Hot and Cold Data Identification: Applications to Storage Devices and Systems

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Dongchul Park

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Professor David H.C. Du

July, 2012
Acknowledgements

There are many people that have earned my gratitude for their contribution to my Ph.D. at the University of Minnesota. First of all, I am grateful to my advisor Professor David H.C. Du from the bottom of my heart for his invaluable advice. He taught me how to research and encouraged me to become a better researcher. I also give my thanks to my Ph.D. committee members for their good suggestion and help for my dissertation: Professor David Lilja, Jon Weissman, and Abhishek Chandra.

I would like to express my appreciation to my family. Especially, I would like to give my sincere thanks to my father (Yonghwan Park) in the heaven and my mother (Yeonok Jeong). They have always lived a honest and industrious life throughout their life, and instilled the value of the sweat of their brow in my mind. I deeply thank my wife, Seonna Kim, for her thoughtful support and sincere love. I also thank my daughter, Yewon, for her bright smile. Without their endless love, I could not achieve my Ph.D. degree. I also would like to thank my elder brother (Dongchan), sister-in-law (Soonae Choi), elder sisters (Sunghee, Myunghhee, Younghee) and brothers-in-law (Changmo Lee, Jaehyung Kim, Jeongchul Kim) for their support and encouragement.

I would like to give my sincere appreciation to my father-in-law (Yongsup Kim) and mother-in-law (Soonae Seo) for their thoughtful love and support. I also thank my sister-in-law (Jungna Kim), her husband (Anmo Kim), brother-in-law (Sungjoong Kim) and his fiancee (Sunyang Kim). I would especially like to thank my aunt (Taesook Park), uncle (Youngrak Jeong), and cousins (Sungwha, Donghwa) for their endless support.

Lastly, many thanks to Korean colleagues (Jinoh Kim, Myunghwan Park, Jaehoon Jeong, Hunjeong Kang, and Taehyun Hwang) in Computer Science and DISC (DTC
Intelligent Storage Consortium) research group members for their help and advice. In particular, I would like to give my special gratitude to the visiting professor (Youngjin Nam) for his valuable advice and collaboration.
Dedication

I dedicate this dissertation to my family, especially...

To my parents, Yonghwan Park and Yeonok Jeong, who made all of this possible, for their endless encouragement and patience.

To my beloved wife, Seonna Kim and my sweetheart, Emily Yewon Park for their love and support.
Abstract

Hot data identification is an issue of paramount importance in storage systems since it has a great impact on their overall performance as well as retains a big potential to be applicable to many other fields. However, it has been least investigated. In this dissertation, I propose two novel hot data identification schemes: (1) multiple bloom filter-based scheme and (2) sampling-based scheme. Then I apply them to the storage device and system such as Solid State Drives (SSD) and data deduplication system.

In the multiple bloom filter-based hot data identification scheme, I adopt multiple bloom filters and hash functions to efficiently capture finer-grained recency as well as frequency information by assigning a different recency coverage to each bloom filter. The sampling-based scheme employs a sampling mechanism so that it early discards some of the cold items to reduce runtime overheads and a waste of memory spaces. Both hot data identification schemes empower each scheme to precisely and efficiently identify hot data in storage with less system resources.

Based on these approaches, I choose two storage fields as their applications: NAND flash-based SSD design and data deduplication system. Particularly in SSD design, hot data identification has a critical impact on its performance (due to a garbage collection) as well as its life span (due to a wear leveling). To address these issues in SSD design, I propose a new hybrid Flash Translation Layer (FTL) design that is a core part of the SSD design. The proposed FTL (named CFTL) is adaptive to data access patterns with the help of the multiple bloom filter-based hot data identification algorithm.

As the other application, I explore a data deduplication storage system. Data deduplication (for short, dedupe) is a special data compression technique that has been widely adopted especially in backup storage systems for backup time saving as well as storage saving. Unlike the traditional dedupe research that has focused more on the write performance improvement, I address its read performance aspect. In this section, I newly design a read cache in dedupe storage for a backup application to improve
read performance by looking ahead their future references in a moving window with the combination of a hot data identification algorithm.

This dissertation addresses the importance of hot data identification in storage areas and shows how it can be effectively applied to them in order to overcome the existing limitations in each storage venue.
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>System Parameters and Values</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Workload Characteristics</td>
<td>27</td>
</tr>
<tr>
<td>3.3</td>
<td>System Parameters</td>
<td>48</td>
</tr>
<tr>
<td>3.4</td>
<td>Workload Characteristics</td>
<td>48</td>
</tr>
<tr>
<td>4.1</td>
<td>Various FTL Schemes</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>Simulation Configuration</td>
<td>85</td>
</tr>
<tr>
<td>4.3</td>
<td>Workload Characteristics</td>
<td>85</td>
</tr>
<tr>
<td>4.4</td>
<td>Various dedupe gain ratio (DGR) of each backup dataset (Unit:%)</td>
<td>106</td>
</tr>
<tr>
<td>4.5</td>
<td>Characteristics of the six backup datasets</td>
<td>111</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Typical System Architecture of Flash Memory-based Storage Systems</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Typical Data Deduplication Process</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Bloom Filter Operations</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Two-level LRU List Scheme</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>Multiple Hash Function Framework. Here, D = 4 and B = 2</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>Our Framework and Its Operations</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Our Aging Mechanism</td>
<td>19</td>
</tr>
<tr>
<td>3.6</td>
<td>Recency Coverage for Each Bloom Filter</td>
<td>19</td>
</tr>
<tr>
<td>3.7</td>
<td>The Number of Unique LBAs within Unit (5,117 Writes) Periods</td>
<td>21</td>
</tr>
<tr>
<td>3.8</td>
<td>Working Process of WDAC Algorithm (Here, the window size is 10)</td>
<td>25</td>
</tr>
<tr>
<td>3.9</td>
<td>Hot Ratios of Two Baseline Algorithms</td>
<td>28</td>
</tr>
<tr>
<td>3.10</td>
<td>The Number of Different Identification between Two Baseline Algorithms</td>
<td>29</td>
</tr>
<tr>
<td>3.11</td>
<td>Hot Ratios of Four Schemes under Various Traces</td>
<td>31</td>
</tr>
<tr>
<td>3.12</td>
<td>False Identification Rates of Both MBF and MHF</td>
<td>32</td>
</tr>
<tr>
<td>3.13</td>
<td>Average Runtime Overhead per Operations</td>
<td>34</td>
</tr>
<tr>
<td>3.14</td>
<td>False Identification Rates between Two Schemes with a Same Memory Space</td>
<td>35</td>
</tr>
<tr>
<td>3.15</td>
<td>Average Hot Ratios over Various Window Sizes and Decay Periods</td>
<td>36</td>
</tr>
<tr>
<td>3.16</td>
<td>Performance Change over Various Numbers of a Bloom Filter in Our Scheme</td>
<td>36</td>
</tr>
<tr>
<td>3.17</td>
<td>HotDataTrap Framework</td>
<td>41</td>
</tr>
<tr>
<td>3.18</td>
<td>False LBA Identification Rates for Each Workload</td>
<td>43</td>
</tr>
<tr>
<td>3.19</td>
<td>Proportion of LBA Access for Each Trace</td>
<td>43</td>
</tr>
<tr>
<td>3.20</td>
<td>HotDataTrap Victim Selection</td>
<td>44</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Over the decade, researchers have explored diverse alternative storage technologies to replace conventional magnetic disk drives. With the recent technological breakthroughs, a NAND flash memory has emerged to become one of the most promising storage technologies for the next generation storage devices. The Flash memory, these days, has become very popular in many mobile devices due to its fast access speed, low power consumption, high shock resistance, and portability. In particular, flash-based Solid State Drives (SSDs) have attracted considerable attention even in enterprise servers as well as in personal computers as a main storage alternative for the conventional hard disk drives (HDDs) [1, 2, 3, 4, 5, 6, 7].

Admittedly, flash memory is an emerging storage technology to replace current magnetic storage devices. However, its distinguished features such as an erase-before-write and limited cell lifetime cause several critical limitations: out-of-place update, slow write performance, reliability issue, etc. To get over these drawbacks, a new software/hardware layer, called Flash Translation Layer (FTL), has been developed and deployed between conventional file systems and raw flash memory (shown in Figure 2.1) [6, 8, 9, 10, 11, 12, 13, 14]. This FTL is a core of the flash-based storage devices like SSDs because the overall performance of the SSDs is overwhelmed by the FTL design. The FTL, basically, plays a role in address translation between Logical Block Address (LBA) and its Physical Block Address (PBA) so that users can utilize the
flash memory with existing conventional file systems without significant modification. A typical FTL consists largely of an address allocator, a cleaner (i.e., garbage collector and wear leveler), and hot data identifier. The address allocator deals with an address translation and the cleaner tries to collect blocks filled with invalid data pages to reclaim them for its near future reuse. Both components are heavily dependent on the hot data identification.

The advent of SSD had a significant effect on the existing storage system hierarchies. By adding the SSDs into the conventional storage hierarchy between the main memory (i.e., DRAM) and magnetic hard drives (HDDs), the SSDs can serve as a second-level cache of the HDDs. Unlike the DRAM, the flash memory in SSDs is non-volatile. Therefore, we can dramatically improve the overall performance of the entire storage systems thereby storing the frequently accessed data (i.e., hot data) into the SSDs and by placing the less frequently accessed data (i.e., cold data) on the HDDs.

In addition to these applications in flash memory, hot data identification has a big potential to be utilized by many other fields such as shingled write disks [15, 16, 17, 18, 19, 20], data deduplication [21, 22, 23, 24], sensor networks [5], etc. Although it has a great impact on designing storage systems, it has been least investigated. The challenges are not only the classification accuracy, but also a limited resource (i.e., SRAM) and computational overheads.

In this dissertation, I propose novel hot and cold data identification schemes and show how they can be exploited by various storage devices and systems. The rest of this document is organized as follows; chapter 2 first describes background knowledge on our target storage device (SSD) and storage system (data deduplication system), and then explores various research issues for each topics. In chapter 3, two novel hot data identification schemes such as a multiple bloom filter-based design and sampling-based design are proposed. In chapter 4, I make an attempt to apply the proposed hot data identification schemes to both an SSD design and data deduplication system, and show how much their performances are improved. Lastly, chapter 5 concludes this dissertation and discusses the future work of the hot data identification for storage applications.
Chapter 2

Background and Research Issues

This chapter explores various research issues in a storage device such as flash memory (including SSD) as well as in storage data deduplication system.

2.1 Flash Memory and Solid State Drives (SSD)

Flash memory space is partitioned into blocks, where each block contains a fixed number (32 or 64) of pages. Unlike magnetic disk drives, flash memory requires no mechanical movement and therefore exhibits as fast of random read performance as sequential read. However, a major drawback is that in-place updating (i.e., overwriting) is not allowed. That is, an overwriting to pages in a flash memory must be preceded by an erasure of the corresponding block. To promote efficiency, a clean (i.e., already erased) block is often used and all the valid (i.e., not outdated) pages within the block need to be copied into the clean block. Then, the original data block is marked as invalid and will be erased by a garbage collector for reclamation in the future. This in-place update problem results from its asymmetric operational granularity that both reads and writes are executed by pages, while erase operations are performed only by blocks. According to [25], read operations take 15 microseconds, write operations take 200 microseconds, and erase operations take 2000 microseconds. This is the main reason erase operations severely degrade the overall performance of flash memory. Therefore, reducing the number of
erasure operations is one of the most critical issues in flash memory. Moreover, balancing the erase count of each cell block, called wear leveling, is also another crucial issue due to the limited life span of a cell block in flash memory [25]. To resolve these problems, the flash translation layer (FTL) has been designed and deployed.

The FTL is a software/firmware layer implemented inside a flash-based storage device to make the linear flash memory device act like a magnetic disk drive (Figure 2.1) [6 8 9 10 11 12 13 14]. FTL emulates disk-like in-place updates for a Logical Page Number (LPN) based data page. It writes the new data page to a different location with a new Physical Page Number (PPN) and FTL maintains the mapping between the LPN and its current PPN. Finally, it marks the old PPN as invalid for reclamation later. In this way, FTL enables existing application to use flash memory without any modification. Internally though, FTL still needs to deal with the physical characteristics of the flash memory. Thus, an efficient FTL scheme has a critical effect on overall performance of flash memory since it directly affects in-place update performance and
balancing the wearing of each data block (i.e., wear leveling problem).

Figure 2.1 describes a typical system architecture of flash memory-based storage systems [26]. Both Memory Technology Device (MTD) and Flash Translation Layer (FTL) are two major parts of flash memory architecture. The MTD provides primitive flash operations such as read, write, and erase. The FTL plays a role in address translation between Logical Block Address (LBA) and its Physical Block Address (PBA) so that users can utilize the flash memory with existing conventional file systems without significant modification. A typical FTL is largely composed of an address allocator and a cleaner. The address allocator deals with address translation and the cleaner tries to collect blocks filled with invalid data pages to reclaim them for its near future use. This garbage collection is performed on a block basis; so all valid pages in the victim block must be copied to other clean spaces before the victim block is to be erased.

Another crucial issue in flash memory is wear leveling. The motivation of wear leveling is to prevent any cold data from staying at any block for a long period of time. Its main goal is to minimize the variance among erase count values for each block so that the life span of flash memory is to be maximized. Currently, the most common allowable number of write per block is typically 10K for MLC and 100K for SLC.

### 2.2 Data Deduplication Systems

Data deduplication (for short, dedupe) is a specialized data compression technique to eliminate coarse-grained redundant data [21]. It is widely adopted to save storage consumption by retaining only one unique instance of data on storage media and replacing redundant data with a pointer to the unique data afterwards. Especially, due to the exponential data growth most of the data centers are experiencing these days, this data de-duplication has received a lot of attention from storage vendors and IT administrators alike.

Figure 2.2 shows a typical dedupe process consisting mainly of chunking and dedupe logics. A typical dedupe process works as follows: it begins to divide an incoming data stream into smaller chunks of variable/fixed lengths and computes a hash value of
each chunk by using a cryptographic hash function (e.g., SHA-1). Generally, dynamic chunking (i.e., variable-length chunks) with fingerprinting [27] or its variants outperforms static chunking (fixed-length chunks) [28] for data backup applications, where an average chunk size ranges from 4KB to 8KB [28, 23, 22]. However, static chunking works well with some other applications such as VM disk images [29].

A hash index table, also known as a key-value store [30], is required to effectively maintain a large number of hash values with their storage locations. Only if the hash value of a chunk is not found in the hash index table (that is, if the chunk has never been appeared or stored before), this (unique) chunk is allowed to be written to the storage. Otherwise, the chunk (i.e., shared chunk) is ignored. The probability of the hash collision is relatively very low enough to accommodate incoming chunks compared to a soft error rate of the storage system [31].

The unique chunks are not directly written to the underlying storage because the chunk sizes are not large enough to achieve high write performance. Instead, they are temporarily organized in a fixed-sized large container (2 or 4MB) [22], where the container size is even bigger than an average chunk size. The unique chunks are initially buffered in the in-memory container that is allocated for each data stream. Once the in-memory container is full of the chunks, it is flushed to the underlying storage. To read a chunk from the storage, it is generally not allowed to read a small chunk in a container; instead the entire container must be read.

Recent dedupe research has focused mainly on maximizing the efficient duplication
detection by using better chunking [32, 33, 34], optimized hash indexing [23, 30], locality-preserving index caching [22], and various bloom filters [22, 35]. Moreover, good write performance has been its primary factor by compressing unique chunks and performing (fixed-length) large writes (2 or 4MB) through containers [36] or similar structures [37].
Chapter 3

Hot and Cold Data Identification

One of the critical issues in designing storage devices and integrating them into the storage hierarchy is how to effectively identify hot data that will be frequently accessed in near future as well as have been frequently accessed so far. In this section, I propose two novel hot data identification schemes: multiple bloom filter-based scheme and sampling-based scheme.

3.1 Background and Related Work

3.1.1 Bloom Filter

The main goal of the bloom filter (for short, BF) is to probabilistically test set membership with a space efficient data structure [38]. Since the space efficiency is an important factor for the BF, the correctness can be sacrificed in order to maximize it. Thus, although a given key is not in the set, a BF may provide a wrong positive answer called a false positive. However, the basic BFs never provide a false negative. We can also adjust design parameters (i.e., BF size ($M$), the number of hash function ($K$) and the number of unique element ($N$)) of a BF to allow a very low probability of the false positive.

The bloom filter is a bit array of $M$ bits and all bits are initially set to 0. In addition to this bit array, there must also be $K$ independent hash functions, each of which maps the given elements to the corresponding bit positions of the array thereby setting them
all to 1. Figure 3.1 gives a specific example of a bloom filter with 4 independent hash functions. To insert an element to the bloom filter, the key value of the element is fed to the 4 hash functions so that we can get 4 hash values which correspond to the bit array positions. Then all the 4 bit positions are set to 1. Similarly all elements can be recorded to the bloom filter. To execute a membership query (i.e., check if the element is in the set), we first need to get the 4 bit positions by feeding the key value of the element to all 4 hash functions. If any of the 4 bits are 0, this means the element is not in the set because all the bits would have been set to 1 when the element was fed. If all are 1, there exist two possible cases: the corresponding element is in the set, or it is just a false positive due to the insertions of other elements. Assuming the probability of a false positive is very low, the answer to the query is positive.

The merit of a BF is its low computational overhead since both insertions and membership test are executed in constant time. Moreover, we can achieve a good performance with high space efficiency, simple algorithm and efficient hash functions. However, it is difficult to store additional information with the keys in the BF [38].

3.1.2 Existing Hot Data Identification Schemes

In this subsection, we describe the existing hot and cold data classification schemes and explore their advantages and disadvantages.

Chiang et al. [39] proposed an out-of-update scheme called Flash Memory Server
(FMS) in order to reduce the number of erase operations in flash memory. They made an attempt to classify data into three types for an efficient garbage collection: read-only, hot and cold data. Although FMS exhibits a good hot and cold data classification performance, it requires a large amount of memory space since it must keep the last access time information of all LBAs.

To resolve this limitation, Chang et al. [40] used a two-level LRU list that consists of a hot LBA list and a candidate list. As shown in Figure 3.2, both lists are of fixed size and operate under LRU. Whenever a write request comes into the FTL driver, if the corresponding LBA already exists in the hot list, the data are classified as hot data. Otherwise, it is defined as cold data. If the LBA is not in the hot list but in the candidate list, the data are promoted to the hot list. If the data are not in either list, it will be inserted into the candidate list. This two-level LRU scheme consumes less memory than FMS; nevertheless, it incurs other problems. The performance of hot data identification is totally dependent on the sizes of both lists. In other words, a small hot list can save memory space, but its performance decreases because highly likely hot data may be demoted to the candidate list or even evicted from the candidate list. Moreover, this scheme requires high computing overheads to emulate LRU discipline.

Recently, Hsieh et al. [41] proposed a multiple hash function framework to identify hot data (Figure 3.3). This adopts multi-hash functions and one K-dimensional bloom
filter to capture not only the frequency of the data access by incrementing the corresponding counters, but also the recency of the hot data by dividing the counter value by two. If any one bit of H-most significant bits is set to 1, the data are classified as hot. For example, as shown in Figure 3.3, assuming this scheme adopts 4-bit counters and 2-most significant bits, 4 numbers of an LBA access count will be its hot threshold value. Thus, the access counter values are greater than or equal to 4, the data in the LBA are identified as hot. After a specified time, this scheme decreases all the counter values by a half at once with 1-bit right shifting manner to implement decay effect. Compared to the other schemes, this achieves relatively less memory consumption as well as less computing overheads. However, it does not appropriately catch recency information due to its exponential decrement of all LBA counters. Moreover, this multihash function framework does not cope well with its counter overflow problem. Since this scheme adopts a 4-bit counter, it cannot deal with the large numbers over 15. When the counter overflows occur, they suggested either freezing the corresponding counter
or decreasing all counter values exponentially. Neither a freezing nor an exponential decay is reasonable to resolve its counter overflow problem. Especially, when we adopt an exponential decay of all counters, this scheme has a tendency to exhibit considerably low hot ratios due to excessively frequent decay process.

3.2 Proposed Scheme 1: A Multiple Bloom Filter-based Hot Data Identification

Hot data identification can be applied to a variety of fields. Particularly in flash memory, it has a critical impact on its performance (due to a garbage collection) as well as its life span (due to a wear leveling). Although the hot data identification is an issue of paramount importance in flash memory, little investigation has been made. Moreover, all existing schemes focus almost exclusively on a frequency viewpoint. However, recency also must be considered equally with the frequency for effective hot data identification. In this paper, we propose a novel hot data identification scheme adopting multiple bloom filters to efficiently capture finer-grained recency as well as frequency. In addition to this scheme, we propose a Window-based Direct Address Counting (WDAC) algorithm to approximate an ideal hot data identification as our baseline. Unlike the existing baseline algorithm that cannot appropriately capture recency information due to its exponential batch decay, our WDAC algorithm, using a sliding window concept, can capture very fine-grained recency information. Our experimental evaluation with diverse realistic workloads including real SSD traces demonstrates that our multiple bloom filter-based scheme outperforms the state-of-the-art scheme. In particular, ours not only consumes 50% less memory and requires less computational overhead up to 58%, but also improves its performance up to 65%.

3.2.1 Introduction

A tendency toward flash-based Solid State Drives (SSD) now holds sway even in the enterprise servers as well as in personal computers, especially, laptops. This trend of
the storage world results from the recent technological breakthroughs in flash memory and dramatic reduction of its price [42].

Flash memory is organized in units of blocks and pages. Each block consists of a fixed number (32 or 64) of pages. Reads and writes in flash memory are performed on a page basis, while erases operate on a block basis. Its most distinguishing feature is that it does not allow in-place updates. That is, the flash memory system cannot overwrite new data into existing data. Instead, the new data are written to clean spaces somewhere (i.e., out-of-place updates), and then the old data are invalidated for reclamation in the future.

In order to resolve this in-place update issue, Flash Translation Layer (FTL) has been developed and deployed to flash memory to emulate in-place update like block devices [43, 44, 45, 46, 47, 11]. This layer enables users to utilize the flash memory like disks or main memory on top of existing conventional file systems without significant modification by hiding the characteristics of the out-of-place update. As time goes on, this out-of-place update inevitably causes the coexistence of numerous invalid (i.e., outdated) and valid data. In order to reclaim the spaces occupied by the invalid data, a new recycling policy is required, which is a so-called garbage collection. However, the garbage collection can give rise to not only considerable amount of valid data copies to other clean spaces, but also data erases to reclaim the invalidated data spaces. Data erase (1,500μs) is the most expensive operation in flash memory compared to data read (25μs) and write (200μs) [25]. Consequently, a garbage collection results in a significant performance overhead as well as unpredictable operational latency. Flash memory exhibits another limitation: cells can be erased for a limited number of times (e.g., 10K-100K). Thus, frequent erase operations reduce the lifetime of the flash memory, which causes a wear leveling issue. The objective of wear leveling is to improve flash lifetime by evenly distributing cell erases over the entire flash memory [25]. Both the wear leveling and garbage collection are fundamentally based on the hot data identification.

We can simply classify the frequently accessed data as hot data. Otherwise, they are regarded as cold data. This definition is still vague and takes only frequency (i.e.,
the number of appearance) into account. However, there is another important factor—recency (i.e., closeness to the present)—to identify hot data. In general, many access patterns in workloads exhibit high temporal localities \[48\]; therefore, recently accessed data are more likely to be accessed again in near future. This is the rationale for including the recency factor in hot data classification. Thus, if we redefine hot data while considering recency as well as frequency, it would be like this: if any recently accessed data will be frequently accessed again in near future, they will be regarded as hot data. Moreover, current hot data may turn into cold if they are no longer accessed frequently in the future and vice versa.

The definition of hot data can be different for each application and also can be applied to a variety of fields. First of all, this can be primarily used in data caching \[49, 4\]. By caching these hot data in the memory space in advance, we can significantly improve system performance. It is also applied to B-tree indexing in sensor networks \[5\]. FlashDB is a flash-based B-tree index structure optimized for sensor networks. In FlashDB, the B-tree node can be stored either in read-optimized mode or in write-optimized mode, whose decision can be easily made on the basis of a hot data identification algorithm. Flash memory adopts this classification algorithm particularly for a garbage collection and a wear leveling \[50, 51, 52\]. We can perform a garbage collection more efficiently by collecting and storing hot data to the same block, which can reduce the garbage collection overhead. Moreover, we also improve flash reliability by allocating hot data to the flash blocks with low erase count. Hybrid SSD is another good application of hot data identification \[53, 54\]. We can store hot data to SLC (Single-Level Cell) flash memory, while cold data can be stored to MLC (Multi-Level Cell) part in the SLC-MLC hybrid SSD. In addition to these, hot data identification has a big potential to be exploited by many other applications. In this paper, we focus on flash-based applications; thus, we consider only write accesses, not read accesses.

Flash memory contains a fixed small amount of memory (SRAM) inside; so we need to exploit the SRAM efficiently. Since FTL needs the majority of this memory for a more efficient address translation, we have to minimize this memory usage for the hot data identification. This is one of the challenges to design an efficient hot data identification
scheme. Moreover, computational overhead is another important issue since it has to be triggered whenever every write request is issued.

Although this hot data identification is an issue of paramount importance in flash memory, it has been least investigated. Existing schemes either suffer from large memory space requirements [39] or incur huge computational overhead [40]. To overcome these problems, Hsieh et al. recently proposed a multiple hash function framework [41] to identify hot data. This scheme adopts multiple hash functions and a counting bloom filter to capture a frequency. Although this approach accurately captures frequency information because it maintains counters, it cannot appropriately capture recency information due to its exponential batch decay process (i.e., to decreases all counter values by a half at a time). While investigating other hot data identification schemes, we also posed a question about the existing baseline algorithm (i.e., direct address method in [41]): it cannot properly capture recency either. The direct address method assumes that unlimited memory space is available to keep track of hot data. It maintains a counter for each LBA to store access counts and periodically decays all LBA counting information thereby dividing by two at once. However, this algorithm also still retains the same limitation as the multihash function scheme.

Considering these observations, an efficient hot data identification scheme has to meet the following requirements: 1) effective capture of recency information as well as frequency information, 2) small memory consumption, and 3) low computational overhead. Based on these requirements, in this paper, we propose a novel hot data identification scheme based on multiple bloom filters. The key idea of this scheme is that each bloom filter has a different weight and recency coverage so that it can capture finer-grained recency information.

Whenever a write request is issued, the hash values of the LBA are recorded into one of multiple bloom filters (for short, BFs) in a round robin fashion. All information of each BF will be periodically erased by turns in order to maintain fine-grained recency information, which corresponds to an aging mechanism. Thus, each BF retains a different recency coverage. We also dynamically assign a different recency weight to each BF: the BF that just erased (i.e., reset BF) has higher recency weight, while the lowest
recency weight is assigned to the BF that will be erased in right next turn because this BF has stored LBA access information for the longest period of time. For frequency, our proposed scheme does not maintain a specific counter for all LBAs; instead, the number of BF recording the LBA information can exhibit its frequency information. The main contributions of this paper are as follows:

- **An Efficient Hot Data Identification Scheme:** A bloom filter can provide computational and space efficiency. Both multihash scheme and our proposed scheme try to take advantage of the bloom filter. Unlike the former using one counting bloom filter, the latter adopts multiple bloom filters. The multiple bloom filters enable our proposed scheme to capture finer-grained recency information so that we can achieve more accurate hot data classification. Multiple and smaller bloom filters empower our scheme to require not only less memory space, but also lower computational overhead.

- **A More Reasonable Baseline Algorithm:** Our proposed approximation algorithm named Window-based Direct Address Counting (WDAC) adopts a window that is a conceptual buffer with a predefined size to store each coming request. Each LBA maintains a corresponding access counter. Whenever a write request is issued, the LBA is stored in the head of the window and the oldest one is evicted like a FIFO (First In First Out) queue. WDAC assigns different recency weights to all LBAs in the window according to the closeness to the present. Thus, when a new request arrives, all LBAs are shifted toward the tail of the window and all their recency values are reevaluated. Consequently, WDAC can catch precise (very fine-grained) recency as well as frequency.

### 3.2.2 Main Design

This section describes our proposed multiple bloom filter-based hot data identification scheme (Section 3.2.2.1) and our Window-based Direct Address Counting (WDAC) scheme as our baseline algorithm (Section 3.2.2.2).
3.2.2.1 The Framework

As shown in Figure 3.4, our scheme adopts a set of $V$ independent bloom filters (for short, BFs) and $K$ independent hash functions to capture both frequency and finer-grained recency. Each BF consists of $M$ bits to record $K$ hash values. The basic operation is simple: whenever a write request is issued to the Flash Translation Layer (FTL), the corresponding LBA is hashed by the $K$ hash functions. The output values of each hash function ranges from 1 to $M$, and each hash value corresponds to a bit position of the $M$-bit BF respectively. Thus, $K$ hash values set the corresponding $K$ bits in the first BF to 1. When the next write request comes in, our scheme chooses the next BF in a round robin fashion to record its hash values. In addition, it periodically selects one BF in a round robin manner and erases all information in that BF to reflect a decay effect.

- **Frequency:** Unlike the multihash function framework adopting 4-bit counters, we do not maintain the specific BF counters for each LBA to count the number of appearance. Instead, our scheme investigates multiple BFs to check if each BF has recorded the corresponding LBA. The number of BF retaining the LBAs can show its frequency.

  For precise frequency capturing, when it chooses one of the $V$ BFs and marks the corresponding bits to 1, if the selected BF has already recorded the hash values of the LBA, it sequentially (in a round robin manner) examines other BFs until it finds a new one that has not recorded the LBA. This sequential examination minimizes disruption of our recency analysis. If it finds such a new one, it records the hash values to the BF. If it turns out that all (or predefined number of) BFs have already contained the LBA information, our scheme simply defines the data as hot and skips its further processes such as a BF checking or a recency weight assignment since this will be definitely over the threshold. This shortcut decision reduces its overhead (this will be demonstrated in our experiment section). Consequently, assuming the hash values of an LBA appear in $r$ ($0 \leq r \leq V$) numbers of the BFs out of $V$ BFs, we can say the corresponding LBA has appeared $r$ times before. On the other hand, if all bloom filters have already recorded the LBA, we do not know its precise frequency number from then on. However, for hot
Whenever write requests are issued, one bloom filter is selected sequentially and the hash values (LBA) are recorded to it. Currently chosen bloom filter

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>h2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>h3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Figure 3.4: Our Framework and Its Operations

data identification in our scheme, the information required is simply the fact if the value \( r \) is larger than the threshold, not the precise counter value. If this \( r \) is larger than a threshold, it passes the frequency check.

- **Recency**: Even though the total access counters of two LBAs are equivalent in a period, the one accessed heavily in the further past and has never been accessed afterwards should be classified differently from the other one heavily accessed recently. If a hot data identifier depends only on counter values (frequency information), it cannot make distinction between them. Even though it has a decay mechanism, it cannot classify them well if the identifier captures coarse-grained recency.

Since our scheme does not maintain LBA counters, we need to devise a different aging mechanism to capture recency information. Figure 3.5 illustrates our aging mechanism to decay old information. In Figure 3.5, a white (i.e., not-shaded) bloom filter corresponds to a reset bloom filter. Consider that we adopt \( V \) independent BFs and the hash values of LBAs are recorded to each BF in a round-robin manner during a
predefined interval $T$. An interval $T$ represents a fixed number of consecutive write requests, not the time interval. As shown in Figure 3.5 (a), after the interval $T$, the BF that has not been selected in the longest time interval is selected and all bits in the BF (i.e., $BF_V$) are reset to 0. As soon as it is reset, the hashed LBA values start to be recorded again to all BFs including the reset BF. After the interval $T$, the next BF ($BF_1$) is selected and all the bits are reset (shown in Figure 3.5 (b)). Similarly, after the next $T$ interval, the next BF ($BF_2$) is chosen in a right cyclic shift manner and all information is erased (shown in Figure 3.5 (c)) as time goes on.

Figure 3.6 shows the recency coverage after the interval $T$ as soon as ($BF_V$) is reset. The reset BF ($BF_V$) can remember LBA information accessed during only the last one
interval $T$ (i.e., latest $1T$ interval). The previously reset BF ($BF_{V-1}$) can record the LBA information accessed during the last two intervals. Similarly, the BF 1 ($BF_1$) which will be chosen as a next reset BF after this period can cover the longest interval $V \times T$. This means $BF_1$ records all LBA information for the last $V \times T$ intervals.

Our proposed scheme assigns a different recency weight to each BF so that recency value is combined with frequency value for hot data decision. The reset BF ($BF_V$) records most recent access information; so highest recency weight has to be assigned to it, whereas lowest recency weight is allotted to the BF that will be chosen as a next reset BF ($BF_1$) because it has recorded the access information for the longest time. We intend to use recency value as a weight to the frequency value such that a final value combining both can be produced. Consequently, even though two different LBAs have appeared once in $BF_V$ and $BF_1$ respectively, both frequency values are regarded differently as follows: assuming $BF_V$ is a current reset BF, we first choose one BF ($BF_{\lfloor V/2 \rfloor}$) that has medium recency coverage and then assign a weight value of 1 to it. Next, we assign a double weight value (this is explained in subsection 3.2.2.2) to the reset BF ($BF_V$), which means the LBA appearance in this BF counts as a double frequency. Third, we calculate the difference of recency weights between each BF by $1/(V - \lfloor V/2 \rfloor)$ and assign evenly decreasing weight value from $BF_{V-1}$ to $BF_1$. For example, consider we adopt 4 BFs ($V = 4$) and $BF_4$ is a current reset BF, we first select $BF_2$ and assign a weight value of 1 to it. Then, we allot a double weight value (i.e., 2) to the reset BF ($BF_4$). Next, we can get 0.5 ($= 1/(4 - 2)$) as a difference of recency weights and finally distribute different recency weight values 2, 1.5, 1, and 0.5 to $BF_4$, $BF_3$, $BF_2$, and $BF_1$ respectively. Here, if any LBA appears both in $BF_3$ and $BF_4$, the original frequency value would be 2, but our scheme regards the final value as 3.5 because each BF has different recency weights. In this paper, we define this combination value of frequency and recency as a hot data index. Thus, if the value of hot data index is greater than or equal to a predefined threshold value, we regard the data as hot. Algorithm 1 describes a regular checkup operation in our multiple BF-based scheme.

\* **Algorithm:** Algorithm 1 describes a regular checkup operation in our multiple bloom filter-based scheme. Whenever write requests are issued, this operation checks whether
the data in the corresponding LBA are hot or not. The algorithm is simple. Consider a write request with LBA \( X \) is issued from a host. This LBA \( X \) is fed to the \( K \) numbers of hash functions to find appropriate bit-positions to be set in a bloom filter. Our scheme chooses one of \( V \) bloom filters (i.e., \( \text{CurrentBF} \)) in a round robin manner and checks if the LBA has been already recorded in the bloom filter before (i.e., all \( K \)-bit positions in \( \text{CurrentBF} \) have already set to 1). If the bloom filter has already recorded the LBA information, our scheme continuously makes an attempt to randomly choose another bloom filter without replacement manner until it finds one that has not recorded the LBA information. If it finds such a bloom filter available, it records the LBA information to the bloom filter by setting all \( K \)-bit positions in the bloom filter to 1. Otherwise (i.e., all bloom filters have already stored the LBA information), it defines the data in the LBA as hot data (i.e., shortcut decision) because the LBA has as much frequently accessed before as all bloom filters recorded the LBA.

If the \( \text{CurrentBF} \) does not contain the LBA information, the scheme sets all the \( K \)-bit positions in the selected bloom filter to 1. Then it starts to check which bloom filter contains the LBA information and calculates the total value of hot data index. Finally if the total value is greater than or equal to the \( \text{THRESHOLD} \), it defines the data as hot data, otherwise, cold data.

- **A Bloom Filter Size and Decay Period:** Applications of a BF are required to configure three parameters: a BF size \((M)\), the number of hash function \((K)\) and the
Algorithm 1 A Regular Checkup Operation

Function HotColdCheckup(BloomFilter)
1: \( V = \text{The Number of Bloom Filter} \)
2: \( K = \text{The Number of Hash Function} \)
3: \( \text{RECENCY}_\text{WEIGHT} = 1 / (V - \lfloor V/2 \rfloor) \)
4: Write Request is issued with LBA X
5: RequestCounter++
6: // LBA X is fed to each hash function
7: for \((i=1; i\leq K; i++)\) do
8: \( i_i = h_i(X) \)
9: end for
10: Choose one bloom filter, CurrentBF
11: if (All K-bit positions in CurrentBF are already set to 1) then
12: Try to choose another bloom filter sequentially
13: if (Found one available) then
14: for \((i=1; i\leq K; i++)\) do
15: \( \text{CurrentBF}[i] = 1 \)
16: end for
17: end if
18: // All bloom filters have already recorded the LBA
19: if (Cannot find any bloom filter available) then
20: \( \text{CurrentBF} = \text{The Next Bloom Filter} \)
21: // Shortcut decision
22: Define this as HOT data and return
23: end if
24: else
25: for \((i=1; i\leq K; i++)\) do
26: \( \text{CurrentBF}[i] = 1 \)
27: end for
28: end if
29: \( \text{CurrentBF} = \text{The Next Bloom Filter} \)
30: for \((i=1; i\leq V; i++)\) do
31: Check which bloom filters have already recorded the LBA before
32: \( \text{TotalHotDataIndex} += \text{CalculateTotalHotDataIndex}(	ext{HotBF_Index}) \)
33: end for
34: if (TotalHotDataIndex \geq \text{THRESHOLD}) then
35: Define this as HOT data
36: end if
37: // Decay process
38: if ((RequestCounter \% \text{RESET\_PERIOD})) then
39: Reset all bits in reset bloom filter, ResetBF
40: \( \text{ResetBF} = \text{The Next Bloom Filter} \)
41: end if

Function CalculateTotalHotDataIndex(Index)
42: \( \text{Diff} = \text{The Index of ResetBF} - \text{The Index of Hot Bloom Filter} \)
43: if (\( \text{Diff} \geq 0 \)) then
44: \( \text{HotDataIndexValue} += (2 - (\text{RECENCY}_\text{WEIGHT} \times \text{Diff})) \)
45: else
46: \( \text{HotDataIndexValue} += (\text{RECENCY}_\text{WEIGHT} \times |\text{Diff}|) \)
47: end if
48: Return HotDataIndexValue
number of unique element \((N)\). These three parameters are closely related with each other and have a crucial impact on the performance of its applications. According to \([55]\), we can estimate an optimal BF size \((M)\) using the following formula, \(M = KN/\ln2\), given \(K\) and \(N\). It is clear that the BF size \((M)\) needs to be quite large compared to the size of the element set \((N)\). For example, assuming we adopt two hash functions \((K = 2)\) and there are 4,096 unique elements, we should adopt an 11,819-bit vector as an appropriate size of the BF to minimize a false positive probability. This basic BF allows the elements to be only added to it, but not removed from it. Thus, as the number of input elements grows, the data recorded in the BF are also continuously accumulated, which causes a higher false positive probability. To reduce this, the traditional BF scheme requires a large BF size as described in the formula. However, we cannot simply apply this relationship to both our proposed scheme and multihash function scheme. The BFs in both schemes retain a distinguishing feature from the basic BF—information in the bloom filters can be removed. Especially our proposed scheme completely removes all information in one of the \(V\) BFs periodically. This prevents continuous data accumulation to the BF. Therefore, we can adopt a smaller BF than the aforementioned traditional BF.

Multihash function scheme consists of 4,096 entries of a BF each of which is composed of a 4-bit counter. It also adopts 5,117 write requests \((N)\) as its decay period. This is based on their expectation that the number of hash table entry \((M)\) can accommodate all those LBAs which correspond to cold data within every \(N\) (where, \(N \leq M/(1 - R)\)) write request (here, \(R\) is the hot ratio of a workload and the authors assumed \(R\) is 20\%) \([11]\). To verify their assumption, as displayed in Figure 3.7, we measured the number of unique LBA for every unit (5,117 write requests) under several real traces such as Financial1, Distilled, MSR and RealSSD (these traces will be explained in our experiment section). Figure 3.7 clearly shows that the number of unique LBA very frequently exceeds their BF size of 4,096 (a dotted square) for each unit period (every 5,117 write request) under all four traces. This necessarily causes hash collisions so that it results in a higher false positive probability.

We also measured the average number of unique LBA within both 2,048 and 4,096
write request periods under the same workloads. They vary from 1,548 to 1,991 for 2,048 unit and from 3,014 to 3,900 for 4,096 unit. This is closely related with average hot ratios of each workload: intuitively the higher average hot ratio, the less number of unique LBA. Based on these observations, as our decay interval $T$, we adopt $M/V$ numbers of write requests, where $M$ is a BF size and $V$ is the number of BF. To reduce the probability of a false positive, the BF size (i.e., the number of hash table entries) must be able to accommodate at least all unique LBAs that came in for the last $V \times T$ interval. Thus, whenever $M/V$ numbers of write requests are issued, one of the BFs are selected in a round robin fashion and all the bits in the BF are reset to 0. For example, assuming our BF size is 2,048, our decay period corresponds to 512 write requests. Since memory consumption is also very important factor, we adopt even less (a half) size of BF than the BF size in multihash scheme.

3.2.2.2 WDAC: A Window-based Direct Address Counting

Hsieh et al. proposed an approximated hot data identification algorithm named a direct address method (hereafter, we refer to this as DAM) as their baseline. By using counters DAM can capture the frequency information well, whereas, with respect to recency, it retains the same limitation as the multihash function scheme: all LBAs accessed within the same decay period have an identical recency regardless of their access time or sequence. For instance, assuming the decay period is 4,096 write requests, the LBA accessed in the first request within this period is considered as having the same recency as the LBA accessed in the last (4,096th) request.

To resolve this limitation, in this subsection, we propose a more reasonable baseline algorithm to approximate ideal hot data identification named Window-based Direct Address Counting (WDAC). As shown in Figure 3.8 (a), WDAC maintains a specific size of buffer like a sliding window. In addition, it maintains total hot data index (HDI) values for each LBA (shown in Figure 3.8 (b)).

Within this window, all elements have a different recency value according to their access sequences: the closer to the present, the higher recency weight is assigned to the LBA. That is, highest recency value (i.e., 2) is assigned to the most recently accessed
element (head) in the window, whereas, the lowest recency is allotted to the last element (tail) in the window. All intermediate LBAs in the window have all different recency values with an evenly decreasing manner. Assuming the window size is $W$, we can get the recency difference between two adjacent elements in the window as $2/W$. Thus, all recency weights assigned to each LBA evenly decrease by $2/W$ from the head to the tail in the window. Whenever a new LBA comes in, all recency values are reassigned to all the LBAs shifted in the window and the last one is evicted from the window.

Like our proposed multiple bloom filter-based scheme, as our highest recency weight, we choose 2 (a double weight value) since the total average of all recency values in the window is equivalent to that within the same decay period in the DAM. Instead, unlike the DAM, we assign a higher recency weight to the recently accessed LBAs and vice versa. Consequently, our proposed WDAC can properly identify hot data thereby using
very fine-grained recency information.

Figure 3.8 illustrates a simple example for our WDAC process. This window can contain 10 LBAs and recency difference corresponds to 0.2 (= 2/10). Whenever a write request comes in to the flash translation layer (FTL), this LBA is stored in the head of the window and the last one is evicted. At the same time, all the others are shifted toward the tail of the window and their recency values are reevaluated by decreasing their values by the recency difference (0.2). All of the total hot data index values also need to be updated accordingly. Finally, if the total hot data index value of the corresponding LBA is greater than or equal to a predefined hot threshold, we regard it as hot data.

In our proposed WDAC scheme, since window size is associated with recency and frequency, a proper window size is an important parameter to this scheme. Therefore, the impact of this window size will be discussed in our experiment part.

3.2.3 Performance Evaluation

This section provides diverse experimental results and comparative analyses.

3.2.3.1 Evaluation Setup

We compare our Multiple Bloom Filter-based scheme (hereafter, refer to as MBF) with three other hot data identification schemes: Multiple Hash Function scheme (hereafter, refer to as MHF) [41], Direct Address Method (refer to as DAM) [41], and our proposed baseline scheme named WDAC. We, however, focus particularly on the MHF since it is the state-of-the-art scheme. We adopt a freezing approach [41] as a solution for a counter overflow problem in MHF since it, in our experiments, showed a better performance than the other approach (i.e., exponential batch decay). DAM is an aforementioned baseline proposed in MHF scheme. Table 3.1 shows system parameters and their values. Our scheme (MBF) adopts a half size of a bloom filter as MHF because ours shows a better performance than MHF even with a smaller one. However, we also evaluate both schemes with the same size of a bloom filter for clearer understanding.
Table 3.1: System Parameters and Values

<table>
<thead>
<tr>
<th>System Parameters</th>
<th>MBF</th>
<th>MHF</th>
<th>WDAC</th>
<th>DAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloom Filter Size</td>
<td>2(^{11})</td>
<td>2(^{12})</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Number of Bloom Filter</td>
<td>4</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Decay (Window Size)</td>
<td>2(^{9})</td>
<td>2(^{12})</td>
<td>2(^{12})</td>
<td>2(^{12})</td>
</tr>
<tr>
<td>Number of Hash Function</td>
<td>2</td>
<td>2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Hot Threshold</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Recency Weight Difference</td>
<td>0.5</td>
<td>N/A</td>
<td>0.000488</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3.2: Workload Characteristics

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Total Requests</th>
<th>Request Ratio (Read:Write)</th>
<th>Inter-arrival Time (Avg.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Financial1</td>
<td>5,334,987</td>
<td>R:1,235,633(22%) W:4,099,354(78%)</td>
<td>8.19 ms</td>
</tr>
<tr>
<td>MSR</td>
<td>1,048,577</td>
<td>R:47,380(4.5%) W:1,001,197(95.5%)</td>
<td>N/A</td>
</tr>
<tr>
<td>Distilled</td>
<td>3,142,935</td>
<td>R:1,633,429(52%) W:1,509,506(48%)</td>
<td>32 ms</td>
</tr>
<tr>
<td>RealSSD</td>
<td>2,138,396</td>
<td>R:1,083,495(51%) W:1,054,901(49%)</td>
<td>492.25 ms</td>
</tr>
</tbody>
</table>

For fair evaluation, we assign the same number of write requests (4,096) for a decay interval in DAM as well as for a window size in our proposed WDAC algorithm. Moreover, we employ 4,096 write requests for the decay interval in MHF scheme to synchronize with DAM and WDAC, while we adopt 512 write requests for our decay period. We also adopt identical hash functions for both MBF and MHF schemes. Lastly, since window size is 4,096 in WDAC, there exist about 0.488 \times 10^{-3} (= 2/4,096) weight difference for each element and we also assign 0.5 weight difference for each BF in MBF.

For more objective evaluation, we adopt four real workloads (Table 3.2). Financial1 is a write intensive trace file from the University of Massachusetts at Amherst Storage Repository [56]. This trace file was collected from an Online Transaction Processing (OLTP) application running at a financial institution. Distilled trace file [40] shows a general and personal usage patterns in a laptop such as web surfing, watching movies, playing games, documentation work, etc. This is from the flash memory research group.
The total requests in Table 3.2 correspond to the total number of read and write requests in each trace. These requests can also be subdivided into several or more sub-requests with respect to LBA accessed. For example, let us consider such a write request as WRITE 100, 5. This means 'write data into 5 consecutive LBAs from the...
LBA 100’. In this case, we regard this request as 5 write requests in our experiments.

### 3.2.3.2 Performance Metrics

A hot ratio is a ratio of hot data to all data. First of all, we choose this *hot ratio* to compare each performance of those four hot data identification schemes and to examine closeness among them. However, even though both hot ratios of two algorithms are identical, hot data classification results of both schemes may be able to be different since an identical hot ratio means the same number of hot data to all data and does not necessarily mean all classification results are identical. Thus, in order to make up for this limitation, we employ another evaluation metric: *false identification rate*. Whenever write requests are issued, we try to compare each identification result of each scheme. This enables us to make a more precise analysis of them. *Memory consumption* is another important factor to be discussed since SRAM size is very limited in flash memory. Finally, *runtime overhead* also must be taken in account. To evaluate it,
we choose two main operations and measure CPU clock cycles per operation in each scheme.

3.2.3.3 Results and Analysis

We discuss our evaluation results in diverse respects.

- **Baseline Algorithm:** We first start to evaluate two baseline schemes: our proposed WDAC and DAM. As shown in Figure 3.9 (a) and (b), the hot ratios of both schemes exhibit considerably different results under Financial1 and MSR, whereas they display almost identical patterns under Distilled and RealSSD traces. However, as mentioned before, the identical hot ratios do not necessarily mean the same performance of them. Thus we make another experiment for a more comparative analysis.

Figure 3.10 plots the number of different identification results between WDAC and DAM. We count the number of different hot classification results during a specified unit time (150K or 300K write requests) between them. Thus, zero value means all the identification results are identical between each scheme during the corresponding unit time. Particularly, Figure 3.10 (c) and (d) illustrate well the performance of both schemes can be different even though two hot ratios look very similar each other as shown in Figure 3.9 (c) and (d).

- **Our Scheme (MBF) vs. MHF:** Now, we make an attempt to evaluate performance of our proposed scheme (MBF) and the multiple hash function scheme (MHF). For references, we include our aforementioned two baselines. In particular, we try to compare with DAM as well as WDAC in order to demonstrate that our proposed WDAC is not an unfairly customized baseline algorithm which is best fit for our scheme.

As illustrated in Figure 3.11, our MBF presents a very close approach to our baseline, even to DAM, while the MHF has a tendency to exhibit considerably higher hot ratios than MBF as well as two baselines under all traces. This results from a higher ratio of false identification in MHF. That is, since the BF size (i.e., hash table size) is limited, even though write requests of cold data are increased, the chance of incorrect counter increments also grows due to hash collisions. As a result, it causes higher hot ratios. As Hsieh et al. are mentioned in their paper, the MHF scheme does not show a good
performance especially when the hot data ratio in traces is very low. To get over this limitation, they suggested two solutions: a larger size of the hash table or a more frequent decay operation. However, both suggestions cannot be fundamental solutions since not only does a larger hash table require more memory consumption, but also a more frequent decay produces a significant performance overhead.

Our scheme, on the other side, resolves that limitation by adopting multiple BFIs. Unlike MFH adopting the exponential batch decay after a longer decay period (here, 4,096 write requests), our scheme erases all information only in one BF out of $V$ BFIs (here, $V=4$) after a shorter decay period (here, 512 write requests). Furthermore, a smaller BF results in lower computational overhead as well as less memory space consumption. Both memory consumption and runtime overhead will be discussed in the next subsections in more detail.

Figure 3.12 shows the false identification rates of both our scheme and MHF scheme. We compare both MBF and MHF with our proposed ideal scheme (WDAC). Thus,
we can call these results false identification rates. As presented in Figure 3.12, our MBF scheme exhibits much lower false identification rates than MHF. It improves its performance by an average of 41%, 65%, 59%, and 36% under the four workloads respectively.

- **Runtime Overhead:** Computational overhead is another important factor to evaluate hot data identification scheme. Figure 3.13 displays runtime overheads for a checkup and decay operation in MBF and MHF. A checkup operation means the verification process to check if the data in the corresponding LBA are hot whenever write requests are issued. A decay operation is the aforementioned aging mechanism. Since both are the most representative operations in hot data identification schemes, we choose and evaluate them. To evaluate each operation, we measure CPU clock cycles for them under the configurations in Table 3.1. This measuring is done over an AMD X2 3800+ (2GHz) system with 2G RAM under Windows XP Professional platform. For fair and precise evaluation of each operation in both schemes, we feed many write requests from MSR.
traces (100K numbers of write requests) and then measure their average CPU clock cycles since a CPU clock cycle is dependent on cache misses (such as code miss and data miss) in CPU caches. In other words, when we measure CPU clock cycles of the operations, a lot more clock cycles are required at the beginning stage due to CPU cache misses, while the clock cycles start to be significantly reduced and stabilized soon afterwards.

As presented in Figure 3.13, the runtime overhead of a checkup operation in MBF is comparable to the checkup operation in MHF because both schemes adopt identical hash functions and BF data structures. However, our scheme requires about 6% less computational overhead than MHF since our scheme (MBF) sets only one bit for each hash function, while MHF frequently needs to set more than one bit. That is, since MHF scheme consists of 4-bit counters, one or more bits can need to be set in order to increase the corresponding counters. In the case that a chosen BF in MBF has already recorded the corresponding LBA information, it tries to choose another BF until it finds one available. Intuitively, this process may require extra overhead in comparison with the MHF. However, when it turns out that all BFs (here, 4 BFs) have already included the LBA information, MBF simply defines the data as hot and skips further processes such as a BF checking or recency weight assignment. This shortcut decision in the checkup operation considerably reduces its overhead by an average of 29% compared to the checkup operation without shortcut decision (478 vs. 675) in our scheme (in Figure 3.13 Checkup_WS means the checkup operation without the shortcut decision).

On the other hand, a decay operation of our scheme outperforms that of MHF. Our scheme consists of 4 numbers of 2,048 sizes of 1-bit array, while MHF is composed of 4,096 sizes of 4-bit array. Thus, whenever a decay operation is executed in MHF, all 4,096 numbers of 4-bit counters must be right-shifted by 1-bit at once. Our scheme, however, needs to reset only 2,048 sizes of 1-bit array. Therefore, although our scheme requires more frequent (here, 4 times more frequent) decay operations than MHF, it requires almost a half (58%) less runtime overhead than MHF.

- **Impact of Memory Size:** Since SRAM is very limited in flash memory and expensive (generally 5~10 times more expensive than DRAM), we must consider memory
consumption to design a hot data identification scheme. Thus, evaluating performance of both MBF and MHF with the same size of memory space is also valuable. In addition, we explore the performance changes of both schemes over various memory sizes in this subsection. Our scheme consumes only 1KB, while MHF requires a double (2KB).

As mentioned before, our proposed scheme comprises 4 BFs each of which consists of 0.25KB (i.e., 2K × 1-bit). On the other hand, MHF is composed of 4K numbers of 4-bit counters (i.e., 4K × 4-bit). Therefore, our scheme consumes only a half of memory space of MHF scheme.

Figure 3.14 illustrates false identification rates between two schemes with same memory space. Since MHF originally requires 2KB, we double each BF size in our scheme from 0.25KB to 0.5KB (i.e., 0.5KB × 4 = 2KB). For reference, we include our original MBF (1KB) into each plot. That is, both original MBF and MHF require 1KB and 2KB respectively. As shown in Figure 3.14, MBF with 2KB improves its performance by lowering its false identification rates further than original MBF (1KB), which means our proposed scheme clearly outperforms MHF under the same memory space condition. MBF with 2K benefits from a larger BF size so that it can reduce the possibility of a false positive in each BF.

Different memory spaces (i.e., different bloom filter sizes) will have an effect on the performance of each scheme. We now explore the impact of memory space on both MBF and MHF. Figure 3.14 (d) exhibits performance changes of both schemes over various
memory spaces under RealSSD trace file. Both schemes with larger memory space show a better performance than those with smaller memory space. As a memory space grows, although each performance is also improved accordingly, our proposed scheme still exhibits better performance than MHF throughout all memory spaces from 0.125KB to 4KB. Since other experimental results under the other traces such as Financial1, MSR and Distilled also show almost identical patterns with this experiment, we do not provide the other ones.

- **Impact of Window Size:** Intuitively, a larger window will cause a higher hot ratio because a window size is directly associated with frequency information. All experiments in Figure 3.15 demonstrate well this intuition. Since total hot data index values for each LBA are the summation of all weighted frequency values of the corresponding LBA within the window, the frequency value has a great impact on the hot data decision. A larger window can contain a more number of LBA access information so that it necessarily causes a higher frequency value for each LBA. This results in a higher hot
ratio. Conceptually the decay period in DAM is similar to the window size in WDAC. Thus, we also explore the relationship between various decay periods and hot ratios in DAM. Figure 3.15 presents the changes of average hot ratios over a different window size in WDAC and a different decay period in DAM. Like WDAC, a longer decay period causes a higher hot ratio in DAM. Thus, overall trends of hot ratios in DAM exhibit very similar results.

- **Impact of the Number of a Bloom Filter:** In our proposed scheme, the number of BF corresponds to the granularity of recency. To explore its impact, we make experiments with various numbers of BF in our scheme. For more objective comparison, we not only assign the same memory space (1KB) and the same number of hash function (2) to each scheme, but also configure BF sizes and decay periods accordingly.
As presented in Figure 3.16 (b), as the number of BF grows, the false identification rates also increase (here, we use RealSSD traces). This result is closely related with BF sizes of each scheme. Since each configuration has the same total memory consumption, smaller BF$s$ have to be assigned to the scheme with a more number of BF, which results in higher false identification rates. Even though the scheme with a more number of BF can capture finer-grained recency information, this benefit is offset by its higher false identification rates. To verify this, we make another experiment with the same configurations except for memory consumption. When we double the number of BF from 4 to 8 while remaining the same BF size (i.e., 1KB to 2KB), the performance is improved by an average of 18% since it can benefit from capturing its finer-grained recency.

3.2.4 Conclusion

We proposed a novel hot data identification scheme for flash memory-based storage systems. Unlike the multihash framework, our scheme adopts multiple bloom filters and each bloom filter has a different weight and a different recency coverage so that it can capture finer-grained recency information. Furthermore, multiple and smaller bloom filters empower our scheme to achieve not only lower runtime overheads, but also less memory consumption.

In addition to this novel scheme, we proposed a more reasonable baseline algorithm to approximate an ideal hot data identification named Window-based Direct Address Counting (WDAC). The existing algorithm (i.e., direct address method) cannot properly catch recency information because it assigns an identical recency weight to all LBAs accessed within a decay period. However, our WDAC allocs all different recency weights to all LBAs within a window according to their access sequences so that it can capture precise recency as well as frequency information. Consequently, WDAC can properly identify hot data.

We made experiments in many respects under diverse real traces including real SSD traces. All hot data identification results (i.e., hot ratios) of our scheme display much closer results to those of the baseline scheme than the other one. Furthermore, to make up for the limitation of a hot ratio-based analysis, we also compared the number
(or rate) of false identification by making one-to-one comparison of each identification result. This pinpoint evaluation not only enables us to make a comparative analysis of each performance, but also demonstrates that our scheme more precisely identifies hot data. Lastly, we carried out experiments on variable memory size, window size, and the number of a bloom filter in our scheme to explore their impacts. Our experiments present that our proposed scheme improves the performance up to 65% although its runtime overhead of, in particular, decay operation in our scheme requires less CPU clocks up to 58% and less memory space, by an average of 50%.

3.3 Proposed Scheme 2: A Sampling Technique-based Hot Data Identification

Hot data identification is an issue of paramount importance in flash-based storage devices since it has a great impact on their overall performance as well as retains a big potential to be applicable to many other fields. However, it has been least investigated. HotDataTrap is a novel on-line hot data identification scheme adopting a sampling mechanism. This sampling-based algorithm enables HotDataTrap to early discard some of the cold items so that it can reduce runtime overheads and a waste of memory spaces. Moreover, its two-level hierarchical hash indexing scheme helps HotDataTrap directly look up a requested item in the cache and save a memory space further by exploiting spatial localities. Both our sampling approach and hierarchical hash indexing scheme empower HotDataTrap to precisely and efficiently identify hot data with a very limited memory space. Our extensive experiments with various realistic workloads demonstrate that our HotDataTrap outperforms the state-of-the-art scheme by an average of 335% and our two-level hash indexing scheme considerably improves further HotDataTrap performance up to 50.8%.

3.3.1 Introduction

These days, flash-based storage devices like a Solid State Drive (SSD) have been increasingly adopted as main storage media even in data centers as well as in mobile
devices \textsuperscript{[1, 6, 4, 5]}. However, flash memory has two main limitations in terms of a write operation. First, once data are written in a flash page, an erase operation is required to update them, where the erase is almost 10 times slower than write and 100 times slower than read \textsuperscript{[25]}. Thus, frequent erases severely degrade the overall performance of flash-based storage. Second, each flash block cell can be erased only for a limited number of times (i.e., 10K-100K) \textsuperscript{[25]}.

To resolve these limitations, a flash-based storage device adopts an intermediate software/hardware layer named Flash Translation Layer (FTL). The FTL treats flash memory as a log device; so it stores updated data to a new physical page, marks the old physical page as invalid for future reclamation (called a garbage collection), and maintains logical-to-physical page address mapping information to keep track of the latest location of the logical page \textsuperscript{[47]}. FTL employs a wear leveling algorithm to improve the life span of the flash memory by evenly distributing block erases over the entire flash memory space.

One of the critical issues in designing flash-based storage systems and integrating them into the storage hierarchy is how to effectively identify hot data that will be frequently accessed in near future as well as have been frequently accessed so far. Since future access information is not given as a priori, we need an on-line algorithm which can predict the future hot data based on past behaviors of a workload. Thus, if any data have been recently accessed more than the threshold number of times, we can consider the data hot data, otherwise, cold data. This definition of hot, however, is still ambiguous and takes only frequency (i.e., how many times the data have been accessed) into account. However, there is another important factor—recency (i.e., when the data have been recently accessed)–to identify hot data. In general, many access patterns in workloads exhibit high temporal localities \textsuperscript{[59]}; therefore, recently accessed data are more likely to be accessed again in near future. This is the rationale for including the recency factor in hot data identification. This hot definition can be different for each application and the hot data identification can be applied to many flash-based storage fields.

First of all, hot data identification has a critical impact on the performance and
reliability of the flash memory due to the aforementioned garbage collection and wear leveling [60, 59, 61]. It can be applied to a cache mechanism between DRAM and HDD [62, 63] as well as to SLC-MLC or Flash-PCM hybrid SSD design [64, 65]. It is also applicable to an FTL address mapping scheme, buffer replacement algorithm, and even sensor networks [11, 66, 5, 14]. In addition to these applications in flash memory, hot data identification has a big potential to be utilized by many other fields. Although it has a great impact on designing and integrating flash memory, it has been least investigated. The challenges are not only the classification accuracy, but also a limited resource (i.e., SRAM) and computational overheads. Existing schemes for flash memory either suffer from high memory consumption [67] or incur excessive computational overheads [68].

To overcome these shortcomings, Hsieh et al. recently proposed a multiple hash function framework [69]. This adopts a counting bloom filter and multiple hash functions so that it can reduce memory consumption as well as computational overheads. However, it considers simply a frequency factor and cannot identify hot data with accuracy due to a false positive probability that is an inborn limitation of a bloom filter. Moreover, it is heavily dependent on the workload characteristic information (i.e., an average hot data ratio of a workload) which cannot be given as a priori in an on-line algorithm.

Considering these observations, an efficient on-line hot data identification scheme has to meet the following requirements: (1) capturing recency as well as frequency information, (2) low memory consumption, (3) low computational overheads, and (4) independence of prerequisite information. Based on these requirements, we propose a novel on-line hot data identification scheme named HotDataTrap. The main idea is to cache and maintain a working set of potential hot data by sampling LBA access requests.

HotDataTrap captures recency as well as frequency by maintaining a recency bit like a CLOCK algorithm [70]. Moreover, it can achieve direct cache lookup as well as lower memory consumption with the help of its two-level hierarchical hash indexing scheme. Consequently, our HotDataTrap provides more accurate hot data identification than the state-of-the-art scheme [69] even with a very limited memory. The main contributions of this paper are as follows:
• **Sampling-based Hot Data Identification:** Unlike other schemes, HotDataTrap does not initially insert all cache-missed items into a cache. Instead, it decides whether to cache them or not with our simple sampling algorithm. Once the item is selected for caching, HotDataTrap maintains it as long as it is stored in the cache. Otherwise, HotDataTrap simply ignores it from the beginning.

• **Two-Level Hierarchical Hash Indexing:** HotDataTrap takes advantage of spatial localities. For sequential accesses, HotDataTrap maintains only their starting (partial) LBAs and offset information by adopting an hierarchical data structure and two hash functions.

  Both the sampling mechanism and hierarchical indexing scheme enable HotDataTrap to achieve low runtime overheads as well as less memory consumption.

### 3.3.2 Main Design

#### 3.3.2.1 Architecture

Most of the I/O traces exhibit localities \([60, 59]\) and in case of sequential accesses, only a few LSBs (Least Significant Bits) of LBAs are changed, while most of the other bits of them are not affected. The basic architecture of HotDataTrap is inspired by these observations.
HotDataTrap caches a set of items in order to identify hot items. For each item, it maintains an ID, a counter, and a recency bit (shown in Figure 3.17). The counter is used to keep track of the frequency information, while the recency bit is adopted to check whether the item has been recently accessed or not. To reduce memory consumption, HotDataTrap uses only partial bits (16 bits) out of 32-bit LBA (assuming LBA is composed of 32 bits) to identify LBAs. This 16-bit ID consists of a primary ID (12 bits) and a sub ID (4 bits).

Our proposed scheme adopts two hash functions: one for a primary ID access and the other for a sub ID access. As shown in Figure 3.17, one hash function for the sub ID access captures only the last 4 LSBs, while the other one for the primary ID access takes the following 12 LSBs. This two-level hierarchical hash indexing scheme can considerably reduce cache lookup overheads by allowing HotDataTrap to directly access LBA information in the cache. Moreover, it empowers HotDataTrap to exploit spatial localities. That is, many access patterns in workloads generally exhibit high spatial localities as well as temporal localities [59]. For sequential accesses, HotDataTrap stores the starting LBA in a sequential access to both the primary ID and the sub ID, and its offset addresses only to the sub IDs, not to the primary IDs. Consequently, it can significantly reduce memory space consumption.

However, this partial LBA capturing can cause a false LBA identification problem. To verify that our partial ID can provide enough capabilities to appropriately identify LBAs, we made an attempt to analyze four workloads (i.e., Financial 1, MSR, Distilled, RealSSD). These traces will be explained in detail in our evaluation section (Section 3.3.3.3).

Figure 3.18 shows false LBA identification rates for each trace. In this figure, F_{16}(20, 24)bit stands for Financial1 traces capturing 16(20, 24) LSBs in LBAs respectively. We first captured the aforementioned 16 LSBs to identify LBAs and then made a one-to-one comparison with the whole 32-bit LBAs. As plotted in Figure 3.18, MSR, Distilled, and RealSSD traces present very low false LBA identification rates, which means we can use only 16 LSBs to identify LBAs. On the other hand, Financial1 trace file shows a different characteristic from the others. When we captured the 16
LSBs of the Financial1 traces, it exhibited a relatively higher false identification rate than those of the other three traces. However, when we adopted 24 LSBs instead, it showed a zero false identification rate, which means the 24 LSBs can perfectly identify all LBAs in the Financial1 traces. To clarify this reason, we made another analysis of those four traces.

Figure 3.19 represents a 100% stacked column chart that is generally used to emphasize the proportion of each data. As displayed in this figure, unlike the others, the Financial1 traces tend to access only a very limited range of LBAs (a dotted square part), whereas MSR, Distilled, and RealSSD traces access much wider ranges of LBAs.
This means Financial1 traces intensively access only a very limited space so that the variance of all accessed LBAs in the workloads is even smaller than the others. Therefore, capturing 16 LSBs in Financial1 causes a relatively higher false LBA identification rate than the others. In this case, if we need a more precise identification performance according to the applications, we can increase our primary ID size, not the sub ID size. Even though we increase our primary ID size, we still can save a memory space with the sub ID by exploiting spatial localities. Moreover, although HotDataTrap adopts 16 LSBs capturing in our experiments, our scheme outperforms the state-of-the-art scheme even in the Financial1 traces. This will be verified in our performance evaluation section (Section 3.3.3.3) later.

### 3.3.2.2 Overall Working Processes

HotDataTrap works as follows: whenever a write request is issued, the request LBA is hashed by two hash functions as described in the section [3.3.2.1](#) to check if the LBA has stored in the cache. If the request LBA hits the cache, the corresponding counter value is incremented by 1 to capture the frequency and the recency bit is set to 1 for recency capturing. If the counter value is greater than or equal to the predefined hot threshold value, it is classified as hot data, otherwise cold data. In case of a cache miss, HotDataTrap makes a decision on inserting this new request LBA into the cache by using our sampling-based approach.
Unlike other hot data identification schemes or typical cache algorithms which initially insert all missed items into the cache and then selectively evict a useless item from the cache afterwards, our HotDataTrap initially stores only selected items in the cache from the beginning. Due to a limited cache space, it would be ideal if we could store only an item that will become hot in near future. However, it is impossible unless we can see the future. Even though we try to predict the future accesses based on the access history, we must pay extra overheads to manage considerable amount of the past access information. This is not a reasonable solution especially for the hot data identification scheme because it must be triggered whenever every write request is issued.

To resolve this problem, HotDataTrap tries to do sampling with 50% probability which is conceptually equivalent to tossing a coin. This simple idea is inspired by the intuition that if any LBAs are frequently accessed, they are highly likely to pass this sampling. This sampling-based approach helps HotDataTrap early discard an infrequently accessed LBA. It, consequently, can reduce not only memory consumption, but also computational overheads. The effectiveness of this sampling-based approach will be verified in our experiments (Section 3.3.3.3).

A hot data identification scheme needs an aging mechanism which decays the frequency information. Like the Multi-hash function scheme [69], HotDataTrap periodically divides the access counter values by two for its aging mechanism, which enables hot data identification schemes to partially forget their frequency information as time goes on. Thus, if any data have been heavily accessed in the past, but not afterwards, they will eventually become cold data and will be selected as victims in the cache soon.

Moreover, HotDataTrap retains another mechanism for better hot data identification by capturing recency as well as frequency information. Both a recency capturing mechanism and an aging mechanism look similar, but are clearly different: the main goal of the aging mechanism is to regularly decay the frequency information over time as well as to prevent a counter overflow problem. On the other hand, a recency capturing mechanism intends to reflect recent access information. Thus, the aging mechanism in Multi-hash cannot appropriately capture recency information. For instance, the Multi-hash scheme does not provide any mechanism to check if any data have been recently
accessed because its aging mechanism simply divides all access counts by two periodically. However, the aging mechanism in HotDataTrap periodically resets all recency bits to 0, and its recency capturing mechanism sets the corresponding recency bit to 1 if any data are accessed.

Figure [3.20](image) illustrates our victim selection process. If the cache is full and we need to insert a newly sampled LBA into the cache, HotDataTrap chooses a victim. To reduce victim search overheads, HotDataTrap maintains a victim candidate list. If an access count is smaller than the predefined hot threshold value and its recency bit is reset to 0, such LBA is classified into a victim candidate. Whenever HotDataTrap executes its decay process, it stores those victim candidates into the list. That is, the victim list is periodically updated for each decay period to reflect the latest information. Thus, when HotDataTrap needs to evict a cold item from the cache, it first selects one victim candidate from the list and directly checks if the candidate can be removed (i.e., it should meet two aforementioned requirements for an eviction: (1) the access count is less than the HotThreshold value and (2) the recency bit is reset to 0). If the candidate is still cold, HotDataTrap deletes the candidate and inserts the new item into the cache. If the candidate has turned to hot data since the last decay period, we cannot evict it. In this case, HotDataTrap checks another candidate from the list. The victim candidate list can notably save the victim search overheads by directly looking up the victim.

- **Algorithm**: Algorithm 2 provides a pseudocode of our HotDataTrap algorithm. When a write request is issued, if the LBA hits the cache, its corresponding access counter is incremented by 1 and its recency bit is set to 1. If the counter value is equal to or greater than a predefined HotThreshold value, it is identified as hot data, otherwise, cold data. If it does not hit the cache, HotDataTrap tries to do sampling by generating a random number between 0 and 1. If the number passes the sampling test (i.e., belongs to its sampling range), the LBA is ready to be added into the cache. If the cache has a space available, HotDataTrap just inserts it into the cache and sets both the access counter and recency bit to 1. Otherwise, HotDataTrap needs to select and remove a victim whose access counter value is less than HotThreshold value and recency bit was reset to 0. Then, it stores the newly sampled LBA into the cache. If
the new request fails to pass the sampling test, HotDataTrap simply discards it.

**Algorithm 2** HotDataTrap

1: Input: write requests for LBAs
2: Output: hot or cold data classification of the LBAs
3: A write request for an LBA $x$ is issued
4: if (Cache Hit) then
5: Increase the counter for $x$ by 1
6: Set the recency bit to 1
7: if (Counter $\geq$ HotThreshold) then
8: Classify $x$ as hot data
9: end if
10: else
11: if (Passed a sampling test) then
12: if (Cache is not full) then
13: Insert $x$ into the cache
14: Set the counter for $x$ to 1
15: Set the recency bit to 1
16: else
17: end if
18: // Need to evict data
19: while (The current candidate is not a victim) do
20: Move to the next candidate in the list
21: Check if the candidate can be evicted
22: end while
23: Evict the victim and insert $x$ into the cache
24: Set the counter for $x$ to 1
25: Set the recency bit to 1
26: end if
27: else
28: // If failed in the sampling test
29: Skip further processing of the $x$
30: end if
31: end if

### 3.3.3 Performance Evaluation

#### 3.3.3.1 Evaluation Setup

We compare our HotDataTrap with a Multi-hash function scheme \[69\] that is the state-of-the-art scheme and a direct address method (refer to as DAM) \[69\]. We select a freezing approach (i.e., when an LBA access counter reaches its maximum value due to a heavy access, it does not increase the counter any more even though the corresponding LBA is continuously accessed) as a solution for a counter overflow problem both in Multi-hash scheme and HotDataTrap since it, according to our experiments, showed a better performance than the other approach (i.e., exponential batch decay: divides all counters by two at once). DAM is an aforementioned baseline algorithm. Table \[3.3\] describes
system parameters and their values.

For fair evaluation, we adopt an identical decay interval (4,096 write requests) as well as aging mechanism (i.e., exponential batch decay) for all those schemes. For more objective evaluation, we employ four realistic workloads (Table 3.4). Financial1 is write intensive block I/O traces from the University of Massachusetts at Amherst Storage Repository [56]. This trace file was collected from an On-Line Transaction Processing (OLTP) application running at a financial institution. Distilled trace file [68] represents a general and personal usage patterns in a laptop such as web surfing, documentation work, watching movies, playing games, etc. This is from the flash memory research group repository at National Taiwan University, and since Multi-hash scheme employed only this trace file for its evaluation, we also adopt this trace for fair comparison. We also select Microsoft Research Trace (refer to as MSR) made up of 1-week block I/O traces of enterprise servers at Microsoft Research Cambridge Lab [57]. We selected, in particular, prn volume 0 trace since it exhibits a write intensive workload [58]. Lastly, we adopt a real Solid State Drive (SSD) trace file that is 1-month block I/O traces of a desktop computer (AMD X2 3800+, 2G RAM, Windows XP Pro) in our lab (hereafter, refer to as RealSSD trace file). We installed Micron’s C200 SSD (30G, SATA)
to the computer and collected personal traces such as computer programming, running simulations, documentation work, web surfing, watching movies, etc.

3.3.3.2 Performance Metrics

We first select a hot ratio to evaluate each performance of those schemes. A hot ratio represents a ratio of hot data to all data. Thus, by comparing each hot ratio of both schemes (HotDataTrap and Multi-hash) with that of the baseline scheme (DAM), we can observe their closeness to the baseline result. However, even though both hot ratios of two schemes are identical, hot data classification results of both schemes may not be able to be identical since an identical hot ratio means the same number of hot data to all data, which does not necessarily mean all classification results are identical. Thus, we need another different metric to make up for this limitation. To meet this requirement, we adopt a false identification rate. Whenever write requests are issued, we try to make a one-to-one comparison between two hot data identification results of each scheme. This

![Figure 3.21: Hot Ratios of Each Scheme under Various Traces](image)
allows us to make a more precise analysis of them. Memory consumption is another important factor to be discussed since the SRAM size is very limited in flash memory and we need to exploit it. Lastly, we must take runtime overheads into account. To evaluate these overheads, we measure CPU clock cycles per each representative operation.

3.3.3.3 Evaluation Results

We discuss our evaluation results in diverse respects.

• Overall Performance: We first start to discuss hot ratios of each scheme under four realistic workloads. As shown in Figure 3.21, our HotDataTrap exhibits very similar hot ratios to the baseline scheme (DAM), while the Multi-hash scheme has a tendency to show relatively even higher hot ratios than DAM. This results from the inborn limitation of a bloom filter-based scheme. If a bloom filter size is not large enough to accommodate all distinct input values, the bloom filter necessarily causes a false positive problem. There exists a fundamental trade-off between a memory space (i.e., a bloom filter size)
and its performance (i.e., a false positive rate) of the bloom filter-based scheme. To verify this, we measure the number of LBA collision for each unit period (i.e., decay period) under those traces. In other words, although the working space of the LBAs is much larger than the size of bloom filter (4,096) in the Multi-hash scheme, all of them have to be mapped to the bloom filter by using hash functions. Thus, many of them must be overlapped to the same positions of the bloom filter so that it causes hash collisions. As plotted in the Figure 3.23, we can observe many collisions (on average, 877) for each unit request (4,096 writes) for each trace. All of these collisions do not necessarily mean the number of false identification of the Multi-hash because it can reduce the possibility by adopting multiple hash functions. However, considering this factor, many collisions necessarily have considerable effect on false identification rates in the Multi-hash scheme. Figure 3.22 demonstrates well this analysis. Multi-hash shows much higher false identification rates than our HotDataTrap. HotDataTrap makes a even more precise decision on hot data identification by an average of 335%.

As we discussed in Section 3.3.2.1, due to the distinguished workload characteristics of Financial1 (i.e., Financial1 intensively accesses only a very limited working space), it shows a relatively higher false LBA identification rate than the other traces when we capture the 16 LSBs to identify LBAs. Figure 3.22 verifies our aforementioned discussion: Financial1 shows a higher false hot data identification rate than the others.

- **Runtime Overheads:** We evaluate runtime overheads of two major operations in
hot data identification scheme: a checkup and decay operation. A checkup operation corresponds to a verification process to check whether the data are hot or not. A decay operation is the aforementioned aging mechanism that regularly divides all access counts by two. To evaluate the runtime overheads, we measure CPU clock cycles of them. They are measured on the system with AMD X2 3800+ (2GHz, Dual Core) and 2G RAM under Windows XP Professional platform. For a fair and precise measurement, we fed 100K numbers of write requests from Financial1 traces to each scheme and ran 100 times of each operation to measure average CPU clock cycles. Since the CPU clock count depends heavily on level-1 cache in the CPU, if both codes and data are not in the level-1 cache, the CPU clock count is enormously high. This is the main reason the CPU clock count at the first run is excessively higher than the subsequent counts that represent the time it takes when everything is in the level-1 cache.

As shown in Figure 3.24 (a), the runtime overheads of a checkup operation in HotDataTrap is almost comparable to that of the Multi-hash checkup operation. Although the HotDataTrap requires a few more cycles than Multi-hash scheme, considering both standard deviations (28.2 for Multi-hash and 25.6 for HotDataTrap) for each scheme, they can be ignorable. This result is intuitive because both schemes not only adopt the same number of hash functions, but also can directly look up and check the data in the cache. However, the decay operation of HotDataTrap requires much lower (40.7%) runtime overheads than that of Multi-hash since Multi-hash must decay 4,096 entries,
while HotDataTrap only has to treat even less number of entries (approximately 1/3 of the Multi-hash, but it depends on workload characteristics) for each decay interval. For each entry, Multi-hash requires 4 bits, while our scheme needs 20 bits or 8 bits. Assuming HotDataTrap entries are 1/3 of Multi-hash entries to decay, its runtime overheads would also be around 1/3 of Multi-hash overheads. However, our scheme requires extra processes to update its victim candidate list during the decay process. Thus, its overall runtime overheads reach almost 60% of Multi-hash scheme’s. In addition, HotDataTrap must consider extra overheads: cache management. Multi-hash does not retain this overhead since it uses a bloom filter, while our scheme manages a set of potential hot items in the cache. However, we can significantly reduce these overheads with the help of our sampling approach since the requests failing to pass the sampling test will be discarded without any further process. Figure 3.24 (b) plots the impact of various sampling rates. As the sampling rate grows, the runtime overheads also increase.

**Impact of Memory Spaces:** In this subsection, we assign a different memory space to each scheme from 0.5KB to 8KB and observe their impacts on both schemes. The Multi-hash scheme consumes 2KB (4 × 4,096 bits). As plotted in Figure 3.25 (a), both schemes exhibit a performance improvement as a memory space grows. However, HotDataTrap outperforms Multi-hash, and moreover, the Multi-hash scheme is more sensitive to a memory space. This result is closely related to the false positives of a bloom filter since the false positive probability decreases as a bloom filter size increases.
Figure 3.26: Impact of Decay Intervals. HotDataTrap is not sensitive to a decay period.

On the other hand, HotDataTrap shows a very stable performance even with a smaller memory space, which means our HotDataTrap can be well employed even in embedded system devices with a very limited memory space. Figure 3.25 (b) exhibits hot ratios of each scheme over various memory spaces. All hot ratios increase as a memory size increases because their decay periods also increase accordingly. A longer decay period allows each scheme to accept more requests, which causes higher hot ratios.

• **Impact of Decay Intervals:** A decay interval has a significant impact on the overall performance. Thus, finding an appropriate decay interval is another important issue in the hot data identification scheme design. Multi-hash adopts a formula, \[ N \leq M / (1 - R), \]
as its decay interval, where \( N \) is a decay interval, \( M \) and \( R \) correspond to its bloom filter size and an average hot ratio of a workload respectively. The authors say this formula is based on their expectation that the number of hash table entry (\( M \)) can accommodate all those LBAs which correspond to cold data within every \( N \) write request (they assumed \( R \) is 20\%). According to the formula, assuming the bloom filter size (\( M \)) is 4,096 bits, they suggested 5,117 write requests as their optimal decay interval. To verify their assumption, we tried to measure the number of a unique LBA for every 5,117 write request under all those four traces. According to our observation (due to a space limit, we do not display this plot here), a large number of unique LBAs exceeded the bloom filter size of 4,096, which necessarily causes many hash collisions so that it results in a higher false positive probability. Moreover, initially fixing an average hot ratio of
a workload to 20% is not reasonable since each workload has different characteristics and we cannot know their average hot ratios in advance. Lastly, even though the authors recommended 5,117 writes as their optimal decay interval, our experiments demonstrated that Multi-hash showed a better performance when we adopted 4,096 writes as its decay interval. Thus, we chose 4,096 writes as its decay interval due to not only a performance issue, but also a fairness issue.

Now, in order to observe the impact of a decay interval on each scheme, we fix the memory space (2KB, i.e., a bloom filter size of 4,096 bits for Multi-hash) for each scheme and change the decay interval from 2,048 to 16,384 write requests. Figure 3.26 displays a performance change over diverse decay intervals. As shown in Figure 3.26 (b), the performance of Multi-hash is very sensitive to the decay intervals. Its false hot data identification rate almost exponentially increases as a decay interval grows. For instance, when the interval increases 4 times (from 4,096 to 16,384), its false identification rate increases approximately 10 times (9.93 times). In particular, the false identification rate
identification rate reaches even on average 72.4% if the decay interval is 16,384. This results from a higher false positive probability of the bloom filter. On the other side, our HotDataTrap shows a stable performance irrespective of decay intervals (Figure 3.26 (a)), which demonstrates well that the decay interval does not have a critical influence on HotDataTrap performance. Therefore, even though we cannot find an optimal decay interval, it is not a critical issue in HotDataTrap unlike Multi-hash.

- **Effectiveness of Hierarchical Hash Indexing:** HotDataTrap is designed to exploit spatial localities in data accesses by using its two-level hierarchical hash indexing scheme. In this subsection, we explore how much we can benefit from it. To evaluate its benefit, we prepared two different HotDataTrap schemes: one for our original HotDataTrap (which initially retains the indexing scheme) and the other for HotDataTrap without two-level indexing scheme (referred to as HotDataTrap_WT). That is, HotDataTrap_WT always use a primary ID as well as its sub ID even though the traces show sequential accesses, which can require a more memory space than the HotDataTrap. Figure 3.27 compares both performances and demonstrates the effectiveness of our two-level hierarchical hash indexing scheme. As shown in the plots, HotDataTrap improves its performance, by an average of 19.7% and up to 50.8% thereby lowering false hot data identification.

- **Impact of Sampling Rates:** When a request LBA does not hit the cache, it may or may not be inserted into the cache according to a sampling rate of HotDataTrap.
Thus, the sampling rate is another factor to affect an overall HotDataTrap performance. Figure 3.28 plots overall impacts of the sampling rates. As shown in Figure 3.28 (a), average hot ratios for each trace gradually approach to those of DAM since a higher sampling rate can reduce a chance to drop potential hot items by mistake. Although this higher sampling may cause higher running overheads due to a more frequent cache management process (Figure 3.24), it can be expected to achieve a better performance as displayed in Figure 3.28 (b). False identification rates, intuitively, decrease as the sampling rate grows. Since HotDataTrap with 10% sampling rate shows an even better performance than the Multi-hash scheme by an average of 65.1%, it is acceptable that we select a 50% sampling rate as our initial sampling rate for HotDataTrap. Instead, we can choose a different sampling rate according to application requirements. If an application requires a precise hot data identification performance at the expense of runtime overheads, it can adopt a 100% sampling rate, not lower sampling rates, and vice versa.

3.3.4 Conclusion

We proposed a novel on-line hot data identification scheme named HotDataTrap. HotDataTrap is a sampling-based hot data identification scheme. This sampling approach helps HotDataTrap early discard some of the cold items so that it can reduce runtime overheads as well as a waste of a memory space. Moreover, the two-level hierarchical hash indexing enables HotDataTrap to directly look up items in the cache and to reduce a memory space further by exploiting spatial localities.

We made diverse experiments in many respects with various realistic workloads. Based on various performance metrics, we carried out experiments on the runtime overheads and the impacts of memory sizes, sampling rates, and decay periods. In addition, we also explored the effectiveness of our hierarchical indexing scheme by comparing two different HotDataTrap schemes with and without the indexing scheme respectively. Our diverse experiments presented that HotDataTrap achieved a better performance up to a factor of 20 and by an average of 335%. The two-level hash indexing scheme improved
a HotDataTrap performance further up to 50.8%. Moreover, HotDataTrap was not sensitive to its decay interval as well as a memory space. Consequently, HotDataTrap can be well employed even in small embedded devices with a very limited memory space.
Chapter 4

Application to A Storage Device and System

4.1 Application 1: A New Flash Translation Layer (FTL) Design

The Flash Translation Layer (FTL) is a software/hardware layer inside NAND flash memory that allows existing disk-based applications to use it without any significant modifications. Since FTL has a critical impact on the performance and reliability of the flash-based storage, a variety of FTL schemes have been proposed. The existing FTLs, however, are designed to perform well for either a read intensive workload or a write intensive workload, but not for both due to their internal address mapping schemes. To overcome this limitation, in this paper we propose a novel hybrid FTL scheme named Convertible Flash Translation Layer (CFTL). CFTL is adaptive to data access patterns with the help of our unique hot data identification algorithm. Thus, CFTL can dynamically switch its mapping scheme to either a page level mapping or a block level mapping in order to fully exploit the benefits of both schemes. By judiciously taking advantage of both schemes, CFTL resolves the intrinsic problems of the existing FTLs. In addition to this convertible scheme, we propose an efficient caching strategy.
and adaptive partitioning of mapping tables to further improve the CFTL performance. Consequently, both the convertible feature and the caching strategies empower CFTL to achieve good read performance as well as good write performance. Our experimental evaluation with a variety of realistic workloads demonstrates that CFTL outperforms other existing FTLs. In particular, our new caching strategy remarkably improves a cache hit ratio, by an average of 245%, and achieves much higher hit ratios, especially for randomly read intensive workloads.

4.1.1 Introduction

Several strong features of flash memory have enabled flash-based storage devices to become popular even in enterprise servers as well as in personal mobile devices. These features include fast access speed, low power consumption, high shock resistance, and portability [7, 6, 4, 71, 72]. Recently, flash-based Solid State Drives (SSDs) have attracted considerable attention because they are increasingly being adopted to enterprise servers as well as to personal computers as a main storage alternative for conventional magnetic disk drives [73, 25, 74, 75, 1].

In flash memory, space is partitioned into blocks, where each block contains a fixed number (32 or 64) of pages. Unlike magnetic disk drives, flash memory requires no mechanical movement and therefore exhibits as fast of random read performance as sequential read. However, a major drawback is that in-place updating (i.e., overwriting) is not allowed. That is, an overwrite to pages in a flash memory must be preceded by an erasure of the corresponding block. To promote efficiency, a clean (i.e., already erased) block is often used and all the valid (i.e., not outdated) pages within the block need to be copied into the clean block. Then, the original data block is marked as invalid and will be erased by a garbage collector for reclamation in the future. This in-place update problem results from its asymmetric operational granularity: both read and write operations are performed on a page basis, while erase operations operate on a block basis. According to [25], read operations take 15 μs, write operations take 200 μs, and erase operations take 2000 μs. This is the main reason erase operations severely degrade the overall performance of flash memory. Therefore, reducing the number of
erase operations is one of the most critical issues in flash memory. Moreover, balancing the erase count of each cell block, called wear leveling, is also another crucial issue due to the limited life span of a cell block in flash memory [25]. To resolve these problems, the flash translation layer (FTL) has been designed and deployed.

The FTL is a software/firmware layer implemented inside a flash-based storage device to make the linear flash memory device act like a magnetic disk drive. As shown in Figure 2.1 it is largely composed of an address allocator, garbage collector, wear leveler, and hot data identifier. Out of these main components, we focus mainly on the address allocator which is in charge of address translation and block assignment. FTL emulates disk-like in-place updates for a Logical Page Number (LPN) based data page as follows: FTL first acquires a clean page and updates the corresponding data on that page. Then, it maps the original LPN into this new Physical Page Number (PPN). Thus, an efficient FTL scheme has a critical effect on overall performance of flash memory since it directly affects in-place update performance and balancing the wearing of each data block.

Existing FTLs employ various logical to physical mapping strategies. The mapping can be maintained at the page level, block level, or a combination of both (hybrid scheme). A page level mapping [76] offers the best read/write performance, but it requires a very large memory space to store its entire page mapping table. An update to a page in a page level mapping may not have to trigger a block erasure since it can use any clean page in any block for updating, while an update to a page in a block level mapping will trigger the erasure of the block containing the page since each page only can be mapped into another page in a different block with the same offset. Thus, the performance of a block level mapping for write intensive workloads is much worse than that of a page level mapping. However, for the read intensive workloads, the performance of a block level mapping is comparable to that of a page level mapping with much less memory space.

To take advantage of both the page and block level mapping, although various hybrid mapping schemes [8, 10, 77, 9, 13] have been proposed, they still suffer from read/write performance degradation because they are originally based on a block level mapping
with an additional page level mapping restricted only to a small number of log blocks. Recently, Gupta et al. proposed a page level mapping scheme called DFTL (Demand-based Flash Translation Layer) [12]. They tried to address the memory space limitation problem by storing its complete mapping table in flash. However, DFTL incurs page mapping lookup overhead for workloads with less temporal locality. In addition, it suffers from frequent updates to the pages storing the page mapping table in flash for write intensive workloads and garbage collection.

Considering the advantages and disadvantages of the existing FTLs and the tradeoff between performance and required memory space, we have made the following observations: 1) With less memory space, a block level mapping has a good performance for read intensive workloads due to its fast direct address translations, 2) A page level mapping manages write intensive workloads well with high block utilization and less erase operations, 3) Spatial locality as well as temporal locality in workloads can help to improve FTL performance, and 4) Dynamic cache assignment can make the best of a limited memory space. Based on these observations, our goal is to design an FTL scheme with limited memory space that adapts to the workload behaviors. For a write intensive workload, it will provide the page level mapping-like performance for hot data, while for a read intensive workload, it will provide the block level mapping-like performance for cold data. A data page updated frequently will be considered hot data and a data page read intensively with very infrequent or no updates will be considered cold data. However, hot data may turn into cold data or vice versa from time to time in a workload. The challenge is how to effectively deal with these conversions.

In this paper, we propose a novel hybrid FTL scheme named CFTL (Convertible Flash Translation Layer). CFTL, unlike other existing hybrid FTLs, is fundamentally rooted in a page level mapping. This is a very meaningful transition in the design paradigm of a hybrid FTL because the core of the existing hybrid FTLs is mostly based on a log-structured block level mapping. Thus, they cannot completely overcome the inherent limitation (i.e., lower write performance) of the block level mapping scheme. However, the core mapping table of CFTL is a page level mapping so that CFTL can fully exploit the main benefit (i.e., good write performance) of page level mapping.
Furthermore, it takes advantage (i.e., good read performance with less memory) of the block level mapping by using its convertible feature. The key idea is that the corresponding mapping scheme is dynamically changed according to the data access patterns. In CFTL, since the mapping table is stored in the flash memory, there can be an overhead to lookup the mapping table like DFTL. To further reduce this overhead, we also designed an efficient caching scheme to get extra benefits from the spatial locality in workloads and an adaptive partitioning scheme of mapping tables on the cache to exploit a small memory space.

Our proposed FTL scheme is not restricted only to SSD design, but more generic flash-based storage systems that can be used in small mobile devices. Unlike SSDs, we cannot install a large size of RAM into these devices. In either case, to reduce the hardware cost we must exploit the smaller memory embedded in the flash memory to implement FTL. The main goal is to explore more efficient mapping between logical addresses to physical addresses with good performance and limited memory space requirement. The main contributions of this paper are as follows:

- **A Convertible FTL Scheme:** CFTL is adaptive to data access patterns: A block level mapping deals with read intensive data to make the best of fast direct address translation, and a page level mapping manages write intensive data to minimize erase operations. Therefore, some parts of flash are addressed by a page level mapping, while other parts are addressed by a block level mapping. In addition, the mapping can be dynamically switched to either scheme according to current access patterns of the underlying data (Section 4.1.3.2).

- **A New Hot Data Identification Scheme:** In CFTL, hot data identification plays an important role in switching address modes from a block mapping to a page mapping, or vice versa. Thus, we developed a novel hot data identification scheme adopting multiple bloom filters. This multiple bloom filters-based scheme can capture finer-grained recency information as well as frequency information so that CFTL can achieve more accurate hot data classification (Section 4.1.3.2).
• **An Efficient Caching Strategy:** For the fast address translation, CFTL employs two small caches to store the mapping data and speed up both the page and block level address translations respectively. In particular, the page level cache is specially designed to make the best use of spatial locality in workloads. Thus, even though the cache does not retain the requested mapping information, CFTL can make the request hit the cache with only a simple hint. Consequently, it significantly improves the cache hit ratio and makes considerable contributions to improve the overall performance by exploiting both temporal and spatial localities (Section 4.1.3.4).

• **Adaptive partitioning of mapping tables:** CFTL does not statically assign memory space to both mapping tables (i.e., page and block) on the cache. Instead, if workloads exhibit write intensive patterns, we can more benefit from a page level mapping than a block level mapping. CFTL, thus, dynamically allots more cache space to the page mapping table in SRAM and vice versa. This adaptive partitioning of both mapping tables makes the effective use of a limited memory space in flash-based storages (Section 4.1.3.5).

### 4.1.2 Background and Related Work

This section describes the characteristics of flash memory and various address mapping schemes including page level, block level, and hybrid mappings. Lastly, merge operations are explained.

#### 4.1.2.1 The Characteristics of Flash Memory

There are two types of flash memory: NOR and NAND flash memory. NOR flash memory has a very similar interface to block devices, fast read speed, and access in a byte unit. So it is more appropriate to save and execute program codes. NAND flash, on the other hand, has a distinct interface to block devices, relatively slower read speed, and allows access data in a page unit [78]. These features make NAND flash more relative media to save data, not to program codes. Consequently NAND flash
memory is widely used in flash memory-based storage devices such as SSD, PDA, and other mobile devices. In this paper, we focus on the NAND flash memory.

Figure 4.1 gives an architecture of flash memory. In a NAND flash memory, data are organized as an array of blocks comprised of either 32 pages (small block NAND flash) or 64 pages (large block NAND flash). Each page contains both sector(s) to store data and spare area(s) to store various meta data. That is, the spare area is used to record the LPN (Logical Page Number) corresponding to the data stored in the data area, the ECC (Error Correction Code) to check data correctness, and the state information (i.e., Valid, Invalid, and Clean) of the corresponding page. Since flash memory does not allow in-place updates, updated data must be written to another clean page if an update operation is issued. This outdated page is called an invalid page because its data are not valid anymore. This out-of-place update causes numerous invalid pages and reduces the number of clean pages as time goes on; we therefore need another mechanism to reclaim those invalid pages or blocks, which is called garbage collection.

The garbage collector reclaims the invalid pages by erasing the corresponding blocks and marking them as clean blocks. Finally, since each flash memory cell has a limited lifespan, the limited number of erasure (10K-100K) is allowed [25]. Thus, evenly distributing each block erase count over the entire flash memory, called wear leveling, is also a crucial issue in flash memory.
4.1.2.2 Address Mapping Schemes

Typical address mapping procedures in FTL are as follows: on receiving a logical page address from the host system, FTL looks up the address mapping table and returns the corresponding physical address. When the host system issues overwrite operations, FTL redirects the physical address to a clean location in order to avoid erase operations. Then the updated data are written to the new physical location. After the overwrite operation, FTL updates the address mapping information and the outdated invalid block can be erased later by a garbage collection process. FTL maintains the mapping table information either in the page-level, block-level, or a hybrid manner.

- **Page Level Mapping:** This is a very flexible scheme in that a logical page can be mapped into any physical page in flash memory (Figure 4.2). So it does not require expensive full merge operations [12, 79]. This shows a good overall performance for both read and write operations. It, however, requires a large size of memory to store the entire mapping table. For instance, 1 TB of flash requires 4GB of memory space for the mapping table (assuming a 2KB page and 8 bytes per mapping entry). DFTL (Demand-based Page Level FTL) [12] is a page level FTL scheme. It tries to overcome this innate limitation by storing its complete page mapping table on flash memory, instead of storing it in the SRAM. This approach can save memory space but incur extra flash memory lookup overhead whenever read/write requests are issued. DFTL is a page level mapping FTL so that it completely removes full merge operations. Consequently, it exhibits a competitive overall read/write performance. However, DFTL suffers from frequent updates to its flash mapping table in case of write dominant access patterns or garbage collection. In addition, it achieves a good write performance but cannot achieve as good read performance as a pure page level mapping or some hybrid schemes especially under random read dominant environments (i.e., low temporal locality) workloads due to its intrinsic address translation overhead. Furthermore, DFTL considers temporal locality but leaves spatial locality unaccounted. In many cases, spatial locality is also an essential factor in efficient data accesses [48]. Budilovsky et al. [80] recently proposed a prototype of an SSD consuming a small amount of RAM by caching address mapping
information on the host DRAM, not on the RAM inside SSD. This prototype also adopts two-level address mapping like DFTL \cite{12} or CFTL \cite{11,14}. It, therefore, necessarily causes extra overheads of its address translation because extra flash reads are required to fetch mapping information from the mapping table stored in flash memory. To reduce this address translation overheads, they proposed a host-assisted hinting mechanism that caches the mapping information on the host DRAM. When the SSD receives a hint (i.e., address mapping information) from the host before the actual read/write request is issued, it uses the mapping in the hint instead of reading the mapping from the flash, which can save one read latency.

- **Block Level Mapping:** In a block level address mapping, a logical page address is made up of both a logical block number and an offset. This can reduce the memory space requirements for mapping information. However, when overwrite operations to logical pages are issued, the corresponding block must be migrated and remapped to a clean physical block. That is, the valid pages and the updated page of the original data block are copied to a new clean physical block before the original physical block can be erased. When it comes to a block level mapping, this *erase-before-write* characteristic is an unavoidable performance bottleneck in write operations.
• **Hybrid Mapping**: To overcome the shortcomings of the page and block level mappings, diverse hybrid schemes have been proposed [8, 10, 6, 9, 13]. Most of these algorithms are based on a log buffer approach by adopting a limited number of log blocks so that they can improve the write performance. The memory usage for mapping can also be reduced since only a small number of log blocks are allocated for a page level mapping. However, log blocks eventually need to be erased and this will trigger several merge operations to reclaim them in the future.

**BAST (Block Associative Sector Translation)** [8] scheme classifies blocks into two types: data blocks for data saving and log blocks for overwrite operations. Once an overwrite operation is issued, a clean log block is assigned and the data is written to this clean block instead of directly calling a block erase operation. As a result of this, erase operations do not need to be performed whenever overwrite operation is issued. However, this scheme suffers from low block utilization due to log block thrashing and hot logical block problems [77]. That is, if the log blocks cannot accommodate all updates, BAST should choose a victim log block to be erased. This triggers the expensive merge operations and causes low block utilization.

**FAST (Fully Associative Sector Translation)** [77] is based on BAST scheme but allows log blocks to be shared by all data blocks unlike BAST in which each log block is associated with one data block exclusively. This scheme subdivides log blocks into two types: sequential log blocks for switch operations and random log blocks for merge operations. Even though this scheme accomplishes better utilization of log blocks, it still remains in low block utilization if overwrite operations are repeatedly requested only to the first page of each block. Moreover, random log blocks give rise to the more complicated merge operations due to the fully associative property.

**AFTL (Adaptive Two-Level Flash Translation Layer)** [13] maintains latest recently used mapping information with fine-grained address translation mechanism and the least recently used mapping information is maintained with coarse-grained mechanisms due to the limited source of the fine-grained slots. However, even though there are a large amount of recently accessed data, they cannot all move to fine-grained slots due to the limited and fixed size of fine-grained mechanism. In other words, coarse-to-fine switches
Table 4.1: Various FTL Schemes.

<table>
<thead>
<tr>
<th></th>
<th>Page (Ideal)</th>
<th>CFTL</th>
<th>DFTL</th>
<th>FAST</th>
<th>AFTL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FTL Type</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping Table Location</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exploits Temporal Locality</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exploits Spatial Locality</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Full Merge Needed</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memory Needed (4G Flash)</td>
<td>8MB</td>
<td>50KB</td>
<td>32KB</td>
<td>512KB</td>
<td>400KB</td>
</tr>
<tr>
<td>Adaptiveness</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

incurs corresponding fine-to-coarse switches, which causes overhead in valid data page copies. Additionally, both corresponding coarse-grained slot and its primary block can only be removed if all of the data in its primary block appear in the replacement block, which leads to low block utilization.

*SuperBlock FTL* [79] scheme attempts to exploit the block level spatial locality in workloads by allowing page level mapping in a superblock (a set of consecutive blocks). This separates frequently updated hot data and non-hot data into different blocks within a superblock and consequently the garbage collection efficiency is improved and reduces the number of full merge operations. However, this approach uses a three-level address translation mechanism which leads to multiple accesses of spare areas in data pages. In addition, it also uses a fixed size of superblock explicitly required to be tuned according to workload requirements.

*LAST (Locality-Aware Sector Translation)* [9] scheme adopts multiple sequential log blocks to make use of spatial localities in order to supplement the limitations of FAST. It classifies random log buffers into hot and cold partitions to alleviate full merge costs. LAST, as its authors mentioned, relies on an external locality detector for its classification, which cannot efficiently identify sequential writes when the small-sized write has a sequential locality. Moreover, the fixed size of the sequential log buffer increases the overall garbage collection overhead.

Table 4.1 summarizes the distinguished features of diverse FTL schemes and I will evaluate these five FTLs in the experiment part.
4.1.2.3 Merge Operations

The out-of-place update in flash memory incurs many invalid pages in blocks and finally invokes a garbage collection in a Flash Translation Layer (FTL). The garbage collection ultimately leads to many merge operations to reclaim those invalid pages. Since the merge operation heavily affects the FTL performance, reducing the numbers of merge operations is one of the main concerns to design FTL schemes.

Figure 4.3 illustrates a variety of merge operations. Merge operations can be classified into three types: switch merge, partial merge, and full merge [12]. A switch merge is triggered only when all pages in a block are sequentially updated from the first logical page to the last logical page. An FTL erases the data block filled with invalid pages and switches the log block into the data block (Figure 4.3 (a)). Since this requires only one block erasure, it is the cheapest merge operation. A partial merge is similar to the switch merge except for additional valid page copies. After all the valid pages are copied to the log block, an FTL simply applies the switch merge operation. The partial merge is executed when the updates do not fill one block sequentially (Figure 4.3 (b)). Therefore, this costs additional page copies as well as one block erasure. A full merge requires the largest overhead among three merge operations. An FTL allocates a clean block and copies all the valid pages from either the data block or the log block into the clean block. After all the valid pages are copied, the clean block becomes the data block and both the former data block and the log block are erased (Figure 4.3 (c)). Therefore,
a single full merge operation requires as many read and write operations as the number of valid pages in a block plus two erase operations [79].

4.1.3 CFTL: Convertible Flash Translation Layer

In this section, we describe our proposed Convertible Flash Translation Layer (CFTL, for short). CFTL judiciously takes advantage of both page level and block level mappings in order to overcome the innate limitations of existing FTLs described in Section 4.1.2.2. In addition, CFTL employs a specially designed caching strategy and dynamic partitioning of mapping tables to speed up the address lookup performance by significantly improving a cache hit ratio.

4.1.3.1 Architecture

Figure 4.4 gives an overview of the CFTL design, where C, V, and I stand for Clean, Valid, and Invalid respectively. The complete page mapping table in CFTL is stored in the flash memory. We define it as a tier-2 page mapping table. As the mapping table is stored in the flash memory, first we need to read the page mapping table from flash...
to find the location of the original data during the address lookup. Next, from that location, we can find the data. Clearly, a page table lookup incurs at least one flash read operation. To solve this problem, CFTL caches parts of the mapping table on the SRAM. As shown in Figure 4.4, CFTL maintains two mapping tables in SRAM: CPMT (Cached Page Mapping Table) and CBMT (Cached Block Mapping Table). CPMT is a small amount of a page mapping table that serves as a cache to make the best of temporal and spatial locality in a page level mapping. This table retains an addition to CPMT called a consecutive field. This simple field provides a smart hint in order to effectively improve the hit ratio of CPMT, thereby exploiting spatial locality. This will be explained in more detail in Subsection 4.1.3.4. CBMT is a block mapping table and, like CPMT, serves as a cache in order to exploit both localities in a block level mapping. CBMT translates logical block numbers (LBN) to physical block numbers (PBN), which enables fast direct access to flash data blocks in conjunction with page offsets. The Block mapping table exists only in SRAM unlike the page mapping tables.

In addition to both CBMT and CPMT (as shown in Figure 4.4), there exists another mapping table. We define this SRAM based mapping table as a tier-1 index table. A tier-1 index table keeps track of tier-2 page mapping tables dissipated over the entire flash memory. Unlike those three tables residing in SRAM, tier-2 mapping tables are stored in flash memory due to its large size. Since tier-2 mapping tables compose the whole page mapping table as in a pure page level mapping scheme, each entry in each table directly points to a physical page in flash. Moreover, since tier-2 mapping table resides in flash memory, whenever any mapping information is updated, a new page is assigned and all mapping information in the old page (mapping table) is copied to the new page, while still reflecting the updated mapping information. This arises from a special feature of flash memory in which both read and write operations to flash memory are performed in terms of a page unit. Thus, we need to maintain a tier-1 index table to trace each tier-2 page mapping table which can be scattered over the flash memory whenever it is updated. Each page (i.e., one tier-2 mapping table) can store 512 page mapping entries. For clarification, assuming each page size is 2KB and 4 bytes are required to address the entire flash memory space, then $2^9$ (2KB/4 bytes) logically
consecutive address mapping entries can be saved for each data page. Therefore, a 4GB flash memory device needs only 8MB \((2^{12} \times 2\text{KB})\) of space to store all the required \(2^{12}\) (4GB/1MB per page) number of tier-2 mapping table pages in flash.

### 4.1.3.2 Addressing Mode Switches

In CFTL, when and how it converts its addressing mode are of importance. This subsection describes the hot and cold data classification and the addressing mode change policies in CFTL.

- **Hot and Cold Data Identification:** When any data block is frequently updated, we define it as *hot* data. On the other hand, if it is accessed in a read dominant manner or has not been updated for a long time, we define it as *cold* data. In the CFTL scheme, hot data identification plays an important role in making decisions on mode switches from a block mapping to a page mapping or vice versa. Therefore, we developed a novel hot data identification algorithm by adopting multiple bloom filters and hash functions [26].

For capturing frequency, unlike other existing hot data detection schemes, our proposed scheme does not maintain a specific counter for all LBAs; instead, the number of BF recording the LBA information can exhibit its frequency information so that it consumes a very small amount of memory (8KB) which corresponds to a half of the state-of-the-art scheme [81]. In addition to the frequency, recency information is another important factor to identify hot data. For precise recency capturing, all information of each BF will be periodically erased by turns in order to maintain fine-grained recency information, which corresponds to an aging mechanism. Thus, each BF retains a different recency coverage. We also dynamically assign a different recency weight to each BF: the BF that just erased (i.e., reset BF) has higher recency weight, while the lowest recency weight is assigned to the BF that will be erased in right next turn because this BF has stored LBA access information for the longest period of time. As a result, we can capture fine-grained recency information.

As shown in Figure 3.4, we assume that our scheme adopts a set of \(V\) independent bloom filters (for short, BFs) and \(K\) independent hash functions to capture both frequency and recency, and that each BF consists of \(M\) bits to record \(K\) hash values (we
employed \( V=4 \), \( K=2 \), and \( M=2,048 \) in our scheme). The basic operation is simple: whenever a write request is issued to the Flash Translation Layer (FTL), the corresponding LBA is hashed by the \( K \) hash functions. The output values of each hash function ranges from 1 to \( M \), and each hash value corresponds to a bit position of the \( M \)-bit BF respectively. Thus, \( K \) hash values set the corresponding \( K \) bits in the first BF to 1. When the next write request comes in, our scheme chooses the next BF in a round robin fashion to record its hash values. To check whether the corresponding data are hot or not, we calculate a total hot data index value by combining the frequency value and the recency value whenever write requests are issued. If the value is greater than a predefined hot threshold, they are identified as hot and otherwise, as cold.

Our multiple BF-based hot data identifier achieves more accurate hot data identification as well as less memory consumption. Hot and cold data identification itself, however, is out of scope of this paper. Different algorithms such as the LRU discipline [82] or multihash function scheme [81] can also substitute for our approach.

- **Page to Block Level Mapping:** If a hot/cold data classifier in CFTL identifies some data as cold data, addressing mode of those data is switched to a block level mapping scheme during garbage collection. In particular, when the cold data pages in a logical block are physically dissipated in flash, we need to consecutively collect those pages into a new physical data block. We then pass this block mapping information into CBMT for a block level mapping. Contrastingly, when all valid physical data pages in a logical block are identified as cold data and saved in a consecutive manner, it can be switched to a block level mapping without any extra cost. Note that although cold data, in general, include also such data that have not been updated for a long time, CFTL does not convert such types of cold data blocks to a block level mapping due to its inefficiency. When they are frequently read accessed, FTL converts this type of cold data blocks to a block level mapping. That is, since only a small fraction of the address space for a disk is frequently referenced [81], converting all of the cold data that have not been referenced for a long time to the block level mapping wastes not only a considerable amount of memory space for the mapping table, but also an expensive read/write cost to collect them.
Unlike other hybrid FTLs, CFTL does not retain log blocks for updates. So we need a mechanism to deal with the fragmentation of updated pages when some data blocks are addressed with a block level mapping scheme. In CFTL, if four pages in a block are updated within one decay period, the corresponding block is switched to a page level mapping. This does not require any cost. However, if less than four updates in a block are issued, the updated information is stored to CPMT and the old page is marked as invalid. That is, the block mapping scheme in CFTL does not need to worry about the large fragmentation of update pages because those blocks are reverted to a page level mapping for better performance. Consequently, CFTL can avoid log blocks which are the main cause of the expensive full merges in other hybrid FTLs.

- **Block to Page Level Mapping:** This addressing mode change is a straightforward process. In the case of write dominant access patterns, the hot/cold data classifier in CFTL makes a decision to switch from block to page level mapping. In addition, even though the workload does not show a write intensive access pattern, the corresponding data block is converted to a page level mapping if at least four data pages in the block are updated within its decay period. The latter case enables CFTL to improve address translation efficiency. That is, if there exist many invalid pages in a block due to frequent updates, we cannot take advantage of direct address translation of a block level mapping scheme because additional accesses are required to look up the valid data pages. To reduce this extra overhead and exploit the benefit (i.e., good write performance) of a page level mapping, CFTL manages those kinds of data with a page level mapping. Unlike the mode change from a page to a block level mapping, this mode switch originally does not require any extra costs because a page mapping table is always valid to all data in flash. Therefore, when a hot/cold data classifier in CFTL identifies some data in flash as hot data, CFTL only has to remove the corresponding block mapping entries from CBMT. Then those data can only be accessed by a page mapping table, not by a block mapping table.
4.1.3.3 Address Translation Process

Figure 4.5 describes the CFTL address translation process to serve read or write requests. Basic overall translation processes are as follows: When a read or write request is issued, if its mapping information has been already stored in either CBMT or CPMT, the request can be directly served with the existing mapping information so that CFTL can significantly reduce address translation overheads. Otherwise, CFTL fetches the requested mapping information from flash by using both the tier-1 index table and tier-2 page mapping table.

When the request does not hit either cached mapping table, CFTL requires more address translation processes as follows: If there are cache spaces available in CBMT or CPMT, the fetched mapping information is stored in either cached mapping table accordingly. Otherwise, CFTL needs to evict an entry from those tables to accommodate the newly fetched mapping information. As an entry replacement algorithm, CFTL adopts the Least Frequently Used (LFU) cache algorithm [83]. CFTL employs delayed updates to reduce frequent cache flush overheads; thus, when CFTL chooses a victim, it first checks if the victim has been updated since it was stored in CPMT. If the victim has not been updated yet, it is simply evicted from the cache without any extra operations. Otherwise, CFTL needs to reflect the updated mapping information in CPMT to both the tier-1 index and tier-2 page mapping table in flash. To update the outdated mapping information in flash, CFTL reads the mapping information from the old page (old tier-2 page mapping table), updates the corresponding mapping information, and then writes to a new physical page (new tier-2 page mapping table) (Step 4 and 5 in Figure 4.5). The corresponding tier-1 index table is updated to reflect the new tier-2 page mapping table (Step 6 in Figure 4.5).

Now, the victim is removed from the cache and the request is served by using both tier-1 index table and tier-2 page mapping table (Step 7 in Figure 4.5). Finally, the newly fetched mapping information is stored into the space available in CPMT (Step 8 in Figure 4.5).

As described above, since CFTL employs a two-level address translation scheme, a
Figure 4.5: Address Translation Process of CFTL.

worst-case read latency (i.e., requested mapping information does not exist in the cache, the cached mapping tables are full, and the chosen victim entry has been updated) requires three page reads and one page write. Similarly, a worst-case write latency needs two page reads and two page writes. However, if the request hits the cached mapping table, CFTL can considerably reduce address translation overheads, which is the main motivation of our efficient caching strategies in CFTL.

**Algorithm:** Algorithm 3 describes the address translation process in CFTL. Consider a request is issued from the host. CFTL first checks CBMT and then CPMT next.

First, if the requested mapping information exists in CBMT, the request can be directly served with the corresponding mapping information in CBMT combined with its offset (Lines 3-5 in Algorithm 3). In this case, if the request entails write operations in flash, we update the corresponding mapping information in CPMT (Line 8 in Algorithm 3). Since the block mapping scheme in CFTL does not maintain log blocks, if any update request is issued to the pages managed by CBMT, the corresponding update information
Algorithm 3 CFTL Address Translation

Function ADTrans(Request, RequestLPN)
1: RequestLBN = RequestLPN / The Number of Pages per Block
2: Offset = Request LPN % The Number of Pages per Block
3: if (Request LBN hits CBMT) then
4:   Get PBN from CBMT, and serve the request with the PBN and Offset
5: if (Type of Request == WRITE) then
6:   if (RequestLPN exists in CPMT) then
7:     PPN in CPMT with RequestLPN = new PPN
8:    else
9:      Follow algorithm line 23 to 39
10: end if
11: end if
12: else
13:   // if RequestLBN miss in CBMT, then check CPMT
14:   if (RequestLPN exists in CPMT) then
15:     Get PPN with RequestLPN from CPMT directly and serve the request
16: if (Type of request == WRITE) then
17:   PPN in CPMT with RequestLPN = new PPN
18: end if
19: end if
20: else
21:   // if RequestLPN miss in both CPMT and CBMT
22:   if (CPMT is full) then
23:     Select victim entry using LFU algorithm
24: if (The victim has been updated) then
25:   VPN = Victim LPN / The Number of Entries per Page
26:   Get Mppn with VPN from Tier-1 index table
27: Invalidate Mppn (old Mppn) and assign new one page (new Mppn)
28:   PPN in new Mppn = Victim PPN in CPMT
29:   Mppn in Tier-1 index table with VPN = new Mppn
30: end if
31: Remove the victim entry from CPMT
32: end if
33: VPN = RequestLPN / The Number of Entries per Page
34: Get Mppn with VPN from Tier-1 index table
35: Get PPN with RequestLPN from Tier-2 page mapping table with Mppn
36: Serve the request and store this mapping information (LPN,PPN) to CPMT
37: end if
38: end if

is stored in CPMT. This update process follows the CPMT address translation process. This will be described in more detail next.

Second, if the request does not hit CBMT, CFTL checks CPMT next. If the mapping information is present in CPMT, the request is also served directly with the information. In addition, if the request is related to a write operation, the corresponding entry in CPMT needs to be updated with the new PPN (Physical Page Number) (Lines 13-20 in Algorithm 3).

Third, if the request misses the CPMT as well as the CBMT, CFTL follows its
regular two-tier address translation process. CFTL first checks whether the CPMT entries are full. In the case that CPMT has space available to store mapping information, the new mapping information can take the free slot in CPMT (Lines 35-39 in Algorithm 3). On the other hand, if CPMT mapping entries are full, CFTL chooses a victim entry to evict it from CPMT in order to allow the new entry to be stored. If the victim selected by the LFU algorithm has not been updated since it was stored into CPMT, the entry is simply removed without any extra operations. Otherwise (this is the worst case), we need to update the corresponding tier-2 page mapping table to reflect the new mapping information because CFTL uses delayed update and batching process during the garbage collection time to prevent frequent update of tier-2 page mapping table. At this moment, we can get the PPN storing the corresponding tier-2 page mapping table from the tier-1 index table. After arriving at the corresponding tier-2 mapping table, we copy all mapping information from the old tier-2 mapping table to the newly assigned tier-2 mapping table while reflecting the new mapping information. Then, we must update the corresponding tier-1 index table entry to reflect the newly assigned PPN of the new tier-2 mapping table (Lines 25-31 in Algorithm 3). Now, we are ready to translate the original request issued from the system. The translation process is the same as above. We finally can find the PPN to serve the request in flash and then store this mapping information into CPMT.

4.1.3.4 An Efficient Caching Strategy

All PPNs (Physical Page Numbers) in a data block are consecutive. Our proposed efficient caching strategy in CFTL is inspired by this simple idea. CFTL maintains two types of cached mapping tables in SRAM: Cached Page Mapping Table (CPMT) and Cached Block Mapping Table (CBMT). Although these are address mapping tables, they serve as a cache for a fast address translation. CPMT is employed to speed up the page level address translation, while CBMT is used for fast block level address translation. As shown in Figure 4.3, in addition to the existing logical to physical address mapping fields in CPMT, CFTL adds one more field named a consecutive field for more efficient address translation. This field describes how many PPNs are consecutive from the
corresponding PPN in CPMT. In other words, whenever FTL reaches a tier-2 page mapping table for an address translation, it identifies how many physical data pages are consecutive from the corresponding page. It then records the number of consecutive PPNs to the consecutive field in CPMT at the time it updates CPMT.

Take for example the CPMT in Figure 4.4. LPN = 0 corresponds to PPN = 110. Moreover, the consecutive field hints that 4 numbers of PPN from PPN = 110 are consecutive: 110, 111, 112, and 113. These physical addresses also correspond to 4 respective logical addresses from LPN = 0. That is, LPN = 1 is mapped to PPN = 111, LPN = 2 is correspondent to PPN = 112, and LPN = 3 corresponds to PPN = 113. If any page in the consecutive pages is updated, we need to maintain both sides of consecutive pages because the corresponding pages are not consecutive any more due to the updated page (i.e., the updated page splits the consecutive pages into two). In this case, we divide the corresponding mapping information into two consecutive parts and update each mapping information accordingly. Algorithm 4 describes this algorithm especially regarding CPMT management.

In summary, the consecutive field in CPMT enables CPMT to exploit spatial locality as well as temporal locality. By using this simple field, even though CPMT does not store the requested address mapping information, the consecutive field provides a hint to increase the hit ratio of the cache. This, ultimately, can achieve higher address translation efficiency with the same number of mapping table entries in the cache, resulting in a higher overall performance.

**Algorithm:** Algorithm 4 gives the pseudocode to manage CPMT containing consecutive field. When CFTL gets RequestLPN (Logical Page Number), for each CPMT entry, it first checks if the LPN in CPMT is equivalent to RequestLPN. If the LPN in CPMT is equal to RequestLPN, CFTL can directly serve the requested operation with the corresponding mapping information. Otherwise, CFTL checks consecutive field to find a hint. If the RequestLPN is greater than LPN in CPMT and smaller than LPN in CPMT plus the value in consecutive field (Line 2 in Algorithm 4), RequestLPN can hit the CPMT even though there does not exist exact mapping information in CPMT. CFTL first calculates the difference between RequestLPN and the corresponding LPN.
Function CPMTCheck(RequestLPN)

1: for (Each CPMT entry) do
2:   if (LPN in CPMT < RequestLPN < LPN in CPMT + Consecutive−1) then
3:     Diff = RequestLPN − LPN in CPMT
4:     PPN = PPN in CPMT + Diff
5:     if (Type of Request == WRITE) then
6:       // if one of the consecutive addresses is updated, the entry in CPMT is divided into two
7:       or three entries. Then they will be updated or newly stored into CPMT accordingly
8:       Update Consecutive value of LPN in CPMT = Diff
9:       Store to CPMT with RequestLPN and new PPN
10:      if (Consecutive value of LPN−Diff−1 > 0) then
11:        // if the hitting Request LPN is not the end of consecutive addresses of LPN
12:         Store to CPMT with Request LPN+1, PPN+Diff+1, and Consecutive−Diff−1
13:     end if
14:   end if
15: end for

in CPMT. Then it can finally determine PPN (Physical Page Number) by simply adding
the difference value to PPN in the corresponding CPMT entry (Line 4 in Algorithm
4). If the operation (Type of request) is WRITE, CFTL needs to carefully update the
corresponding mapping information in the CPMT because WRITE operation causes a
page update, which produces an effect on the consecutive value in the corresponding
mapping entry. The update process is as follows. First, CFTL updates the original con-
secutive value to difference value it already calculated. Next, it saves the new mapping
information with RequestLPN and new PPN (Lines 5-10 in Algorithm 4). If the the
RequestLPN is not at the end of the consecutive address of LPN, it needs to save one
more mapping entry with RequestLPN+1 for LPN and PPN+Diff+1 for PPN. Finally
CFTL needs to update the consecutive value to Consecutive−Diff−1.

For clarification, a mapping entry in CPMT assumes the form of (LPN, PPN, Con-
secutive). We additionally assume that RequestLPN is 105 and the mapping entry
corresponds to (100, 200, 10). This request typically does not hit that mapping entry
since 100 (LPN in CPMT) is not equivalent to 105 (RequestLPN). However, CFTL can
make the request hit the table, using its efficient caching strategy as follows. First of all,
CFTL looks up the consecutive field in the corresponding mapping entry and checks if
the RequestLPN lies in between 100 (LPN in CPMT) and 109 (LPN+Consecutive−1).
In this case, the RequestLPN (105) lies between 100 and 109. Then, CFTL can derive
the requested PPN (205) by adding 5 (Diff) to 200 (PPN). However, if this request is a WRITE operation, we separate the one corresponding entry into three mapping entries because the WRITE operation hurts the consecutiveness of PPNs in a data block. In order to manage CPMT, CFTL first updates the consecutive value from 10 to 5 (i.e., (100, 200, 5)), then need to save a new mapping entry. Assuming WRITE operation updates data in a page with PPN 205 to a new page with PPN 500, CFTL then needs to save a new mapping entry (105, 500, 1) to CPMT. Finally, since the RequestLPN (105) is not correspondent to the end of consecutive address of LPN (109), it needs to save one more entry (106, 206, 4) to CPMT. If the RequestLPN is 109, it needs to save only two entries, namely, (100, 200, 9) and (109, 500, 1).

4.1.3.5 Adaptive Partitioning of Mapping Tables

Initially CFTL assigns the same amount of a memory space to both CBMT and CPMT. However, this static assignment can cause a waste of memory. To make the best use of this limited memory space, CFTL dynamically adjusts the sizes of both CBMT and CPMT in SRAM according to workload patterns. That is, as the number of a write request in a workload grows, CFTL assigns more spaces to CPMT from CBMT since only a part of a CBMT space can be appropriated for block level mapping. Especially at the beginning, since CPMT is always filled with mapping information even faster than CBMT, CFTL utilizes this space available in CBMT by assigning the space to CPMT. This can considerably improve initial cache hit ratios. After both tables are full of mapping information, CFTL adaptively tunes the ratio of CBMT and CPMT according to the workload characteristics. When CFTL takes a space from CBMT, it can remove the block mapping information without any extra cost and reuse the space for CPMT. However, when CFTL takes a space with mapping information from CPMT, it should first check if the corresponding mapping information has been updated. Then it follows the cache replacement policy in CPMT described in Subsection 4.1.3.3. Similarly, since read intensive workloads can take advantage of CBMT, CFTL allots more spaces to CBMT from CPMT. This adaptive partitioning of both CBMT and CPMT enables CFTL to find an optimal memory space for them so that CFTL not only efficiently
utilize a limited memory space, but also improves its performance further.

4.1.3.6 Discussions

• **CFTL vs. DFTL:** Although our CFTL was basically inspired by DFTL, it is not just a simple extension of DFTL because both are very different FTL schemes. First of all, CFTL adopts a hybrid mapping, while DFTL is a page level mapping scheme. So, some parts in CFTL are managed by a block level mapping and the others are addressed by a page level mapping. Second, CFTL employs an adaptive address translation mechanism, while DFTL uses a static address mapping. Thus, CFTL adapts its mapping schemes to data access patterns. Third, CFTL offers two efficient mechanisms to fully exploit a small cache: a consecutive field and adaptive partitioning. These make a significant contribution to improve a cache hit ratio so that they can considerably improve address translation efficiency in CFTL. Fourth, CFTL can take advantage of both temporal and spatial localities, while DFTL does not consider spatial localities. Fifth, we newly designed our hot data identification scheme by adopting multiple bloom filters to efficiently classify hot and cold data.

Based on these observations, we discuss the advantages of CFTL compared to other existing FTLs in several aspects.

• **Read Performance:** DFTL [12] shows a good read performance under the condition of high temporal locality. However, under totally random read intensive patterns (i.e., low temporal locality), DFTL inevitably causes many cache misses in SRAM, which degrades its overall read performance compared to the existing hybrid FTLs. CFTL, on the other hand, displays a good read performance even under the low temporal locality since read intensive data are dynamically converted to a block level mapping. Moreover, its efficient caching strategy improves its read performance further by exploiting spatial locality.

• **Temporal and Spatial Localities:** When it comes to data access, both temporal and spatial localities play a very important role in data access performance [48]. DFTL takes temporal locality into consideration by locating a cached mapping table in SRAM, but leaves spatial locality unaccounted for. On the other hand, CFTL, superblock FTL [79]
and LAST take both localities into account.

- Full Merge Operations: Although CFTL is a hybrid FTL scheme, it, unlike other existing hybrid FTLs, is fundamentally rooted in a page level mapping. It therefore eradicates expensive full merge operations like DFTL. However, other existing hybrid FTLs stick to the inborn limitations of a block level mapping because they are originally based on a block level mapping. Even though they make an attempt to adopt a page level mapping scheme, it is restricted to a small amount of log blocks.

- Block Utilization: Existing hybrid FTLs maintain relatively small numbers of log blocks to serve update requests. These ultimately lead to low block utilization. Although there are many unoccupied pages in data blocks, garbage collection can be unnecessarily triggered to reclaim them. On the other hand, DFTL and CFTL resolve this low block utilization problem because updated data can be placed into any of the flash data blocks. CFTL makes full use of the benefits of a page level mapping.

- Write Performance: While hybrid FTLs maintain log blocks, they cannot be free from a poor write performance. Many random write operations inevitably cause many full merge operations, which ultimately results in a poor write performance. A page level mapping, however, can get rid of full merge operations. Although CFTL uses a hybrid approach, it achieves the good write performance of a page level mapping scheme since all data in CFTL is fundamentally managed by a two-tier page level mapping. Thus, both CFTL and DFTL achieve good write performance.

4.1.4 Performance Evaluation

This section provides diverse experiments and results.

4.1.4.1 Evaluation Setup

We simulate a 32GB NAND flash memory with configurations shown in Table 4.2. Our experiments of flash memory are based on the latest product specification of Samsung’s K9XXG08UXM series NAND flash part [25][84].

For an ideal page level mapping, we assume that the whole mapping table can be
Table 4.2: Simulation Configuration

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Read to Register</td>
<td>25µs</td>
</tr>
<tr>
<td>Page Write from Register</td>
<td>200µs</td>
</tr>
<tr>
<td>Block Erase</td>
<td>1.5ms</td>
</tr>
<tr>
<td>Serial Access to Register (Data Bus)</td>
<td>50µs</td>
</tr>
<tr>
<td>Page Size</td>
<td>2KB</td>
</tr>
<tr>
<td>Data Register Size</td>
<td>2KB</td>
</tr>
<tr>
<td>Block Size</td>
<td>128KB</td>
</tr>
<tr>
<td>Entries in Mapping Tables</td>
<td>4,096 entries</td>
</tr>
</tbody>
</table>

Table 4.3: Workload Characteristics

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Total Requests</th>
<th>Request Ratio (Read:Write)</th>
<th>Inter-arrival Time (Avg.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Websearch3</td>
<td>4,261,709</td>
<td>R:4,260,449(99%) W:1,260(1%)</td>
<td>70.093 ms</td>
</tr>
<tr>
<td>Financial1</td>
<td>5,334,987</td>
<td>R:1,235,633(22%) W:4,099,354(78%)</td>
<td>8.194 ms</td>
</tr>
<tr>
<td>Financial2</td>
<td>3,699,194</td>
<td>R:3,046,112(82%) W:653,082(18%)</td>
<td>11.081 ms</td>
</tr>
<tr>
<td>Random_read</td>
<td>3,695,000</td>
<td>R:3,657,822(99%) W:37,170(1%)</td>
<td>11.077 ms</td>
</tr>
<tr>
<td>Random_even</td>
<td>3,695,000</td>
<td>R:1,846,757(50%) W:1,848,244(50%)</td>
<td>11.077 ms</td>
</tr>
<tr>
<td>Random_write</td>
<td>3,695,000</td>
<td>R:370,182(10%) W:3,324,819(90%)</td>
<td>11.077 ms</td>
</tr>
</tbody>
</table>

stored in SRAM. For fair evaluation, we assign the same number of mapping entries (4,096) for the cached mapping tables in SRAM and use the same size of cache memory (16KB) for both tier-1 index table in CFTL and Global Translation Directory (GTD) in DFTL. We additionally assume that the SRAM is sufficient enough to store the address mapping table for both FAST and AFTL, and approximately 3% of entire space is assigned for log blocks in FAST (this is based on [79]).

For a more objective evaluation, various types of workloads including real trace data sets are selected (Table 4.3). Websearch [85] trace made by Storage Performance Council (SPC) [86] shows well a read intensive I/O trace. Although a Websearch trace consists of three trace files (i.e., Websearch1, Websearch2, and Websearch3), each trace
file exhibits almost the same data access patterns (i.e., heavily read intensive) and our experiments displayed near the same results for each. Thus, we employ just one of them (i.e., Websearch3). As a write intensive trace, we employ Financial1 [56] made from an OLTP application running at a financial institution. For the totally random performance measurements, we based random traces upon Financial2 [56] which is also made from an OLTP application. Three types of random trace workloads (read intensive, 50% read and 50% write, and write intensive) are employed for more complete and objective experiments of the purely random access performance. An interesting point is that Gupta et al. [12] classified Financial2 trace as a write dominant trace, but ours clearly shows read dominant data access pattern (refer to Table 4.3). So, we employ Financial2 trace as read intensive I/O trace.

4.1.4.2 Performance Metrics

An average response time is a good measure of the overall FTL performance estimation in the sense that it reflects the overhead of a garbage collection and address translation time as well as system service time. Since, as mentioned in Section 3.2.1, we focus mainly on efficient address translation and block assignment, our experiments also concentrate on comparing relative address translation performance of different FTL schemes. We also make an attempt to evaluate cache hit ratios in order to verify the effectiveness of our proposed efficient caching strategy and adaptive partitioning scheme. Block erase count is also another essential measure of evaluating FTL performance. An ideal pure page level mapping is selected as our baseline scheme.

4.1.4.3 Results and Analysis

We illustrate our simulation results with a variety of plots to demonstrate that CFTL outperforms existing FTL schemes in terms of overall read and write performance under various workload conditions.

1) Overall Performance: As shown in Figure 4.6, overall performance of CFTL is very close to that of an ideal page level mapping. That is, CFTL outperforms other FTLs
in terms of both read and write performances under realistic workloads as well as random workloads since read intensive workload reflects well the read performance and write intensive workload the write performance. CFTL shows its better read performance against FAST [6] which has a strong point in read performance. It also exhibits a better write performance over DFTL [12] which has an excellent write performance.

Figure 4.7 depicts the overall performance over time for each scheme, and enables us to analyze these results in more detail. Under the read intensive access patterns (Figure 4.6(a) and 4.7(a), Figure 4.6(b) and 4.7(b), and Figure 4.6(d) and 4.7(d)), CFTL switches more and more data to a block level mapping over time in order to make
the best of its fast direct address translation. Moreover, the efficient caching strategy
has a considerable effect on read performance since almost all of the consecutive data
pages in a data block are not dispersed under the environment of the read intensive
access patterns. In particular, as shown in Figure 4.6(d), our proposed caching strategy
exhibits its noticeable effectiveness in read performance, especially under randomly read
intensive workloads. Compared to Figure 4.6(a), the random read performance of most
of the FTLs is significantly degraded. However, CFTL still maintains good random read
performance due to its efficient caching strategy. We will discuss this in more detail in
the efficient caching strategy part.

As we expected, FAST, which is a hybrid mapping scheme primarily based on a
block level mapping, also shows a good performance under the read intensive workloads.
However, FAST does not reach the performance of CFTL because even though there are
not as many update operations in this workload, they still affect the overall performance
of FAST. In addition to this expensive merge operation in FAST, the extra read cost in
log blocks is another factor that deteriorates its read performance. DFTL is a two-tier
page level mapping. If, however, the workload does not contain a high temporal locality,
it frequently requires an additional address translation overhead. This is the main reason
that DFTL does not exhibit relatively good random read performance (Figure 4.6(d)
and 4.7(d)). Unlike these other schemes, AFTL[13] does not show a good overall
performance in read intensive workload because when the fine-grained slots are full
of mapping information, it starts to cause valid data page copies for a coarse-grained
mapping. This brings about overhead that degrades its overall performance.

As the number of write requests in workloads grows, performance variations among
each scheme also increase rapidly since frequent write operations cause frequent updates
in flash. As shown in Figure 4.6(c) and 4.7(c), Figure 4.6(e) and 4.7(e), and Figure 4.6(f)
and 4.7(f), the overall performances of AFTL and FAST are severely degraded under
the 50% read and 50% write workload, not to mention write intensive workloads. Thus,
we eliminate both schemes in Figure 4.7(c) and 4.7(f). This fundamentally stems from
frequent erase operations in both schemes. Frequent updates cause frequent merge op-
erations in FAST and frequent erase to both primary and replacement blocks in AFTL.
However, since DFTL is a two-tier page level mapping and CFTL is fundamentally based on a page level mapping, both schemes shows a considerably better overall performance than AFTL and FAST. This is particularly evident under write dominant access patterns (Figure 4.6(c) and 4.7(c) and Figure 4.6(f) and 4.7(f)). However, CFTL shows better overall write performance than DFTL due to its faster address translation resulting from our efficient caching strategy. Note that CFTL requires no extra cost to switch from a block level mapping to a page level mapping and does not maintain log blocks under a block level mapping. We do not exhibit both FAST and AFTL both in
the Figure 4.7(c) and Figure 4.7(f) since their performances are significantly degraded compared to the others.

In summary, these simulation results strongly support our assertion that CFTL is adaptive to data access patterns in order to fully exploit the benefits of page level and block level mapping. Consequently, it achieves a good read and write performance under read intensive, write intensive, and totally random access patterns.

2) An Efficient Caching Strategy: In order to verify the effectiveness of our proposed caching strategy, we prepare two different CFTL schemes: CFTL and CFTL_WC (CFTL_WC stands for the CFTL without our efficient caching strategy). CFTL originally contains our proposed caching strategy by adding a consecutive field to CPMT. On the other hand, CFTL_WC does not contain this field in CPMT. For fair evaluation, both schemes have the same parameters except for the caching strategy.

As shown in Figure 4.8, our proposed caching strategy is effective to improve overall performance of CFTL further. Due to its simple consecutive field, we remarkably

Figure 4.8: Performance Improvement with the Efficient Caching Strategy
improve the hit ratio of CPMT even though the requests do not hit the cache (Shown in Figure 4.8(c)). This yields a faster address translation and ultimately yields great influence on the enhancement of overall performance. Figure 4.8(c) supports our assertion on the effectiveness of our proposed caching strategy. It illustrates the request hit ratios in CPMT under diverse workloads. As shown in the Figure 4.8(c), CFTL exhibits its dominant request hit ratio against CFTLWC especially under read intensive workloads (Websearch3, Financial2, and Random_read) since read operations do not hurt the consecutiveness of data pages in a block. In particular, CFTL surprisingly improves cache hit ratio by a factor of almost eight for the randomly read intensive workload (i.e., Random_read), compared to CFTLWC. These results demonstrate that our proposed caching strategy exhibits its excellent performance for the read intensive workloads, and much better performance for randomly read intensive workloads.

On the other hand, as write requests grow like Random_even (50% Read : 50% Write), Financial1 (23% Read : 77% Write), and Random_write (10% Read : 90% Write), the frequent updates break the consecutiveness of data pages in a block so that the variation of hit ratios between CFTL and CFTLWC is relatively reduced. Even though the gaps are lessened, CFTL still shows higher hit ratios than CFTLWC with the help of its consecutive field.

Figure 4.8(a) illustrates that over time, both schemes converge on the same average response time. In other words, even though the CFTLWC does not have the efficient caching strategy, its read performance gradually approaches that of CFTL as more and more read intensive requests come in. In our experiment, both read performances converge on the same performance after approximately 500 seconds corresponding to 104,420 intensive read requests in this workload (Websearch3). Note that the convergence time totally depends on the types of workloads. In other words, most read intensive data switch to a block level mapping so that the read performance of CFTLWC gradually approaches that of CFTL after some time. This proves the effectiveness of the adaptive feature of CFTL. However, CFTL exhibits its better read performance from the beginning, even before the read intensive data are converted to a block level mapping.
In summary, our proposed efficient caching strategy remarkably improves the cache hit ratio (on average, by 245%) even with the same number of mapping entries in SRAM so that it boosts the overall performance of CFTL further. These experimental results prove the fact that even though the request mapping information does not exist in the cache, CFTL can help make it hit the cache by using simple hints provided by a consecutive field in CPMT.

3) Adaptive Partitioning of Mapping Tables: Our adaptive partition scheme adaptively tunes the ratio of CBMT and CPMT. To demonstrate its effectiveness, we again prepare two different CFTL schemes: CFTL and CFTL_WA (similarly, CFTL_WA stands for the CFTL without our adaptive partitioning scheme). CFTL retains our proposed dynamic partitioning scheme as well as aforementioned efficient caching strategy, while CFTL_WA does not contain the dynamic partitioning scheme but includes the efficient caching strategy.

As exhibited in Figure 4.9, both CFTL and CFTL_WA start identical read and write
performance at the beginning since both CBMT and CPMT are not full of mapping information yet. However, as time goes on, CFTL shows a better performance due to the adaptive partitioning scheme. These performance gaps ultimately result from total cache hit ratios of both CBMT and CPMT, and Figure 4.9 (c) adds support to this claim.

4) The Number of Block Erases: Block erase operations significantly degrade overall performance of flash memory since erase operations are almost 10 times slower than write operations and over 100 times slower than read operations [25]. Thus, the number of block erases is a good measure of the performance analysis of diverse FTL schemes. Figure 4.10 shows the number of block erases for each FTL under various workloads. Since excessively read dominant workloads (i.e., Websearch3 and Random_read) cause a significantly smaller number of block erases than the others, we do not employ both workloads. As shown in Figure 4.10, both CFTL and DFTL do not give rise to as
many block erases as FAST and AFTL because DFTL uses page level mapping and CFTL is fundamentally rooted in page level mapping. These experimental results explain well why both CFTL and DFTL show a good write performance compared to other hybrid FTLs. However, CFTL causes a slightly less number of block erases than DFTL due to its efficient caching scheme. Assuming the Cached Mapping Table in DFTL is already filled with address mapping information, whenever a write request is issued, one mapping entry must be evicted from the cache unless the request hit the cache. This evicted entry brings about a new write operation of one page in order to update page mapping table. On the other hand, CFTL maintains consecutive field to improve its cache hit ratio. Although, compared to read operation, this field does not have a considerable influence on cache hit ratio in write operations due to a breakdown of the address consecutiveness, it does help to increase cache hit ratio, which results in a less number of block erases than DFTL. Moreover, this gives a clear explanation of the reason there are no big differences between the numbers of block erases of both CFTL and DFTL especially under totally random access workloads (i.e., Random_even and Random_write).

FAST generates a significantly large number of block erases because of many merge operations. When FAST used up all log blocks in data updates, it must perform merge operations which can cause a lot of block erases. Similarly, assuming AFTL used up all fine-grained slots in storing recently used data, whenever write requests are issued, it forces the corresponding number of entries evicted from the fine-grained slots to move.
to a coarse-grained slots, which causes a lot of valid page copies and block erases in the long run.

Figure 4.10 also gives support to the experimental results of overall performance in Figure 4.6. Figure 4.6 shows that as the proportion of write requests to read requests in workloads grows, the performance variations among each FTL increase rapidly. This, as shown in Figure 4.10, results from the variations of the number of block erases for each scheme.

5) Impact of SRAM Size: Evaluating performance of both CFTL and DFTL with the same size of SRAM is also very meaningful. Figure 4.11 displays the normalized average response time in CFTL and DFTL with variable SRAM size. Since each value is normalized in terms of the ideal page level mapping, the value 1 means the performance of each scheme is comparable to that of the page mapping. Thus, adding more SRAM beyond the corresponding SRAM size for the faster address translation does not provide performance benefit to each scheme. As shown in Figure 4.11, CFTL exhibits better performance than DFTL with the same SRAM under each workload. More read workloads enable CFTL to benefit from block level mapping so that it can reduce memory consumption. On the other hand, since a write intensive workload breaks the consecutiveness of data pages in a block, CFTL converts its mapping scheme to page level mapping. Therefore, both CFTL and DFTL consume almost comparable memory in the long run.

6) Memory Space Consumption: Most of the FTLs maintain their own address mapping information in SRAM for a faster address translation. Since SRAM size is very limited in flash memory, memory space requirements are also another monumental factor among diverse FTL schemes. For simplification, we assume the entire flash size is 4GB and each mapping table in the SRAM consists of 2,048 mapping entries. We also assume that approximately 3% of the entire space is allocated for log blocks in hybrid FTLs (this is based on [79]).

An ideal page level mapping requires 8MB to store its complete page mapping table. This is an exceptionally large space compared to the other schemes. Both AFTL [13] and
FAST [6] also consume a lot of memory space (400KB and 512KB respectively). Almost over 90% of total memory requirements in AFTL are assigned for coarse-grained slots and most of them in FAST are allocated for page level mapping tables. On the contrary, CFTL and DFTL [12] consume only about 10% of the total memory space (50KB and 32KB respectively) in FAST and AFTL since a complete page mapping table is stored in flash memory and not in SRAM. CFTL requires a little more memory (18KB) than DFTL since it adds a *consecutive field* to CPMT for more efficient caching along with maintaining one additional mapping table, called CBMT, for a block mapping that does not exist in DFTL. In addition, both CFTL and DFTL consume the same size of cache memory for the both tier-1 index table in CFTL and GTD in DFTL. However, this extra small amount of memory empowers CFTL to take advantage of a page level mapping and a block level mapping, with which CFTL achieves a good read and write performance. In fact, we did not consider the memory space consumption for a hot data identification scheme since each FTL scheme does not clarify their hot data identification algorithms. As mentioned before, CFTL adopts bloom filters to significantly reduce memory space consumption as well as computation overheads (CFTL consumes only 8KB for the hot data identification scheme). Typical FTL schemes maintain block access counters for hot data identification, which, in general, requires much more memory spaces.
4.1.5 Conclusion

We proposed a novel hybrid FTL scheme named Convertible Flash Translation Layer (CFTL) for flash-based storage devices. CFTL can dynamically convert its mapping scheme, either to a page level mapping or a block level mapping, in accordance with data access patterns in order to fully exploit the benefits of both schemes. Since CFTL, unlike other existing hybrid FTLs, is originally rooted in a page level mapping, it overcomes the innate limitations of them. CFTL stores the entire mapping table in the flash memory. Thus, there is an overhead to look up values in the mapping table. To remedy this problem, we also proposed an efficient caching strategy and adaptive partitioning of mapping tables on the cache.

Our experimental results show that for the realistic read intensive workloads, CFTL outperforms DFTL (which is the state-of-the-art FTL scheme) [12] by up to 24%, for a random read intensive workload outperforms by up to 47%, and for the realistic write intensive workloads outperforms DFTL by up to 4%. Our experiments also demonstrate the new caching strategy significantly improves cache hit ratio, by an average of 245%, and by up to a factor of almost eight, especially for the randomly read intensive workload.

4.2 Application 2: A Read Cache Design for Data Deduplication System

Data deduplication (for short, dedupe) is a special data compression technique and has been widely adopted especially in backup storage systems with the primary aims of backup time saving as well as storage saving. Thus, most of the traditional dedupe research has focused more on the write performance improvement during the dedupe process while very little effort has been made at read performance. However, the read performance in dedupe backup storage is also a crucial issue when it comes to the storage recovery from a system crash. In this paper, we newly design a read cache in dedupe storage for a backup application to improve read performance by taking advantage of
its special characteristic: the read sequence is the same as the write sequence. Thus, for better cache utilization, we can evict the data containers with smallest future references from the cache by looking ahead their future references in a moving window. Moreover, to achieve better read cache performance, our design maintains a small log buffer to judiciously maintain future access data chunks. Our experiments with real world workloads demonstrates that our proposed read cache scheme makes a big contribution to read performance improvement.

4.2.1 Introduction

Digital data explosion empowers data deduplication (for short, dedupe) to have been in the spotlight and over 80% of companies are drawing their attention to dedupe technologies [87]. Data dedupe is a specialized technique to eliminate duplicated data so that it retains only one unique data on storage and replaces redundant data with a pointer to the unique data afterwards. These days, dedupe technologies have been widely deployed particularly in secondary storage systems for data backup or archive due to considerable cost (i.e., time as well as space) saving. Thus, major concerns have been mostly related to write performance improvement thereby efficiently detecting and removing as many duplicates as possible with the help of efficient data chunking, index optimization/caching, compression, and data container design [36, 37, 23, 22, 88]. On the other hand, its read performance has not attracted considerable attention to researchers because read operations are rarely invoked in such dedupe storage systems. However, when it comes to system recovery from a crash, it has a significantly different story. Long term digital preservation (LTDP) communities were recently very emphatic on the importance of read performance in dedupe storage [89, 90]. Moreover, some primary storage systems have started to equip the dedupe technologies [91]. Although this read performance as well as write performance is also a crucial factor in dedupe storage, very little effort has been made at this issue. Typical data chunk (generally, a few KB) read processes in secondary dedupe storage are as follows: first, the dedupe storage system identifies the data container ID retaining the corresponding data chunks to be read. Then, it looks up the container (generally, 2 or 4MB) in the read cache.
Once hitting the cache, it reads the chunks from the cache. Otherwise, it fetches one whole container from the underlying storage and then it can read the corresponding data chunks in the container. However, these read processes result in low cache utilization because even though there exist spatial locality in the data container, only partial data chunks in the containers are mostly accessed [87]. Furthermore, the higher dedupe rates, the higher data fragmentation rates. This can lower spatial locality so that it worsens cache utilization. Our key idea lies in exploiting future access patterns of data chunks. In general, read sequences are identical to write sequences in the dedupe storage for backup. Inspired by this special characteristic inherent in such an application, our read cache design can take advantage of future read access patterns during dedupe processes, but general cache algorithms such as LRU do not consider this special feature in dedupe mechanisms.

Based on these observations, we propose a lookahead read cache design in dedupe storage for a backup application. In this design, we make the following main contributions:

- **Exploiting Future Accesses:** We maintain access information for future read references during dedupe (i.e., write) processes. Thus, the proposed design evicts a victim with a smallest future reference count from the read cache.

- **Design Extension with a Log Buffer:** We assign a portion of a read cache space into a log buffer which can effectively maintain future access chunks on the basis of my hot data identification scheme.

- **Extensive Dataset Analysis:** The proposed design is fundamentally inspired by our diverse real dataset analyses.

Since the proposed design is a read cache scheme, unlike a selective duplication/deduplication approach that allows partial data duplication to improve read performance while hurting its write performance, this design not only does not hurt write performance at all, but also can be applied to other dedupe systems.
4.2.2 Background and Related Work

There have been several technologies to save a storage space by reducing the amount of data. This subsection, first, explores those data reduction technologies and then discusses a read performance issue in data deduplication systems.

4.2.2.1 Data Reduction

As more and more data are generated, various data reduction techniques as well as the efficient and reliable data storage systems are of importance because there is a substantial amount of data from a variety of data sources that remains the same between two consecutive backups and sometimes there are several copies of the same data in backup data storage. Similarly, in primary storage systems, different users of the same application may be able to store the same file such as an email message or an operating system related files [92]. Thus, data reduction via simply deleting unwanted or unneeded data is the most effective way to reduce a company’s data footprint and, in turn, its energy needs. To reduce the amount of such duplicated or redundant information that is stored in storage, diverse approaches have been proposed: data deduplication (file-level or block-level deduplication, and delta block optimization), data compression, and thin provisioning.

Deduplication technology shrinks the data footprint by removing redundant data at the file or block-level. It’s most common use is with backups and archives, but it has been increasingly adopted in primary storage as well [57]. File-level deduplication eliminates multiple copies of the same file, where the duplicate files are replaced with a pointer to the unique version of the file. However, this technique is not effective at removing files that have minor changes compared with previous versions of the file. On the other hand, block-level deduplication removes redundant or duplicate data by maintaining just one unique block or chunk of data, where the block size can be fixed or variable. Similarly, the redundant blocks are replaced with a pointer to the block of a unique data. This block-level deduplication can be even more efficient as identical data is stored multiple times on the storage devices. A major downside to both deduplication
techniques lies in their asymmetrical read and write performance. In general, as more duplicates are eliminated from incoming data stream, the read performance is in marked contrast to its good write performance due to the higher likelihood of the shared data fragmentation [87]. This is the fundamental challenging issue of the tradeoff between read performance and write performance in data deduplication storage. Delta block optimization another technique that is included in the data deduplication technology. Delta block optimization is also designed to reduce the amount of data backed up from the source and the amount of data stored. When the most recent version of a file that has already been backed up is backed up again, the software attempts to figure out which blocks are new. Then it writes only these blocks to backup and ignores the blocks in the file that have not changed. This technique has a similar shortcoming to file deduplication and compression in that if two users sitting in the same office or two separate servers have identical copies of the same file or identical blocks, then delta block optimization will create two identical backups instead of storing just one.

Data deduplication eliminates redundant data, while data compression reduces the size of every piece of data based on well-known algorithms by eliminating redundant bits. Compression can be done standalone or in conjunction with data deduplication. Data compression techniques are essentially transparent to applications and storage hardware. Depending on data types, this compression can achieve significant reductions in data. However, it cannot detect or reduce multiple copies of the same files.

Thin provisioning is a technique of optimizing the efficiency with which the available space is utilized in storage area networks (SAN). It operates by allocating disk storage space in a flexible manner among multiple users, based on the minimum space required by each user at any given time [93]. If applications run out of storage space, they crash. Storage administrators, therefore, commonly install more storage space than required to avoid any potential application failures. This practice provides a headroom for future growth and reduces the risk of application failures. However, it requires the installation of more physical disk capacity than is actually used, which causes a waste of the storage space. To get over this, thin provisioning software allows higher storage utilization by eliminating the need to install physical disk capacity that goes
unused. In most implementations, thin provisioning provides storage to applications from a common pool of storage on an as required basis. Thin provisioning works in combination with storage virtualization, which is essentially a prerequisite to effectively utilize the technology. With thin provisioning, a storage administrator allocates logical storage to an application as usual, but the system releases physical capacity only when it is required. When utilization of that storage approaches a predetermined threshold (e.g. 90%), the array automatically provides capacity from a virtual storage pool which expands a volume without involving the storage administrator. The volume can be over allocated as usual, so the application thinks it has plenty of storage, but now there is no stranded storage. Thin provisioning is on-demand storage that essentially eliminates allocated but unused capacity [93].

4.2.2.2 Read Performance in Data Deduplication

The read performance (mainly throughput) of dedupe storage, such as backup systems, has not been spotlighted since it was widely adopted that the occurrence of reads is much lesser than that of writes in such systems. However, the read performance becomes extremely critical when it comes to restoring the entire system from crash [22, 94]. Higher read performance can significantly save the recovery time while accomplishing higher system availability. Thus, a system may demand guaranteed read performance of dedupe storage in order to satisfy a target system recovery time from its crash, i.e., a target system availability level. In addition, since the dedupe storage has limited capacity, it occasionally needs to stage the backed data streams stored in the underlying storage to the archive storage such as a virtual tape. This requires reconstruction of their original data streams because the archive storage operations are typically stream-based. In fact, this staging frequency is remarkably higher than the user-triggered data retrieval frequency.

Recently, long-term digital preservation (LTDP) communities [89, 90] also have placed great emphasis on the importance of read performance in dedupe storage. One of LTDP requirements is that the repository needs effective mechanisms to detect bit
corruption or loss \[89\]. Some of primary storage has started to equip the dedupe feature \[91\], where reads are definitely as much important as writes. One example is to store images of virtual machines on shared network storage. In \[95\], virtual machine images of idle desktops are migrated onto network storage to save energy by completely shutting down the idle desktops.

As duplicate data (shared chunks) increases, read performance of data dedupe storage to retrieve a data stream gets worse in general because reading a data stream needs to retrieve both unique and shared chunks, whereas the writes store only the unique chunks. Read performance drops because shared chunks are likely to have been physically dispersed over different containers in the underlying storage called chunk fragmentation. In Figure 2.2, the chunk $b_1$ is deduplicated with the chunk $a_0$ away from the chunk $b_0$. Figure 4.13 illustrates another example of the chunk fragmentation with a data stream having 8 incoming chunks. Assume that the chunks marked with 10, 11, 12, and 13 are unique chunks. On the contrary, the chunks marked with 200, 201, 300, and 400 are shared chunks duplicated with the chunks stored in containers 20, 30, and 7, respectively. We also simply assume that each container has four chunks. If none of the eight incoming chunks are shared with the already existing chunks in the in-storage containers, the data stream read requires to retrieve only two containers from the storage. In this example, however, the data stream read needs to access four different containers. Obviously, this would degrade read performance considerably. In practice, the actual degradation becomes even worse because much smaller portion of the container would be accessed.

Zhu et al. in \[22\] put an emphasis on importance of read performance in dedup storage, in particular, for data recovery. They also addressed that read performance substantially decreased during dedup process. Koller et al. \[96\] proposed a selective duplication scheme (named I/O Deduplication) to increase the read/write performance by reducing a disk head movement. They suggested a content-based cache design for both read and write requests as a part of their dedup scheme, but it was implemented in a separate cache area from a virtual file system cache layer and just adopted the existing cache replacement algorithms such as LRU or ARC. Recently, Srinivasan et al. \[87\] proposed
primary inline dedup system design (named iDedup) taking the read performance issue into consideration. iDedup tried to exploit both spatial locality by selectively deduplicating primary data and temporal locality by maintaining dedup metadata completely in memory, not on disk. However, it did not address the file system’s read cache design. Nam et al. in [94] originally introduced an indicator for the degraded read performance named chunk fragmentation level (CFL) and observed a strong correlation between the CFL value and the read performance under backup datasets. However, their CFL has a limitation if a data stream contains many identical (shared) chunks in the same version itself. That is, by definition of their CFL, since the CCF becomes smaller than the OCF, it makes the CFL higher than one. However, its read performance becomes much worse than the sequential read performance. This problem is mainly caused by not taking into account the read cache effect.

4.2.3 A Lookahead Read Cache Design

4.2.3.1 Rationale and Assumptions

In general, as more duplicates are eliminated from incoming data stream, the read performance stands in marked contrast to its good write performance due to the higher likelihood of the shared data fragmentation [87]. This is the fundamental challenging issue of the trade-off between read performance and write performance in dedupe
storage. To address this read performance issue, I propose a novel read cache design leveraging future access information. In dedupe storage for backup or archive, the read access sequence is highly likely to be identical to its write sequence. Based on this key observation, the proposed scheme records write access metadata information for the future read access during each dedupe process, which enables my lookahead cache to exploit future read references. We assume that each data chunk size is variable and a data container retaining many (generally, 200-300) data chunks is a basic unit for reads.

4.2.3.2 Dataset Analysis

Admittedly, the more duplicates in backup data, the higher likelihood of data fragmentation [87]. Table 4.4 shows various dedupe gain ratio (DGR) in successive versions of each backup dataset, where DGR represents the ratio of a data saving size to an original data size. I made an extensive analysis of six real backup datasets and also observed that a considerable portion of data is, in general, duplicated for each version of backup datasets (Table 4.4). This implies the dedupe read cache can be poorly utilized in general cache design because only a small number of data chunks in a data container are accessed. Figure 4.14 exhibits the distributions of the number of accessed data container with respect to the percentage of accessed chunks in the shared container for each real backup dataset. That is, I observed how many data chunks in shared containers are accessed when duplicate data chunks are requested to read. When we reach a percentage of five (note: logarithmic scale for X-axis), it can accommodate most of the number of data container. This implies most of the data containers are only accessed of less than 5% of data chunks in each container. Therefore, this scheme adopts 5% as its initial hot threshold value.

Figure 4.15(a) and (b) show container access patterns of the dedupe backup datasets. The X-axis represents a chunk sequence ID of successive versions of each backup dataset. The Y-axis represents accessed container IDs storing chunks indexed by the chunk sequence ID in the X-axis and the container ID starts from zero. For example, if a dataset includes many unique chunks, the accessed container ID increases linearly like the first version (ver-1) of the dataset ds-1: there are no fluctuations within the ver-1 section in
Table 4.4: Various dedupe gain ratio (DGR) of each backup dataset (Unit:%).

<table>
<thead>
<tr>
<th></th>
<th>ver-1</th>
<th>ver-2</th>
<th>ver-3</th>
<th>ver-4</th>
<th>ver-5</th>
<th>avg. DGR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ds-1</td>
<td>99.9</td>
<td>3.5</td>
<td>6.9</td>
<td>5.6</td>
<td>31.2</td>
<td>29</td>
</tr>
<tr>
<td>ds-2</td>
<td>100</td>
<td>28</td>
<td>24.7</td>
<td>14.9</td>
<td>20.6</td>
<td>37</td>
</tr>
<tr>
<td>ds-3</td>
<td>99.6</td>
<td>95.2</td>
<td>97.7</td>
<td>97.3</td>
<td>96.6</td>
<td>97</td>
</tr>
<tr>
<td>ds-4</td>
<td>90.5</td>
<td>55.4</td>
<td>63.6</td>
<td>20.8</td>
<td>20.6</td>
<td>50</td>
</tr>
<tr>
<td>ds-5</td>
<td>84.1</td>
<td>3.3</td>
<td>2.5</td>
<td>11.9</td>
<td>2.6</td>
<td>20</td>
</tr>
<tr>
<td>ds-6</td>
<td>54.4</td>
<td>22.4</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>38</td>
</tr>
</tbody>
</table>

Figure 4.14: Distributions of The Number of Accessed Container for 6 Real Backup Datasets

Figure 4.15(a), whereas if a dataset contains many duplicates, the chart fluctuates due to fragmented accesses of previous containers. Four datasets (ds-1 through ds-4) show similar data access patterns: most of the data chunks are unique in the initial backup version (ver-1) and the duplicate chunks increase for each successive backup dataset (Figure 4.15(a)). In fact, this is a typical characteristic of most backup datasets (due to space limit, I show only one figure (ds-1)). On the other hand, the other two datasets (ds-5 and ds-6) exhibit different access patterns: there are many duplicates even in the initial version of backup datasets. Thus, we can see many vertical lines even in version 1 as well as successive versions of backup datasets (Figure 4.15(b)).

4.2.3.3 Architectural Overview

Figure 4.16 shows an overview of our proposed base read cache design for dedupe storage. We assume each data chunk with same color belongs to the same data container. The
proposed scheme consists of three key components: a future sliding window, hot data identification scheme, and the read cache. The sliding window can be thought of as a lookahead window to see future chunk read accesses. The data chunk in its tail position is always considered a current data chunk to read and after reading the current chunk, the window takes a slide toward the future by one for each time. Based on this sliding window scheme, the hot or cold decision is made. Our hot data identification scheme maintains data container reference counters. When any data chunk comes into the head position in the window, its reference counter of the corresponding data container is incremented by 1 (note: a data container is a basic unit of the chunk read). On the contrary, if any data chunk is evicted from the tail position in the window, the reference counter is decreased by 1. If the total container reference count of a current data chunk is over than a predefined threshold value (aforementioned 5%) within the window, the corresponding container is classified into a hot container, otherwise, a cold container. This hot threshold value was empirically derived in our workload analysis. More detailed architecture of this hot data identification scheme is described in the following subsection 4.2.3.4. Lastly, our read cache stores the accessed containers and does not adopt the existing cache algorithms such as LRU or ARC. Instead, our read cache selects the least effective (i.e., smallest reference count) container as a victim. I will extend this read cache design by fitting a small log buffer into this base design to
4.2.3.4 Hot Data Identification Scheme

Hot data identification plays an important role in our cache design. Since it is invoked at each time a data chunk is accessed, it must achieve low computational overheads and small memory consumption \[97\]. To meet these requirements, I adopt counting bloom filter and multiple hash functions as displayed in Figure 4.17. That is, the aforementioned container reference counters are implemented by the counting bloom filter. This hot data identification scheme works as follows: whenever a data chunk comes in the sliding window to the head position, the chunk ID is converted into its container ID and then the container ID is fed to hash functions (we adopt multiple hash functions to reduce a false identification rate). Each hash value corresponds to their bit positions in the bloom filter. Finally, each reference counter is increased by 1 respectively. Similarly, outgoing data chunk from the window decreases each counter by 1 accordingly. Based on these basic operations, for identification of a current access chunk, its container ID is fed to hash functions and our scheme, then, checks its corresponding reference counters to improve performance further.
in each bit position of the bloom filter. Due to the likelihood of a hash collision, the scheme always chooses a smallest reference count. If it is greater than a predefined threshold, the data container is classified as hot, otherwise, cold.

### 4.2.3.5 Base Design

The proposed read cache is mainly designed for the secondary dedupe storage for the backup or archive and makes an attempt to exploit its aforementioned special characteristic. In addition, its basic cache policy is managed by our hot data identification scheme. I initially assign a small (8MB=2MB container×4) read cache since we may consider multiple streams/processes environments. Overall working process is described as follows: whenever any data chunk is accessed for read, its container ID is first identified and then the container is ready to be stored in the cache. If the read cache is full of data containers, we need to choose a victim. The proposed scheme selects a container with a smallest future reference count as a victim. However, all data containers in the cache may turn to a different classification over time because the window continues to move forward. Thus, we need to update their reference counts accordingly, which will cause ignorable overheads because there are only a few numbers (basically, 4) of
containers in the cache. Even though this algorithm is simple, it considerably improves dedupe read performance and outperforms the widely adopted cache algorithm, LRU. Unlike our extended design, our base design utilizes our hot data identification scheme mainly for its efficient reference count management, not for hot data identification.

4.2.3.6 Design Extension

This extended read cache design is inspired by my observations on dataset analysis. As shown in Figure 4.18, I assign a part of a read cache space into a log buffer to exploit future access information. That is, the key idea of this extended design is to maintain a small buffer to log future access chunks before a container is evicted from the read cache in accordance with the hot data identification. This log buffer is managed by a circular queue. Before eviction, the extended design attempts to classify the victim container into hot or cold by using our hot data identification scheme. If the victim container is identified as cold (that is, a small number of chunks in the container will be accessed in the near future), its remaining future access chunks within the window is stored into the log buffer. We do not store already accessed data chunks. Since the scheme employs 5% as its hot threshold value, the maximum amount of data chunks to be logged is at most 5% of the total data chunks (typically 10-20 chunks) in the victim container. However, in most cases, the amount of logging chunks will be less than the threshold because the
Table 4.5: Characteristics of the six backup datasets.

<table>
<thead>
<tr>
<th>dataset</th>
<th># of chunks</th>
<th>avg. DGR</th>
<th>total size</th>
<th># of ver.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ds-1</td>
<td>1,970,836</td>
<td>0.29</td>
<td>20GB</td>
<td>5</td>
</tr>
<tr>
<td>ds-2</td>
<td>1,960,555</td>
<td>0.37</td>
<td>20GB</td>
<td>5</td>
</tr>
<tr>
<td>ds-3</td>
<td>1,951,271</td>
<td>0.97</td>
<td>20GB</td>
<td>5</td>
</tr>
<tr>
<td>ds-4</td>
<td>1,892,285</td>
<td>0.50</td>
<td>20GB</td>
<td>5</td>
</tr>
<tr>
<td>ds-5</td>
<td>1,922,420</td>
<td>0.20</td>
<td>20GB</td>
<td>5</td>
</tr>
<tr>
<td>ds-6</td>
<td>771,465</td>
<td>0.38</td>
<td>8GB</td>
<td>2</td>
</tr>
</tbody>
</table>

window moves forward over time and only the remaining future access chunks in the window are stored to the log buffer. On the other hand, if a victim is identified as hot, we just evict the victim container without any logging because we can still achieve high container utilization with this hot container in the near future access. The rationale of this policy is that if we store many remaining future access chunks (in the hot container) into the log buffer, they also lead to the unnecessary eviction of many logged chunks in the buffer.

4.2.4 Performance Evaluation

This section describes our experimental setup and characteristics of the backup datasets, followed by the performance results for the proposed scheme.

4.2.4.1 Experimental Setup

I implement this dedupe storage simulator on the basis of the DiskSim simulator [98]. The underlying storage includes 9 individual disks (IBM DNE S-309170W), each of which provides storage capacity of 17,916,240 blocks (8.54GB). It is configured as RAID0 to provide better performance and enough storage space to accommodate all the chunks in the backup datasets. The stripe unit size is set to 32KB. We base our chunk indexing (hash index table) on google-sparsehash [99]. For read and write performance measurements, we ignore an elapsed time to execute the typical dedupe or our scheme because it is much smaller than storage I/O time. For the dedupe, we use a fixed sized container of 2MB for both and each container read (write) accesses 64 stripe units, where each
individual disk serves about 7–8 stripe unit reads (writes).

The experiments focus on a single data stream, not multiple data streams in parallel. However, this can be applicable to such multiple data stream environments. Considering multiple data streams in parallel, a small read cache size of each single data stream is an important factor. Thus, we adopt a small size of the read cache: from 2 (4MB) to 8 (16MB) size of the container. To read a chunk, the dedupe storage reads a container from the storage and caches it into the read cache after evicting one of the in-memory containers in the read cache based on our cache policy.

I employ six backup datasets (Table 4.5) traced in real data backup environments. Each dataset consists of 2 or 5 successive versions of (full) backup data streams. The ds-1, ds-2, and ds-3 datasets were obtained from all Exchange Server data. The ds-4 contains system data for a revision control system. The ds-5 includes data from the /var directory in the same machine. The ds-6 contains data from home directories with several users. For our experimental purpose, all the datasets except the last one (ds-6) were truncated to be 20GB in size and to have about 1,900K chunks in total. Each dataset contains chunked data streams by using variable-length chunking with an average chunk size of 8KB.

Each chunked data stream (hereafter, data stream) in the datasets consists of a sequence of chunk records each of which specifies the key chunk information including 20-byte chunk ID (SHA-1 hash value), its LBA information, dataset and version IDs, and a chunk size (not compressed size). The dedupe gain ratio (DGR) represents the ratio of the stored data stream size to the original data stream size. Most datasets except ds-3 contains many duplicated chunks.

4.2.4.2 Experimental Results

Figure 4.19 shows the read performance for LRU and our base design with various cache sizes. Note the the cache size of 2, 4, and 8 means the number of a data container (2MB). Thus, the cache size of two corresponds to 4MB cache. Moreover, due to the space limitation, we present only two datasets (ds-1 and ds-5) because, based on our extensive dataset analysis, four datasets (ds-1, ds-2, ds-3, and ds-4) exhibit very
similar data patterns and show almost identical result patterns. Similarly, the other two datasets (ds-5 and ds-6) also exhibit similar patterns. Therefore, we just choose one of each on behalf of the other(s) afterwards. As plotted in the Figure 4.19, our base design shows better read performance than LRU for all cache sizes only with the change of a cache replacement algorithm because our algorithm exploits future access information. Our proposed base scheme improves the dedupe read performance by an average of 14.5% and 16.8% respectively.

Figure 4.20, 4.21, 4.22, and 4.23 depict the performance improvement with our extended cache design with various configurations. First of all, in Figure 4.20 we explore
both the impact of a hot threshold value and performance improvement of our extended design. Hot data identification plays an important role in making decision about logging future access data chunks in a container into the log buffer. The hot threshold of 5% means that if the percentage of total data chunk access in a data container is over than 5%, the container is identified as hot, otherwise, cold. Only if it is identified as cold, the remaining future access chunks will be logged in the small buffer. Thus, intuitively, the higher a hot threshold, the better read performance because the more number of future access chunks will be able to be stored in the buffer. However, our design with the higher threshold than 3 or 5 does not lead to higher performance gain. In addition, we observe that our extended design (with a log buffer) does not considerably achieve performance improvement in ds-1, ds-2, ds-3, and ds-4 (2.3% improvement on average). However, in both ds-5 and ds-6, it significantly improves the read performance by an average of 63.1%. These results stem from the workload characteristics of the datasets and we have already addressed this in 4.2.3.2 Moreover, Figure 4.21 supports the experimental results in Figure 4.20 and shows very similar performance patterns because all performance gains of our extended design are fundamentally originated from the log buffer. We also observed that our extended design outperforms LRU by an average of 64.3% in ds-5 and ds-6.

The sliding window size is another factor to be discussed since it implies how much
we can look ahead in the near future. Interestingly, its impact is almost ignorable in $ds-1$ through $ds-4$ and a bigger window even worsens the performance in $ds-5$ and $ds-6$ (Figure 4.22). This is because the farther future accesses in a big window can contaminate a cache space, which can lead to low cache utilization in my cache policies. Lastly, I explore the impacts of various cache and log buffer sizes assuming the same total cache space. I vary both a cache size and a log buffer size.

Figure 4.23 shows the impact of diverse cache and log buffer sizes, where a total space (cache plus log buffer) is fixed. C and L stand for a cache size and a log buffer size respectively. As shown in Figure 4.23, I can observe that a read cache size is a more important factor in $ds-1$ through $ds-4$. Thus, assigning a more space into a read cache than into the log buffer will a better cache design in those types of datasets (we choose $ds-3$ instead of $ds-1$ in Figure 4.23 (a) since it shows clearer results). On the other hand, a larger log buffer has a more impact on the read performance in both $ds-5$ and $ds-6$.

4.2.5 Conclusion

In this section, I proposed a novel read cache design in dedupe storage for a backup application to improve read performance by looking ahead its future references. My idea was inspired by its special characteristic: the read sequence is the same as the
write sequence. As a result, for better cache utilization, the design can evict the data containers with smallest future references from the cache by looking ahead their future references in a moving window. Moreover, to achieve better read cache performance, this design maintains a small log buffer to judiciously maintain future access data chunks, not data containers. My experiments demonstrates that the proposed read cache scheme makes a big contribution to read performance improvement in dedupe storage without hurting any dedupe write performance.

This work has assumed that the basic read unit in secondary dedupe storage is a data container retaining hundreds of small data chunks. As my future work, I will remove such assumption so that each data chunk can be directly read from the underlying dedupe storage. In addition, I am considering a variable container size and an adaptive scheme to workloads by dynamically changing the hot threshold. Furthermore, I am designing a novel dedupe system which can provide guaranteed read performance while assuring enough write performance in dedupe storage. Ultimately, I plan to combine both designs for our complete dedupe system to achieve both good read performance and write performance.
Chapter 5

Conclusion and Future Work

This dissertation addressed the importance of hot data identification in storage areas and showed how it can be effectively applied to them to overcome the existing limitations in diverse storage venues. With this end, I proposed two hot data identification schemes: (1) multiple bloom filter-based scheme and (2) sampling-based scheme. Then I applied them to the Solid State Drives (SSD) design and data deduplication system.

The first hot data identification scheme adopted both multiple bloom filters and multiple hash functions to efficiently capture finer-grained recency as well as frequency information. For finer-grained recency capturing, a different recency coverage is assigned to each bloom filter. For frequency capturing, the number of bloom filter implicitly retains frequency information even though this scheme does not maintain any access counter. In addition to this proposed scheme, I propose a Window-based Direct Address Counting (WDAC) algorithm to approximate an ideal hot data identification as our baseline.

The second approach employed a sampling mechanism. This sampling-based hot data identification enabled the proposed scheme to early discard some of the cold items so that it can reduce runtime overheads and a waste of memory spaces. Moreover, its two-level hash indexing scheme helped the proposed scheme directly look up a requested item in the cache and saved a memory space further by exploiting spatial localities. Our experimental evaluation with diverse realistic workloads demonstrated that both
hot data identification schemes outperformed the state-of-the-art scheme.

In order to verify the effectiveness and importance of hot data identification in data storage fields, I selected two applications: Solid State Drive (SSD) design and data deduplication system.

First of all, in SSD design, hot data identification has a critical impact on both its performance and its reliability. To resolve these issues, I designed a new Flash Translation Layer (FTL) named Convertible Flash Translation Layer (CFTL). The CFTL is a new hybrid FTL and adaptive to data access patterns by adopting the multiple bloom filter-based hot data identification algorithm. CFTL, thus, can dynamically switch its mapping scheme to either a page level mapping or a block level mapping to fully exploit the benefits of both schemes. By smartly taking advantage of both schemes, CFTL resolved the intrinsic problems of the existing FTLs. In addition to this adaptive hybrid FTL design, I propose an efficient caching strategy and adaptive partitioning of mapping tables to further improve the CFTL performance.

As a second application, I explored data deduplication storage systems. Data deduplication (for short, dedupe) is a special data compression technique and has been widely adopted especially in backup storage systems to save backup time as well as a storage space. Unlike the traditional dedupe research that has focused more on the write performance improvement during the dedupe process, I focused my research on its read performance improvement. To improve the read performance without hurting its write performance, I newly designed a read cache in dedupe storage for a backup application by looking ahead their future references in a moving window with the combination of a hot data identification algorithm. Moreover, to achieve better read cache performance, our design maintained a small log buffer to judiciously maintain future access data chunks.

Both applications supported well the rationale for the importance of the effective hot data identification in data storage areas thereby efficiently improving the performance of each applications.

As my future work, I will design a dynamic hot data identification scheme. To the best of my knowledge, all of the existing hot data identification designs are static
schemes where their hot data definition or thresholds are not changed during their processes. In other words, I am considering an adaptive hot data identification design that is automatically adapted to various workloads. Hot data definition can be different from the different applications and my adaptive design will meet this motivation.
References


[17] Yuval Cassuto, M. A. A. Sanvido, C. Guyot, D. R. Hall, and Z. Z. Bandic. Indirect-

ion systems for shingled-recording disk drives. In Proceedings of the 2010 IEEE


issues for a shingled write disk system. In Proceedings of the 2010 IEEE 26th


Hot Data Identification into Shingled Write Disks. In Proceedings of the 20th IEEE

International Symposium on Modeling, Analysis and Simulations of Computer and

Telecommunication Systems (MASCOTS), August 2012.

[20] Rekha Pitchuman, Andy Hospodor, Ahmed Amer, Yangwook Kang, Ethan L.

Miller, and Darrell Long. Emulating a Shingled Write Disk. In Proceedings of

the 20th IEEE International Symposium on Modeling, Analysis and Simulations of


of USENIX HotStorage, 2010.

[22] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain
deduplication file system. In Proceedings of the 6th USENIX File and Storage

Technologies (FAST), 2008.

[23] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding up inline storage
deduplication using flash memory. In Proceedings of USENIX Annual Technical

Conference (ATC), 2010.

[24] Youngjin Nam, Dongchul Park, and David Du. Assuring Demanded Read Per-

formance of Data Deduplication Storage with Backup Datasets. In Proceedings of

the 20th IEEE International Symposium on Modeling, Analysis and Simulations of


the 10th USENIX Conference on File and Storage Technologies (FAST), February 2012.


