed an SSD-friendly IO scheduler. This scheduler is motivated by both our own studies and previous research. First, we reveal the ineffectiveness of conventional IO schedulers in Linux system by running benchmark on three commercial SSDs. We compared Noop, Deadline, FAQ and Anticipatory and show that performance is not sensitive to different schedulers, which proves the ineffectiveness of conventional schedulers. Second, we show the fact that mixing read and write request significantly reduces the performance by running benchmark on three commercial SSDs, thus highlight the importance of splitting read and write requests. Third, based on the evaluation in Chapter 4, we conclude that the key factory that determines the performance of SSD is request size. Therefore, in our design, we want to merge the request as much as possible. The trace-driven experiment on the Intel 320 SSD shows that the SSD-friendly scheduler is effective for workloads with small but sequential requests. We also show that randomizing requests in user space, although the improvement is not significant, does helps the SSD to further parallel processing requests.

Chapter 6

Related Work

6.1 NVRAM

In the course of storage technology evolution, several non-volatile memory technologies have been proposed. EEPROM flash, battery powered RAM, NOR and NAND flash are four representative solid-state storage technologies. EEPROM [40] is a popular non-volatile byte-programmable memory. Because of its low write speed, it is typically used to store small amount of critical data in embedded devices. Battery-powered SRAM or DRAM, which is sometimes called NVRAM [6], has the same speed as SRAM or DRAM. Due to its battery-backed nature, data in NVRAM will not be lost when there is a power failure. However, battery-powered SRAM is very expensive while battery-powered DRAM is very energy inefficient. Besides, none of them can offer the same storage capacity of modern-day HDDs. While NOR flash [46] is being used as random-access byte-programmable ROM because it can be programmed in a random access manner, NAND flash [46] is mainly used in mass storage devices because of its higher density, larger capacity and lower cost than NOR. Due to its capacity advantage, NAND flash is considered the best candidate for the building block

of SSDs to replace the conventional HDDs. However, as discussed earlier, the *erase-before-write* problem of flash memory SSD significantly degrades the performance and decreases the life-time of SSD, particularly for write-intensive workloads. So how to alleviate this problem by minimizing the number of erasures is a key to improving the performance and longevity of SSD. To this end, Flash Translation Layer (FTL) design and flash-aware write-buffer management have been two general approaches to improving SSD's performance and longevity.

6.2 Study categorization

NAND flash SSD has been extensively studied in recent years. The research topics fall into several general categories, including (1) FTL design and on-board RAM management, (2) deployment of SSD in the storage hierarchy, and (3) SSD endurance, coding and security and (4) the internal parallelism of SSD. Many studies in (1) focus on the FTL design to ease the *erase-before-write* problem [9, 13, 14, 25, 29, 30, 33, 48], by either employing multiple mapping granularities or exploiting the content similarities in the write requests [9,14]. Other popular studies involve the management of the on-board RAM inside SSD [16, 21, 23, 24, 34, 43]. The research on FTL and the management of the on-board RAM inside SSD both treat the SSD as a white box to improve the performance of SSD from the inside, which helps the SSD vendors improve the design of SSD.

On the other hand, studies in category (2) treat the SSD as a black box and take advantage of SSD's superior features to those of HDD [22, 27, 28, 31]. These studies either deploy the SSD as the main storage device at the same level of HDD [27, 28] in the storage hierarchy, or deploy the SSD as a cache device above the main storage [22, 31]. These studies discuss how to effectively and efficiently use the SSD instead of the

design of the SSD. Several emerging enterprise-level products deploy SSD as a cache device and manage it by software [10, 19, 32, 47].

The third category of studies focuses on the endurance, coding and security issue of SSDs. Gokiul Soundararajan et al. [22] use HDD as a cache to increase the sequentiality of workloads and reduce the amount of data being written to the SSD. Simona Boboila et al. [5] use a reverse engineering technique to determine whether the chip or the algorithm is the key factor to determine the endurance of SSD. Wu et al. [9,14,49] use the content similarity in the workload to reduce the amount of data written to the SSD. The same content does not have to be written. Therefore, the lifetime of SSD is improved. Wu et al. [50] presents a cross-layer co-design approach to reduce SSD read response latency by changing the coding strength. The key is to cohesively exploit the NAND ash memory device write speed vs. raw storage reliability trade-off at the physical layer and run-time data access workload variation at the system level. It enables an opportunistic use of weaker error correction schemes that can directly reduce SSD read access latency.

The fourth category of studies exploit the internal structures of SSDs [2, 7, 8]. Agrawal Nitin et al. [2] expose the various internal structures and operations that affect the performance of SSD and give possible solutions to users. Feng Chen et al. [7, 8] reveal some unanticipated aspects in the performance dynamics of SSD. They also reveal that high-speed data processing benefit from the rich parallelism of the SSD.

Chapter 7

Conclusion and Future Work

7.1 PUD-LRU write buffer management algorithm

In this dissertation we first demonstrated that typical server and online transaction processing workloads exhibit strong temporal locality based on the observation that a large percentage of the requested addresses are updated repeatedly in the relevant traces. We further show that buffer management plays an important role in improving the performance of SSDs and cannot be replaced by sophisticated FTLs based on our experimental study. To fully exploit the temporal locality and increase the destaging efficiency, we propose a new erase-efficient write-buffer management algorithm for SSDs, called PUD-LRU, that groups blocks in the buffer into two groups based on a combined measure of frequency and recency, Predicted average Update Distance (PUD). PUD-LRU chooses a block to destage from the group that is considered less frequently and less recently updated, such that the chosen block has the most valid pages in it. By doing so our scheme maximizes the efficiency of each destaging

operation. We evaluated the effectiveness of our PUD-LRU scheme through an extensive trace-driven simulation study that compares PUD-LRU with the state-of-the-art buffer management scheme BPLRU, the state-of-the-art page-mapping DFTL and the pure page-mapping FTL scheme in terms of the number of erasures and average response time. The results show that PUD-LRU significantly and consistently outperforms BPLRU in both measures. Further, PUD-LRU significantly outperforms DFTL except for the Financial 2 trace. We also found that for workloads that are highly sequential or with low temporal locality, pure page-mapping FTL, though most flexible, underperforms the log-block FTL. This is because the former spends a significant amount of time collecting blocks that contain very few invalid pages and copying valid pages to other available blocks, while the latter initiates frequent switch merges that are optimal based on SSD characteristics. We further compared the destaging efficiency between PUD-LRU and BPLRU, which shows that PUD-LRU has a higher destaging efficiency than BPLRU.

In our future study of PUD-LRU, we plan to design and implement a dynamic and adaptive tuning mechanism for obtaining the optimal threshold value that facilitates the grouping of blocks in the write buffer.

7.2 GC-ARM on-board RAM management algorithm

In this dissertation, we propose GC-ARM, a garbage-collection aware RAM management scheme for SSDs. GC-ARM is motivated by a number of important observations. First, we observe the significant impact of the low GC efficiency problem to SSDs by evaluating two commercial SSDs. On the other hand, the existing write buffer man-

agement schemes are by and large oblivious of GC efficiency, thus failing to help improve performance for workloads that tend to induce low GC efficiency. GC-ARM can detect this situation using a *benefit value* to decide whether to destage a page or a block at a time. Second, based on the observed high write-back traffic resulting from address translation, GC-ARM groups the LPN-to-PPN mapping entries based on their LPN. Finally, the observation of the randomness of workloads varying over time and from workload to workload motivated the GC-ARM design that dynamically adjusts the size ratio between the write buffer and the cached mapping table. Extensive trace-driven evaluation results show that GC-ARM consistently outperforms BPLRU, DFTL and AP in terms of the number of erasures, average response time, GC efficiency and write traffic reduction.

In our future study of GC-ARM, we plan to design and implement a adaptive scheme to detect the randomness of workloads. In other words, we do not use the fixed 500 instinctive blocks method. We also plan to design GC-ARM on top of other FTLs such as log-block FTL and block-mapped FTL. In other words, we want to make GC-ARM FTL orthogonal.

7.3 Identifying performance anomalies of SSD in enterprise environment

In this dissertation, we conduct intensive experiments on a number of commercial SSD products from high-end PCI-E SSD to low-end SATA SSD. We expose performance anomalies of SSDs in enterprise-level application environment by customizing workloads in several different ways. In particular, we have made several insightful observations and provided useful suggestions for system developers and designers to

take full advantage of SSDs' superior features while avoiding their limitations in the enterprise-level application environment. We have shown that, while GC efficiency has a significant impact on the performance of SSD, it is possible to avoid this problem with a judicious reservation of the LPN range at the cost of losing some addressable space. Further, we have found that mapping policies of the FTL in SSD affect the dynamics of SSD. Thus system developers and designers can make use of this knowledge to decide how to best deploy SSDs with different mapping policies. We develop an analytical model based on our experimental findings in this study to estimate the residual lifetime of an SSD as a function of the LPN reservation range, the chip type and capacity of the SSD to help system developers and designers to anticipate and plan for the possible failure of SSD. We have demonstrated that random reads, which are commonly believed to be fast for SSDs, suffer noticeably from the problem of low GC efficiency and their performance can be highly unpredictable. We also find, experimentally, the optimal combination of queue length and request size that maximizes the bandwidth while minimizing response time for an SSD.

In our future study, we want to evaluate the analytical model of residual lifetime to see how accurate this model will be. We want to compare our analytical model with tools coming with SSDs and other state-of-the-art studies. Further more, we also would like to design possible solutions based on our findings. For example, how to find a best LPN reservation scheme based on a given workload to both reduce cost and improve GC efficiency? Moreover, we would like to implement a tool that can run directly on the SSD without intervening from the user to identify key parameters such as page size, block size, FTL scheme, GC efficiency with respect to reserved LPN range, etc. This tool will be general and it can run on SSD with any brand.

7.4 SSD-friendly IO scheduler

In this dissertation, we design and implement an SSD-friendly IO scheduler in user space based on the findings we made by evaluating a number of current commercial products and previous study. First, the request size is the determining factor that determines the maximal bandwidth an SSD can achieve. Therefore, we try to merge the requests as much as possible to achieve a better performance. Second, we show that SSD can achieve the maximal bandwidth under a fixed request size when the number of requests sent simultaneously is 8. Increasing the queue length further does not improve the bandwidth but hurt the average response time. Therefore, we divide the SSD into 8 zones based on LBA. This design is based on the assumption that these logically divided zones work independently because of SSD's physical characteristics. Next, based on previous studies and our own evaluations of several commercial SSDs, we conclude that mixing read and write requests degrades the performance of SSD significantly. Thus, for each of the zones, we create a read request queue and a write request queue. When sending requests to the SSD, the scheduler selects either read requests or write requests from all the zones. By doing this, we can make sure that the SSD will not process read and write requests at the same time. The simulation results show that the SSD-friendly IO scheduler is effective. It increases the average request size and improves both the bandwidth and average response time.

In our future study, we plan to implement this design into the Linux kernel and compare it with the conventional schedulers.

Bibliography

[1] SandForce SF-100 datasheet. `http://www.sandforce.com`.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC '08)*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[3] Amir Ban. Flash File System. *US Patent 5,404,485*, April 1995.

[4] Amir Ban. Flash File System Optimized For Page-mode Flash Technologies. *US Patent 5,937,425*, August 1999.

[5] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proceedings of the 8th USENIX conference on File And Storage Technologies (FAST '10)*, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[6] Peter Chan. Datasheet: X4C105 NOVRAM features and applications, 2005.

[7] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the 11th International Joint conference on Measurement and*

Modeling of Computer Systems (SIGMETRICS '09), pages 181–192, New York, NY, USA, 2009. ACM.

[8] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceeding of 17th international conference on High-Performance Computer Architecture (HPCA '11)*, pages 266–277, Washington, DC, USA, 2011. IEEE Computer Society.

[9] Feng Chen, Tian Luo, and xiaodong Zhang. CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX conference on File And Stroage Technologies (FAST '11)*, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[10] FlashSoft, 2012. http://www.flashsoft.com.

[11] Eran Gal and Sivan Toledo. Algorithms and Data Structures for Flash Memories. *ACM Comput. Surv.*, 37(2):138–163, June 2005.

[12] M.E. Gomez and V. Santonja. Characterizing temporal locality in I/O workload. In *SPECTS*, 2002.

[13] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 229–240, New York, NY, USA, 2009. ACM.

[14] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing nand flash-based ssds. In *Proceed-*

ings of the 9th USENIX conference on File and Stroage Technologies (FAST '11), pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[15] Jian Hu, Hong Jiang, and Prakash Manden. Understanding performance anomalies of ssds and their impact in enterprise application environment. In *Proceedings of the 14th International Joint conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, New York, NY, USA, 2012. ACM.

[16] Jian Hu, Hong Jiang, Lei Tian, and Lei Xu. PUD-LRU: An Erase-Efficient Write Buffer Management Algorithm for Flash Memory SSD. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '10)*, pages 69–78, Washington, DC, USA, 2010. IEEE Computer Society.

[17] Jian Hu, Hong Jiang, Lei Tian, and Lei Xu. GC-ARM: Garbage Collection-Aware RAM Management for Flash based Solid State Drives. In *Proceedings of the 2012 IEEE International Conference on NetWorking, Architecture, and Storage*, Washington, DC, USA, 2012. IEEE Computer Society.

[18] Yingbo Hu and David Moore. MLC vs. SLC NAND Flash in Embedded Systems, 2009. http://www.smxrtos.com/articles/mlcslc.htm.

[19] IOTurbine, 2012. http://www.fusionio.com/systems/ioturbine.

[20] Song Jiang, Xiaoning Ding, and Feng Chen. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality. In *FAST*, 2005.

[21] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. FAB: Flash-Aware Buffer Management Policy for Portable Media Players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, May 2006.

[22] Taeho Kgil, David Roberts, and Trevor Mudge. Improving NAND Flash Based Disk Caches. In *Proceedings of the 35th annual International Symposium on Computer Architecture (ISCA '08)*, pages 327–338, Washington, DC, USA, 2008. IEEE Computer Society.

[23] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, pages 16:1–16:14, Berkeley, CA, USA, 2008. USENIX Association.

[24] Hyojun Kim, Jin-Hyuk Kim, ShinHo Choi, HyunRyong Jung, and JaeGyu Jung. A page padding method for fragmented flash storage. In *Proceedings of the 2007 International Conference on Computational Science and its Applications (ICCSA '07)*, pages 164–177, Berlin, Heidelberg, 2007. Springer-Verlag.

[25] Jesung Kim, Jong Min Kim, Sam H.Noh, Sang Lyul Min, and Yookun Cho. A Space-efficient Flash Translation Layer for Compactflash Systems. *IEEE Transaction on Consumer Electronics*, 48(2):366–375, May 2002.

[26] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST '11)*, pages 1–12, Washington, DC, USA, 2011. IEEE Computer Society.

[27] Ioannis Koltsidas and Stratis D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1):514–525, August 2008.

[28] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*, pages 1075–1086, New York, NY, USA, 2008. ACM.

[29] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[30] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, October 2008.

[31] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: Analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer Systems (EuroSys '09)*, EuroSys '09, pages 145–158, New York, NY, USA, 2009. ACM.

[32] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*, pages 25–25, Berkeley, CA, USA, 2012. USENIX Association.

[33] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A Reconfigurable FTL (Flash Translation Layer) Architecture for

NAND Flash-based Applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):38:1–38:23, August 2008.

[34] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: A Replacement Algorithm for Flash Memory. In *Proceedings of the 2006 International conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, pages 234–241, New York, NY, USA, 2006. ACM.

[35] David Patterson, Garth Gibson, and Randy Katz. A caes for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.

[36] SNIA: IOTTA Repository. Exchange Server Traces.

[37] SNIA: IOTTA Repository. Microsoft Enterprise Traces.

[38] SNIA: IOTTA Repository. MSN Storage File Server.

[39] UMass Trace Repository. OLTP Application I/O.

[40] George Rostky. Remembering the PROM knights of Intel, 2002.

[41] Samsung K9F8G08UXM datasheet. 2007.

[42] Making Multi-level Cell Flash Practical for Enterprise Solid State Drives, 2009. http://www.wwpi.com.

[43] Hyotaek Shim, Bon-Keun Seo, Jin-Soo Kim, and Seungryoul Maeng. An adaptive partitioning scheme for dram-based cache in solid state drives. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST '10)*, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[44] Solid state storage performance test specification, 2011. http://www.snia.org.

[45] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD Lifetimes with Disk-based Write Caches. In *Proceedings of the 8th USENIX conference on File And Storage Technologies (FAST '10)*, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[46] Arie Tal. NAND vs. NOR flash technology.

[47] VeloBit, 2012. http://www.velobit.com.

[48] Qingsong Wei, Bozhao Gong, Suraj Pathak, Bharadwaj Veeravalli, LingFang Zeng, and Kanzo Okada. WAFTL: A Workload Ddaptive Flash Translation Layer with Data Partition. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST '11)*, pages 1–12, Washington, DC, USA, 2011. IEEE Computer Society.

[49] Guanying Wu and Xubin He. Delta-ftl: improving ssd lifetime via exploiting content locality. In *Proceedings of the 7th ACM European conference on Computer Systems (EuroSys '12)*, pages 253–266, New York, NY, USA, 2012. ACM.

[50] Guanying Wu, Xubin He, Ningde Xie, and Tong Zhang. Diffecc: Improving ssd read performance using differentiated error correction coding schemes. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '10)*, pages 57–66, Washington, DC, USA, 2010. IEEE Computer Society.

[51] Michael Wu and Willy Zwaenepoel. eNVy: a non-volatile, main memory storage system. In *Proceedings of the 6th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '94)*, pages 86–97, New York, NY, USA, 1994. ACM.

[52] Xiaojian Wu and A. L. Narasimha Reddy. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[53] Aayush Gupta Youngjae Kim, Brendan Tauras and Bhuvan Urgaonkar. FlashSim: a simulator for NAND flash-based solid-state-drives. In *SIMUL*, 2009.