IEICE TRANS. INF. & SYST., VOL.E99–D, NO.6 JUNE 2016

PAPER

# A Dynamic Switching Flash Translation Layer Based on Page-Level Mapping

**Dongchul PARK**[†*a)], ***Member***, **Biplob DEBNATH**[†**b)], ***and*** **David H.C. DU**[††c)], ***Nonmembers***

**SUMMARY** The Flash Translation Layer (FTL) is a firmware layer inside NAND flash memory that allows existing disk-based applications to use it without any significant modifications. Since the FTL has a critical impact on the performance and reliability of flash-based storage, a variety of FTLs have been proposed. The existing FTLs, however, are designed to perform well for either a read intensive workload or a write intensive workload, not for both due to their internal address mapping schemes. To overcome this limitation, 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 design that adopts multiple bloom filters. Thus, CFTL can dynamically switch its mapping scheme to either page-level mapping or block-level mapping to fully exploit the benefits of both schemes. In addition, we design a spatial locality-aware caching mechanism and adaptive cache partitioning to further improve CFTL performance. Consequently, both the adaptive switching scheme and the judicious caching mechanism empower CFTL to achieve good read and write performance. Our extensive evaluations demonstrate that CFTL outperforms existing FTLs. In particular, our specially designed caching mechanism remarkably improves the cache hit ratio, by an average of 2.4×, and achieves much higher hit ratios (up to 8.4×) especially for random read intensive workloads.

***key words:*** *FTL, flash translation layer, CFTL, SSD, flash memory*

## 1. Introduction

Several distinguished features of flash memory have enabled flash-based storage devices (e.g., Solid State Drives) to be successfully adopted in enterprise markets as well as personal mobile markets [1]–[3]. They include fast random access, lower power consumption, shock resistance, and light weight [4], [5]. However, since it does not allow *in-place* update (i.e., overwrite), overwriting a page in a flash memory must be preceded by an erasure of the corresponding block (the flash memory space is partitioned into many blocks, where each block contains a fixed number (32 or 64) of pages). This in-place update problem results from the asymmetric operational granularity of flash memory: both read and write operations are performed on a page basis, whereas erase operations are executed on a block basis. According to [6], read operations take 15 $\mu$s, write operations take 200 $\mu$s, and erase operations take 2,000 $\mu$s. 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 fundamental 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 [7]. To resolve these problems, the flash translation layer (FTL) has been designed and deployed.

The FTL is a firmware layer implemented inside a flash-based storage device and emulates disk-like in-place updates: it first acquires a clean page and updates the corresponding data on that page. Next, it maps the original Logical Page Number (LPN) into this new Physical Page Number (PPN). Thus, an efficient FTL scheme has a critical impact on overall performance of flash memory since it directly affects in-place update performance and balancing the wearing of each data block.

In general, existing FTLs classifies largely into three schemes: page-level, block-level, and hybrid mapping. Page-level mapping [8] provides a fine granularity (i.e., page-level) address mapping so that it can achieve the best read/write performance, while it consumes a very large memory space to store a number of page mapping entries. An update to a page in page-level mapping may not have to trigger a block erase since it can use any clean page in any block for updating. On the other hand, an update to a page in block-level mapping will trigger the erase of the block containing the corresponding page. Consequently, the performance of block-level mapping for write intensive workloads is even lower than that of page-level mapping. However, for the read intensive workloads, the performance of block-level mapping is comparable to that of page-level mapping with much *less* memory space (theoretically, $1/N$, where $N$ is the number of pages in a block) because a read operation does not trigger block erasure. Although various hybrid mapping schemes [9]–[13] have been proposed to take advantage of both, they still suffer from a performance degradation because they are fundamentally designed on the basis of block-level mapping and added very limited page-level mapping restricted only to a small number of log blocks.

Considering the pros and cons of the existing FTLs and the tradeoff between the performance and the required memory space, we make the following observations: **1)** *with an*

Manuscript received October 5, 2015.
Manuscript revised January 26, 2016.
Manuscript publicized March 14, 2016.

[†]The authors were with the University of Minnesota–Twin Cities, Minneapolis, MN, U.S.A.
[††]The author is with Computer Science and Engineering, the University of Minnesota–Twin Cities, Minneapolis, MN, U.S.A.
[*]Presently, with Memory Solutions Lab (MSL) at Samsung Semiconductor Inc., San Jose, CA, U.S.A.
[**]Presently, with NEC Laboratories America, Princeton, NJ, U.S.A.
a) E-mail: park@cs.umn.edu
b) E-mail: biplob@umn.edu
c) E-mail: du@cs.umn.edu
DOI: 10.1587/transinf.2015EDP7406

*even less memory space, block-level mapping can offer a good performance for read intensive workloads due to its fast direct address translations*, **2)** *page-level mapping is best fit for write intensive workloads with a high block utilization and less erase operations*, **3)** *spatial locality (as well as temporal locality) in workloads can also help improve FTL performance*, and **4)** *a dynamic cache assignment can make the best use of a limited memory space*. Based on these observations, our goal is to design a dynamic FTL scheme adaptive to the workload behaviors. For write intensive workloads, it can provide the page-level-mapping-like performance for hot data, while for read intensive workloads, it provides the block-level-mapping-like performance for cold data. We define a data page updated frequently as *hot* data, and a data page read intensively with very infrequent or no updates as *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 judiciously deal with these conversions.

This paper proposes a novel hybrid FTL scheme named CFTL (Convertible Flash Translation Layer). CFTL, unlike other existing hybrid FTLs, is fundamentally rooted in 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 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 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 block-level mapping by using its adaptive feature. The key idea is that the mapping scheme is dynamically switched according to the data access patterns. In CFTL, since the mapping table is stored in the flash memory, there can be an overhead to look up the mapping tables. To reduce this overhead, we specially design a spatial locality-aware caching mechanism to get extra benefits from the spatial locality in workloads as well as a temporal locality. Moreover, our adaptive cache partitioning boosts the address translation efficiency of CFTL by fully exploiting a small memory space further. The main contributions of this paper are as follows:

- ***A Dynamic Switching FTL Scheme:*** CFTL is dynamically switched to either scheme according to data access patterns: Block-level mapping is in charge of read intensive data to exploit its fast direct address translation, while page-level mapping manages write intensive data to minimize erase operations.
- ***A Spatial Locality-Aware Caching Mechanism:*** For a faster address translation, CFTL adopts 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 localities (as well as temporal localities) in workloads.
- ***Adaptive Cache Partitioning:*** CFTL does not statically assign a memory space to both mapping tables (i.e., page and block) on the cache. Instead, it dynamically assigns a more cache space to either one in accordance with workload patterns to makes the effective use of a limited cache space.
- ***A New Hot Data Identification Scheme:*** In CFTL, hot data identification plays an important role in address mode switching. Thus, we developed a novel hot data identification scheme adopting multiple bloom filters and multiple hash functions to capture finer-grained recency information as well as frequency information.

The rest of this paper is organized as follows. Section 2 gives an overview of FTL and describes existing FTL schemes. Section 3 explains the design and operations of the CFTL scheme. Section 4 provides performance evaluation. Finally, Sect. 5 concludes the discussion.

## 2. Background and Related Work

### 2.1 Address Mapping Schemes in Flash Memory

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 mechanism [14]. FTL maintains the mapping table information either in a page-level, block-level, or hybrid manner.

Page-Level Mapping can map a logical page into any physical page in flash memory. Although it can achieve a good overall performance for both read and write operations, it requires a large amount of memory space to maintain the entire mapping table [15], [16]. As an example, 1 TB of a flash-based storage device requires 4GB of memory space only for the mapping table (assuming a 2KB page and 8 bytes per mapping entry).

In block-Level Mapping, a logical page address is made up of both a logical block number and an offset. This can save the memory space 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.

To overcome the aforementioned limitations of both mapping schemes, diverse hybrid approaches have been proposed [9], [10], [12], [13], [17]. Most of them are based on a log buffer approach by adopting a limited number of

log blocks to improve the write performance. The memory usage for mapping can also be lessened since only a small number of log blocks are allocated for page-level mapping. However, the log blocks eventually need to be erased and this will trigger multiple block merges to reclaim them in the future.

### 2.2 Related Work

DFTL (Demand-based Page-Level FTL) [15], [18] is a page-level FTL scheme and tries to overcome the memory space problem by storing its complete mapping table on flash memory, not in DRAM. This approach can save a memory space, but gives rise to extra overheads for flash memory lookup. Moreover, DFTL suffers from frequent updates to its flash mapping table under the write-dominant workloads.

Several hybrid schemes have been proposed. BAST (Block Associative Sector Translation) [9] scheme classifies blocks into two types: data blocks for data saving and log blocks for overwrite operations. This log buffer concept enables BAST to reduce erase operations, but still suffers from block merge operations and low block utilization. FAST (Fully Associative Sector Translation) [17] is based on BAST scheme, but allows log blocks to be shared by all data blocks. Even though this scheme accomplishes better utilization of log blocks, random log blocks give rise to the more complicated block 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. However, coarse-to-fine switches incur corresponding fine-to-coarse switches, which causes overheads in valid data page copies.

## 3. CFTL: Convertible Flash Translation Layer

### 3.1 Architecture

CFTL stores the complete page mapping table (named *tier-2 page mapping table*) in the flash memory. Thus, it needs to read the page mapping table from flash to look up the location of the original data. This page table lookup requires at least one flash read operation. To get over this overhead, it caches parts of the mapping table in SRAM. As shown in Fig. 1, 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 use of a 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 to improve the hit ratio of CPMT by judiciously exploiting the spatial locality. This will be explained in more detail in Sect. 3.4. CBMT is a block mapping table and, like CPMT, serves as a cache to exploit both localities in a block-level mapping. CBMT translates logical block numbers (LBN) to physical

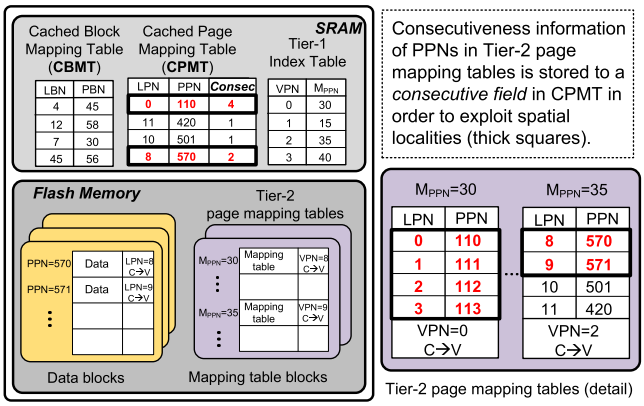



**Fig. 1** CFTL architecture. Here, PPNs (110-113 and 570-571) are consecutive. So, the numbers (4 and 2) are stored to the *consecutive field* in CPMT. VPN, LPN and PPN stand for virtual page number, logical page number and physical page number respectively. In addition, C, V and I correspond to clean, valid and invalid respectively.

block numbers (PBN), which enables fast direct access to flash data blocks in conjunction with page offsets.

In addition to both CBMT and CPMT, there exists another mapping table in SRAM. We define this SRAM-based mapping table as a *tier-1 index table*. This tier-1 index table keeps track of the on-flash locations of pieces of the tier-2 page mapping tables. 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 correspond to the complete page mapping table 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 tables reside in flash memory, whenever any mapping information is updated, a new page is assigned and all mapping information in the old page (i.e., mapping table) is copied to the new page with the updated mapping information. This is because both read and write operations to flash memory are performed on a page basis. Therefore, we need to maintain a tier-1 index table to keep track of each tier-2 page mapping table which can be distributed over the flash memory whenever it is updated. Each page (i.e., one tier-2 mapping table) can hold 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, for instance, a 4GB flash memory device needs only 8MB ($2^{12} \times 2$KB) of space to store all the required $2^{12}$ (4GB/1MB per page) number of tier-2 mapping table pages in flash.

### 3.2 Addressing Mode Switches

• *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. CFTL makes decisions about address mode switching based on the hot data identification. Therefore, we developed a

novel hot data identification algorithm by adopting multiple bloom filters (for short, BF) and multiple hash functions [19].

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 can present 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 [20]. Recency (as well as the frequency) is another important factor to identify hot data. To precisely capture recency, all information of each BF will be periodically erased by turns, 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, whereas 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. Consequently, it can capture fine-grained recency information.

Figure 2 presents the operation of our hot data identification scheme. We assume that our scheme adopts a set of $V$ independent BFs and $K$ independent hash functions to capture both frequency and recency, and each BF consists of $M$ bits to record $K$ hash values (we employed $V = 4$, $K = 2$, and $M = 2{,}048$ in our scheme). Whenever a write request is issued to the FTL, the corresponding LBA is hashed by the $K$ hash functions, and $K$ hash values set the corresponding $K$ bits in the first BF to 1. For the next write request, it chooses the next BF in a round robin manner to record its hash values. To classify the incoming data as hot or cold, it calculates a total hot data index value by combining the frequency and the recency value for each write request. If it 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 [19]. Hot and cold data identification itself, however, is out of scope of this paper. Different algorithms such as the LRU discipline [21], sampling-based approach [22], or multihash function scheme [20] can also substitute for our approach.

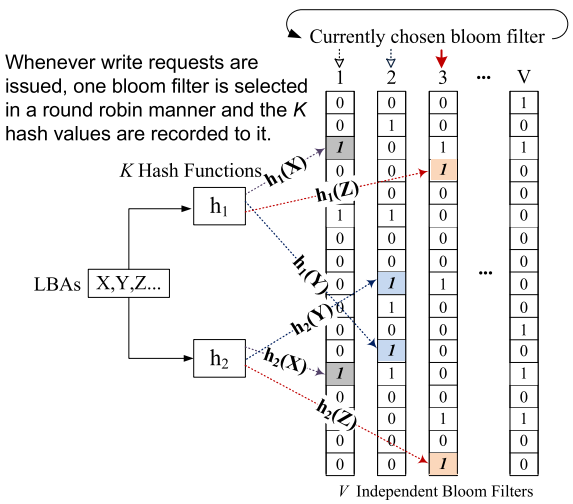



**Fig. 2** Our hot and cold data identification scheme adopting multiple bloom filters and multiple hash functions.

• *Page to Block Mapping:* If the hot data identifier classifies some data into cold data, addressing mode of those data is switched to a block-level mapping during the garbage collection. In particular, when the cold data pages in a logical block are physically distributed over flash, we need to collect those pages into a new physical data block. Then, we pass this block mapping information into CBMT for a block-level mapping. Whereas, when all valid physical data pages in a logical block are identified as cold data and saved in a consecutive manner, they can be switched to a block-level mapping without any extra cost. However, CFTL does not convert such types of cold data that has not been updated for a long time to a block-level mapping. Converting all cold data that have not been referenced for a long time to the block-level mapping wastes a memory space for the mapping table and pays an expensive read/write cost to collect them. This is based on the fact that generally only a small fraction of the address space for a disk is frequently referenced [20]. When they are frequently accessed in a read manner, CFTL converts this type of cold data blocks to a block-level mapping.

• *Block to Page Mapping:* In the case of write dominant access patterns, the hot data classifier makes a decision to switch from a block to a page-level mapping. Even though the workload does not exhibit 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 the same 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. Consequently, CFTL can completely remove log blocks for page updates which are the main cause of the expensive full block merges in other hybrid FTLs. Unlike the mode change from a page to a block mapping, this mode switch does not require any extra costs because a page mapping table is always valid to all data in flash. Therefore, when a hot data classifier in CFTL identifies some data as hot data, CFTL simply remove the corresponding block mapping entries from CBMT. Then, those data can be only accessed by a page mapping table, not by a block mapping table.

### 3.3 Address Translation Process

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 map-

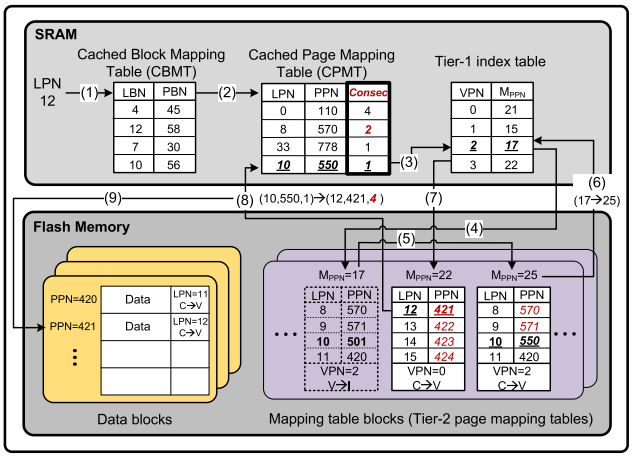



**Fig. 3** CFTL address translation process. This example presents a worst case scenario.

ping 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 CBMT or CPMT, 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 [23]. CFTL employs delayed updates to reduce frequent cache flush overheads. Thus, when CFTL chooses a victim, it first checks if the victim in CPMT is valid. If it is valid, 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. 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 Fig. 3). The corresponding tier-1 index table is updated to reflect the new tier-2 page mapping table (Step 6 in Fig. 3).

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 Fig. 3). Finally, the newly fetched mapping information is stored into the space available in CPMT (Step 8 in Fig. 3).

CFTL employs a two-level address translation scheme so that a worst case (i.e., requested mapping information does not exist in the cache, the cached mapping tables are full, and the victim entry is invalid) read latency requires three page reads and one page write. Similarly, a worst case write latency needs two page reads and two page writes. However, CFTL can considerably reduce this overhead, which is the main motivation of our efficient caching mechanism.

### 3.4 A Spatial Locality-Aware Caching Mechanism

Our proposed caching strategy in CFTL is inspired by the following idea: all PPNs (Physical Page Numbers) in a data block are consecutive [24]. As in Fig. 1, CFTL adds one more field named a *consecutive field* in CPMT 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.

In Fig. 1, 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 correspond to 4 respective logical addresses from LPN = 0. That is, LPN = 1 is mapped to PPN = 111, similarly, LPN = 2 is to PPN = 112, and LPN = 3 is to PPN = 113. If any page in the consecutive pages is updated, we need to split and update both consecutive fields information accordingly.

This consecutive field in CPMT enables CPMT to judiciously exploit a spatial locality as well as a temporal locality. With the help of this simple field, even though CPMT does not store the requested mapping information, the consecutive field can provide a hint to increase the cache hit ratio. This achieves higher address translation efficiency with the same number of mapping table entries.

### 3.5 Adaptive Cache Partitioning

Originally CFTL assigns an even memory space to both CBMT and CPMT. However, this static assignment can waste a memory space. To make the best use of this, CFTL with adaptive cache partitioning dynamically adjusts the sizes of both CBMT and CPMT in SRAM according to workload patterns. CFTL is fundamentally based on page-level mapping and, especially at the beginning, it always uses up CPMT earlier than CBMT. Consequently, CFTL with adaptive cache partitioning initially assigns a more memory space to CPMT than CBMT (initially 80% vs. 20%, but this ratio can be configurable). 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. That is, as write requests increase, CFTL assigns more spaces to CPMT. When CFTL takes a space from CBMT, it can remove the block mapping information without any extra cost and reuses the space for CPMT. However, when CFTL takes a space 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 Sect. 3.3. Similarly, CFTL allots more spaces to

CBMT with read intensive workloads. The ratio between CPMT and CBMT changes dynamically and one mapping table can take a one entry space from the other *on demand*. This adaptive partitioning mechanism enables CFTL to find an optimal ratio between them so that CFTL not only efficiently utilize a limited memory space, but also improves its performance further.

### 3.6 Discussion

• *Read Performance:* DFTL [15] shows a good read performance under the condition of high temporal locality. However, under totally random read intensive patterns (i.e., low temporal locality), it suffers from performance degradation because of many cache misses in SRAM. CFTL, on the other hand, exhibits a good read performance even under the low temporal locality because read intensive data are dynamically converted to block-level mapping. Moreover, its elaborate caching mechanism improves it further by exploiting spatial locality. FAST shows a comparable read performance to page-level mapping only if the workloads contain no write operations. Theoretically, if there exist workloads that have never been updated since they were initially deployed (to prevent extra page read overhead in log blocks) and retains 100% read access patterns, FAST may be able to show a comparable read performance to CFTL. However, in practice, FAST does not reach both page-level mapping and CFTL in realistic read intensive workloads. AFTL provides relatively lower read performance than the other schemes in read intensive workloads because the full fine-grained slots inevitably trigger valid data page copies for coarse-grained mapping. However, if workloads contain write accesses, AFTL shows a better read performance than FAST (but still lower than both CFTL and DFTL) because FAST significantly suffers from merge operations.

• *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 in FAST and frequent erase operations to both primary and replacement blocks in AFTL, which ultimately results in a poor write performance. On the other hand, a page-level mapping can get rid of full merge operations. Thus, both FAST and AFTL write performance never reach a page-level mapping scheme. Although CFTL uses a hybrid approach, it achieves the good write performance of page-level mapping since all data in CFTL is fundamentally managed by two-tier page-level mapping. Thus, both CFTL and DFTL achieve a good write performance. However, CFTL shows a better overall write performance than DFTL due to its faster address translation resulting from our elaborate caching scheme.

## 4. Performance Evaluation

### 4.1 Evaluation Setup

A 32GB NAND flash memory is simulated with configurations shown in Table 1. Our experiments of flash memory are based on the product specification of Samsung's K9XXG08UXM series NAND flash part [6], [25]. For an ideal page-level mapping (selected as our baseline scheme), we assume that the whole mapping table can be 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 [16]). Various types of workloads including real trace data sets are employed for more objective evaluations (Table 2). Websearch [26] trace made by Storage Performance Council (SPC) represents well a read intensive I/O trace. Although a Websearch trace consists of three trace files (i.e., Websearch1, Websearch2, and Websearch3), the characteristics of each trace file are almost identical (i.e., heavily read intensive). Since our experiments also showed almost the same results for each trace, we adopt just one of them (i.e., Websearch3). As a write intensive trace, we employ Financial1 [27] made from an OLTP application running at a financial institution. For the totally random performance measurements, we based random traces upon Financial2 [27] 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. Interestingly Gupta et al. [15] classified Financial2 trace as a write dominant trace, but our study clearly shows read dominant access patterns (refer to Table 2). So, we classify the Financial2 trace as a read intensive I/O trace.

**Table 1** Simulation configurations

| Parameters | Values |
|---|---|
| Page Read to Register | 25$\mu$s |
| Page Write from Register | 200$\mu$s |
| Block Erase | 1.5ms |
| Serial Access to Register (Data Bus) | 50$\mu$s |
| Page Size | 2KB |
| Data Register Size | 2KB |
| Block Size | 128KB |
| Entries in Mapping Tables | 4,096 entries |

**Table 2** Workload characteristics

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

### 4.2 Evaluation Results and Analyses

***1) Overall Performance:*** Figure 4 presents an overall performance under various workloads. The overall performance of CFTL is very close to that of an ideal page-level mapping. CFTL provides a better read performance than FAST [17] which has a strong point in a read performance. It also exhibits a better write performance over DFTL [15] which has an excellent write performance.

Under read intensive workloads (Fig. 4 (a), (b), and (d)), CFTL switches more and more data to block-level mapping over time to make the best use of its fast direct address translation. Moreover, our caching mechanism makes a considerable contribution to read performance improvement since almost consecutive data pages in a data block are not updated with these access patterns. In particular, as in Fig. 4 (d), our proposed caching scheme demonstrates its notable effectiveness in read performance improvement especially under random read intensive workloads. Compared to Fig. 4 (a), the random read performance of most FTLs is significantly degraded. However, CFTL still maintains good performance due to its smart caching mechanism. We will explore this in more detail later.

FAST also provides a good performance under the read intensive workloads. However, it does not reach CFTL performance because even though there are not many updates in this workload, they still affect its overall performance due to merge operations in FAST. In addition, the extra data read cost in log blocks is another factor to degrade its read performance. DFTL is two-tier page-level mapping like CFTL. If, however, the workload does not retain a high temporal locality, it cannot avoid an additional address translation overhead. This is the main reason that DFTL does not exhibit relatively good random read performance (Fig. 4 (d)). Unlike other schemes, AFTL [13] does not show a good performance in read intensive workload because when the fine-grained slots are full, it starts to cause valid data page copies for coarse-grained mapping. This incurs extra overheads.

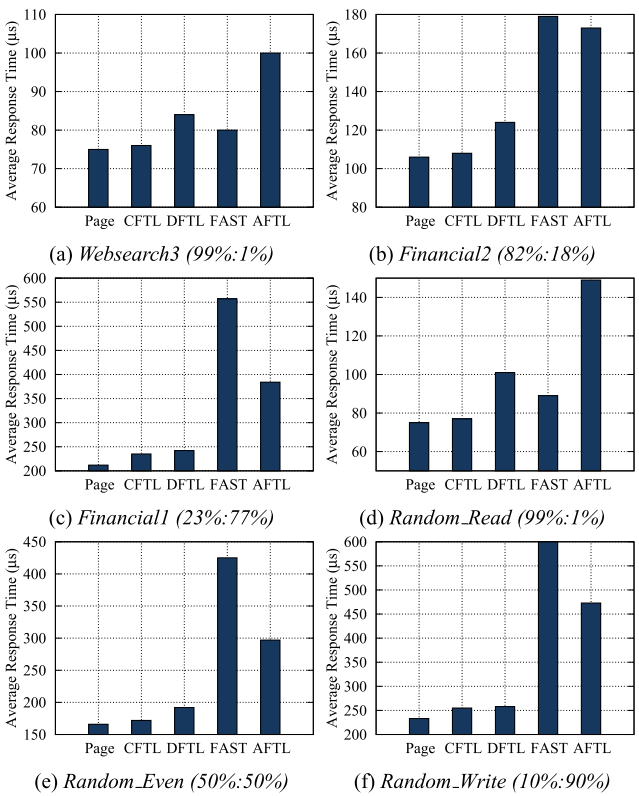



**Fig. 4** Overall performance under various workloads (read %: write %)

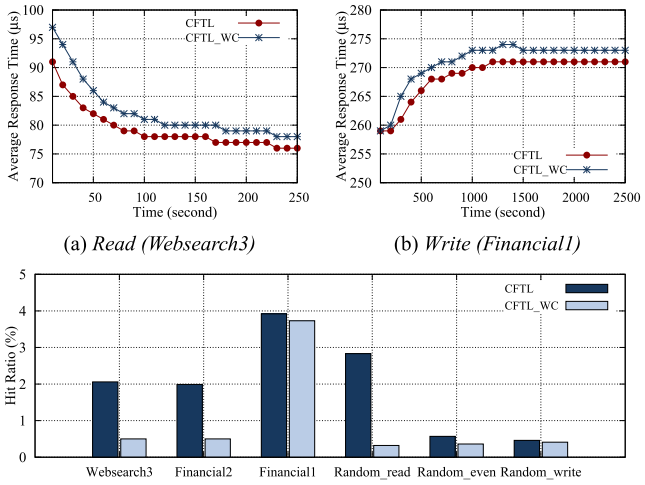
As the ratio of write requests in workloads grows, performance variations among each scheme also increase rapidly because the frequent write operations cause frequent updates in flash. As plotted in Fig. 4 (c), (e), and (f), the overall performances of AFTL and FAST are severely degraded under the 50% read and 50% write workload, not to mention write intensive workloads. This stems fundamentally from frequent erase operations in both schemes. Frequent updates produce frequent merge operations in FAST and frequent erase operations to both primary and replacement blocks in AFTL. On the other hand, since DFTL is page-level mapping and CFTL is fundamentally based on the page-level mapping, both schemes show significantly better performances than AFTL and FAST. This is evident particularly under write dominant workloads (Fig. 4 (c) and (f)). However, CFTL shows better overall write performance than DFTL due to its faster address translation resulting from our elaborate caching mechanism.

***2) A Spatial Locality-Aware Caching Mechanism:*** We prepare two different CFTL schemes: CFTL and CFTL_WC. 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 identical parameters only except the caching strategy.

As in the Fig. 5, CFTL exhibits its dominant request hit ratio against CFTL_WC 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 a cache hit ratio by 8.8× for the random read intensive workload (i.e., Random_read) compared to CFTL_WC. These results demonstrate that our spatial locality-aware caching mechanism achieves significant performance improvement for the read intensive workloads, and surprising performance improvement particularly for the random 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 CFTL_WC is relatively reduced. Even though the gaps are lessened, CFTL still shows higher cache hit ratios than CFTL_WC.

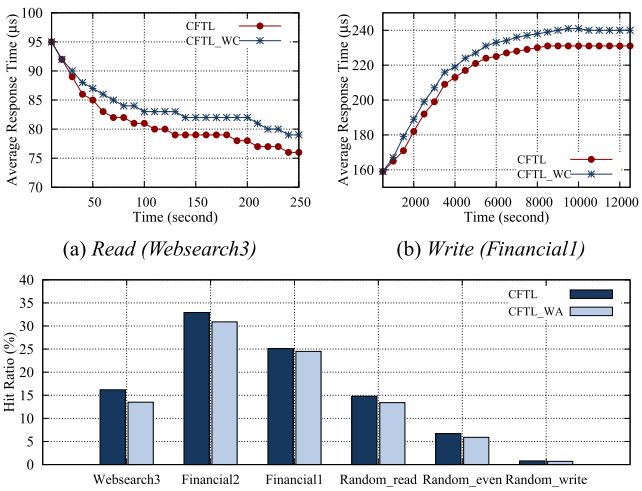
Figure 5 (a) presents that even though the CFTL_WC




(a) *Read (Websearch3)* (b) *Write (Financial1)*

(c) *Request hit ratios in CPMT*

**Fig. 5** Performance improvement with our spatial locality-aware caching mechanism. CFTL_WC stands for the CFTL *without* our proposed caching strategy.




(a) *Read (Websearch3)* (b) *Write (Financial1)*

(c) *Total request hit ratios in both CPMT and CBMT*

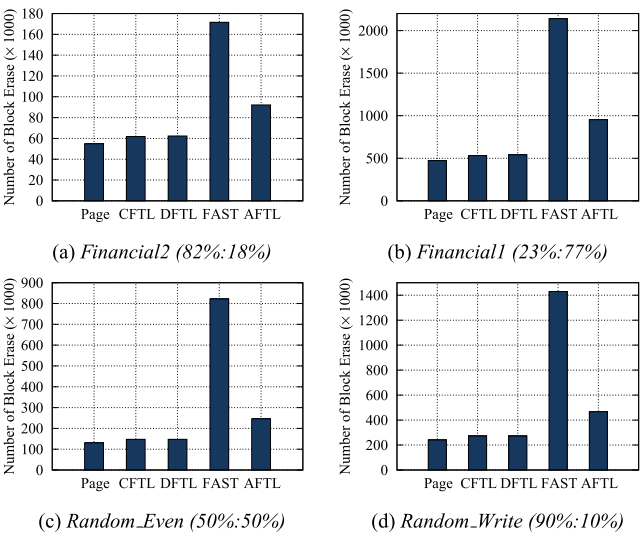
**Fig. 6** Performance improvement with our adaptive cache partitioning. CFTL_WA stands for the CFTL *without* our adaptive partitioning scheme.

does not have the efficient caching mechanism, 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 block-level mapping so that the read performance of CFTL_WC gradually approaches that of CFTL after some time. This proves the effectiveness of the adaptive feature of CFTL. However, CFTL with the caching mechanism can offer its better read performance from the beginning, even before CFTL_WC are switched to block-level mapping.

***3) Adaptive Cache Partitioning:*** Our adaptive cache partitioning scheme dynamically tunes the ratio of CBMT and CPMT. To evaluate its effectiveness, we again prepare two different CFTL schemes: CFTL and CFTL_WA. CFTL retains our proposed dynamic partitioning scheme as well as the aforementioned caching mechanism, while CFTL_WA is not equipped with this dynamic partitioning scheme but includes the spatial locality-aware caching mechanism.

As in Fig. 6, both CFTL and CFTL_WA start identical read and write performance at the beginning since both CBMT and CPMT are not full yet. However, as time goes on, CFTL shows a better performance due to our proposed adaptive cache partitioning scheme. These performance gaps ultimately result from total cache hit ratios of both CBMT and CPMT, and Fig. 6 (c) adds support to this claim.

***4) The Number of Block Erase:*** Figure 7 shows the number of block erase 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 adopt both workloads. As shown in Fig. 7, both CFTL and DFTL do not give rise to as many block erases as FAST and AFTL because DFTL is

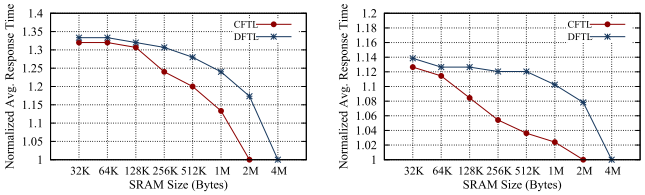



(a) *Financial2 (82%:18%)* (b) *Financial1 (23%:77%)*

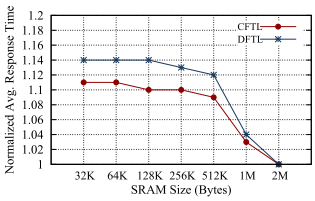
(c) *Random_Even (50%:50%)* (d) *Random_Write (90%:10%)*

**Fig. 7** The number of block erase. We do not present heavily read intensive workloads (Websearch3 and Random_read) due to their significantly small numbers of erases. (read %: write %)

page-level mapping and CFTL is fundamentally rooted in page-level mapping. These 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 judicious caching schemes. Assuming the Cached Mapping Table in DFTL is already full, 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 flash page to update a page mapping table. On the other hand, CFTL maintains a consecutive field to improve the cache hit ratio. Although, compared to a read operation, this field does not considerably improve the cache hit ratio in write operations due to a break of the address consecutiveness, it still does help in-




(a) *Websearch3 (99%:1%)* (b) *Random_Even (50%:50%)*




(c) *Financial1 (23%:77%)*

**Fig. 8** Impact of SRAM size in CFTL and DFTL. The average response time is normalized in terms of the ideal page-level mapping scheme. (read %: write %)

crease the cache hit ratio, which results in a little bit less number of block erases than DFTL. Moreover, this clearly explains why there are no big differences of block erases between 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 many block erases. Similarly, assuming AFTL used up all fine-grained slots, whenever write requests are issued, the corresponding number of entries evicted from the fine-grained slots must move to a coarse-grained slots, which eventually causes a lot of valid page copies and block erases.

***5) Impact of SRAM Size:*** Evaluating performance of both CFTL and DFTL with the same size of SRAM is also very meaningful. Figure 8 presents the normalized average response time in CFTL and DFTL with variable SRAM size. Since each value is normalized in terms of 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 in Fig. 8, CFTL provides a 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 (SRAM) Consumption:*** 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 [16]).

Ideal page-level mapping consumes 8MB to accommodate its complete page mapping table. This is an exceptionally large space compared to the other schemes. Both AFTL [13] and FAST [17] also consume a lot of memory spaces (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 other hand, both CFTL and DFTL [15] 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 and maintains one additional mapping table (CBMT). 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 page-level mapping and 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. CFTL adopts bloom filters to notably save a memory space as well as computation overheads (it consumes only 8KB for the hot data identification scheme). Typical FTL schemes maintain block access counters for hot data identification, which generally requires a lot more memory spaces.

## 5. Conclusion

This paper proposed a novel hybrid FTL scheme named Convertible Flash Translation Layer (CFTL) for flash-based storage devices. CFTL can dynamically switch its mapping scheme between page-level mapping and block-level mapping according to data access patterns to fully exploit the benefits of both. Since CFTL, unlike other existing hybrid FTLs, is fundamentally based on page-level mapping, it overcomes the inborn limitations of them. CFTL stores the entire mapping table in the flash memory. Thus, there is an overhead to look up address mapping information in the data page. To resolve this issue, we also proposed a smart caching scheme including a spatial locality-aware caching mechanism and adaptive cache partitioning.

Our experiments show that CFTL outperforms DFTL [15] by up to 24% for the read intensive workloads, by up to 47% for the random read intensive workloads, and by up to 4% for the write intensive workloads. Moreover, our proposed caching mechanism improves the address translation efficiency by significantly increasing the cache hit ratio, by an average of 2.4×, and by up to 8.4× especially for the random read intensive workloads.

### References

[1] X. Jimenez, D. Novo, and P. Ienne, "Wear unleveling: Improving NAND flash lifetime by balancing page endurance," FAST, 2014.

[2] J. Jeong, S.S. Hahn, S. Lee, and J. Kim, "Lifetime improvement of NAND flash-based storage systems using dynamic program and erase scaling," FAST, 2014.
[3] S. Lee, T. Kim, K. Kim, and J. Kim, "Lifetime management of flash-based SSDS using recovery-aware dynamic throttling," FAST, 2012.
[4] K. Bang, K.-I. Im, D.-G. Kim, S.-H. Park, and E.-Y. Chung, "Power Failure Protection Scheme for Reliable High-Performance Solid State Disks," IEICE Transactions on Information and Systems, vol.E96-D, no.5, pp.1078–1085, 2013.
[5] Samsung, "Why SSDs are Awesome: An SSD Primer," tech. rep., Samsung Electronics, 2014.
[6] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," USENIX ATC, 2008.
[7] D. Park, B. Debnath, and D.H.C. Du, "A Workload-Aware Adaptive Hybrid Flash Translation Layer with an Efficient Caching Strategy," MASCOTS, pp.248–255, July 2011.
[8] CompactFlashAssociation, "http://www.compactflash.org."
[9] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A Space-efficient Flash Translation Layer for CompactFlash Systems," IEEE Transactions on Consumer Electronics, vol.48, no.2, pp.366–375, 2002.
[10] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of Flash Translation Layer," J. Syst. Archit., vol.55, no.5-6, pp.332–343, 2009.
[11] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation," ACM Trans. Embed. Comput. Syst., vol.6, no.3, 2007.
[12] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-aware Sector Translation for NAND Flash Memory-based storage Systems," SIGOPS Oper. Syst. Rev., vol.42, no.6, pp.36–42, 2008.
[13] C.-H. Wu and T.-W. Kuo, "An adaptive two-level management for the flash translation layer in embedded systems," ICCAD, pp.601–606, 2006.
[14] F. Chen, T. Luo, and X. Zhang, "CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives," FAST, 2011.
[15] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," ASPLOS, pp.229–240, 2009.
[16] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A Superblock-based Flash Translation Layer for NAND Flash Memory," EMSOFT, pp.161–170, 2006.
[17] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer-Based Flash Translation Layer Using Fully-Associate Sector Translation," ACM Transactions on Embedded Computing System, vol.6, no.3, 2007.
[18] Y. Kim, A. Gupta, and B. Urgaonkar, "A Temporal Locality-Aware Page-Mapped Flash Translation Layer," Journal of Computer Science and Technology, vol.28, no.6, pp.1025–1044, 2013.
[19] D. Park and D.H.C. Du, "Hot Data Identification for Flash-based Storage Systems Using Multiple Bloom Filters," MSST, pp.1–11, 2011.
[20] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient Identification of Hot Data for Flash Memory Storage Systems," ACM Transactions on Storage, vol.2, no.1, pp.22–40, 2006.
[21] L.-P. Chang and T.-W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," RTAS, pp.187–196, 2002.
[22] D. Park, B. Debnath, Y. Nam, D.H.C. Du, Y. Kim, and Y. Kim, "HotDataTrap: A Sampling-based Hot Data Identification Scheme for Flash Memory," SAC, pp.1610–1617, March 2012.
[23] R. Karedla, J.S. Love, and B.G. Wherry, "Caching Strategies to Improve Disk System Performance," Computer, vol.27, no.3, pp.38–46, 1994.
[24] D. Park, B. Debnath, and D. Du, "CFTL: A Convertible Flash Translation Layer Adaptive to Data Access Patterns," SIGMETRICS, vol.38, no.1, pp.365–366, 2010.
[25] SamsungElectronics, "K9XXG08XXM Flash Memory Specification," 2007.
[26] UMass, "Websearch Trace from UMass Trace Repository." http://traces.cs.umass.edu/index.php/Storage/Storage, 2002.
[27] UMass, "OLTP Trace from UMass Trace Repository." http://traces.cs.umass.edu/index.php/Storage/Storage, 2002.

**Dongchul Park** is currently a research scientist in Memory Solutions Lab. (MSL) at Samsung Semiconductor Inc. in San Jose, California, USA. He received his Ph.D. in Computer Science and Engineering at the University of Minnesota–Twin Cities in 2012, and was a member of Center for Research in Intelligent Storage (CRIS) group under the advice of Professor David H.C. Du. His research interests focus on storage system design and applications including non-volatile memories, in-storage computing, big data processing, Hadoop MapReduce, data deduplication, key-value store, hot/cold data identification, and shingled magnetic recording (SMR) technology.

**Biplob Debnath** is a research scientist at NEC Laboratories America, Princeton, New Jersey, USA. He received the Ph.D. degree in Electrical and Computer Engineering from the University of Minnesota–Twin Cities in 2010, and the and B.S. degree in Computer Science and Engineering from Bangladesh University of Engineering and Technology, Bangladesh, in 2002. He was a member of Center for Research in Intelligent Storage (CRIS) group under the advice of Professor David J. Lilja. His research interests include Nonvolatile memory, data deduplication, key-value store, and storage systems.

**David H.C. Du** is currently the Qwest Chair Professor in Computer Science and Engineering at the University of Minnesota–Twin Cities and the Center Director of the Center of Research in Intelligent Storage (CRIS). He received his Ph.D. from University of Washington, Seattle in 1981. His current research focuses on intelligent storage systems, multimedia computing, sensor networks, and cyber physical systems. He was a Program Director (IPA) at National Science Foundation (NSF) CISE/CNS Division from March 2006 to August 2008, and has served as a conference chair, program committee chair, and general chair for many major conferences. He is an IEEE Fellow (since 1998) and editorial boards of several major international journals.