Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 09-023

CFTL: A Convertible Flash Translation Layer with Consideration of Data Access Patterns

Dongchul Park, Biplob Debnath, and David Du

September 14, 2009

CFTL: A Convertible Flash Translation Layer with Consideration of Data Access Patterns

Dongchul Park, Biplob Debnath, and David Du
University of Minnesota, Twin Cities
200 Union Street SE, Minneapolis, MN 55455, USA
park@cs.umn.edu, Biplob@umn.edu, du@cs.umn.edu

ABSTRACT

NAND flash memory-based storage devices are increasingly adopted as one of the main alternatives for magnetic disk drives. The flash translation layer (FTL) is a software/hardware interface inside NAND flash memory, which allows existing disk-based applications to use it without any significant modifications. Since FTL has a critical impact on the performance of NAND flash-based devices, a variety of FTL schemes have been proposed to improve their performance. However, existing FTLs perform well for either a read intensive workload or a write intensive workload, not for both of them due to their fixed and static address mapping schemes. To overcome this limitation, in this paper, we propose a novel FTL addressing scheme named as Convertible Flash Translation Layer (CFTL, for short). CFTL is adaptive to data access patterns so that it can dynamically switch the mapping of a data block to either read-optimized or write-optimized mapping scheme 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 so as to considerably improve the CFTL performance further with only a simple hint. Consequently, both of the convertible feature and caching strategy empower CFTL to achieve good read performance as well as good write performance. Our experimental evaluation with a variety of realistic workloads demonstrates that the proposed CFTL scheme outperforms other FTL schemes.

1. INTRODUCTION

NAND flash memory has come into wide use as main data storage media in mobile devices, such as PDAs, cell phones, digital cameras, embedded sensors, and notebooks due to its superior characteristics: smaller size, lighter weight, lower power consumption, shock resistance, lesser noise, non-volatile memory, and faster read performance [1, 2, 3, 4, 5]. Recently, to boost up I/O performance and energy savings, flash-based Solid State Drives (SSDs) are also being increasingly adopted as a storage alternative for magnetic disk drives by laptops, desktops, and enterprise class servers [3, 4, 5, 6, 7, 8, 9]. Due to the recent advancement of the NAND flash technologies, it is expected that NAND flash-based storages will have a great impact on the designs of the future storage subsystems [4, 5, 7, 10].

A distinguishing feature of flash memory is that read operations are very fast compared to magnetic disk drive. Moreover, unlike disks, random read operations are as fast as sequential read operations as there is no mechanical head movement. However, a major drawback of the flash memory is that it does not allow *in-place* updates (i.e., overwrite). In flash memory, data are stored in an array of blocks. Each block spans 32-64 pages, where a page is the smallest unit of read and write operations. Page write operations in a flash memory must be preceded by an erase operation and within a block pages need be to written sequentially. The in-place update problem becomes complicated as write operations are performed in the page granularity, while erase operations are performed in the block granularity. The typical access latencies for *read*, *write*, and *erase* operations are 25 microseconds, 200 microseconds, and 1500 microseconds, respectively [7]. In addition, before the erase is done on a block, the live (i.e., not over-written) pages in that block need to be moved to pre-erased blocks. Thus, an erase operation incurs a lot of page read and write operations, which makes it a performance critical operation. Besides this asymmetric read and write latency issue, flash memory exhibits another limitation: a flash block can only be erased for a limited number of times (e.g., 10K-100K) [7]. Thus, frequent block erase operations reduce the lifetime of the flash memory. This is known as *wear-out* problem.

The flash translation layer (FTL) is a software/firmware layer implemented inside a flash-based storage device to make the linear flash memory device to act like a magnetic disk drive (as shown in Figure 1). FTL emulates disk-like in-place update for a logical page number (LPN) by writing the new page data to a different physical page number (PPN). It maintains a mapping between each LPN and its current PPN. Finally, it marks the old PPN as invalid for the later garbage collection. Thus, FTL enables existing application to use flash memory without any modification. However, internally FTL needs to deal with physical characteristics of the flash memory. Thus, an efficient FTL scheme makes a critical effect on overall performance of flash memory as it directly affects in-place update performance and wear-out problem.

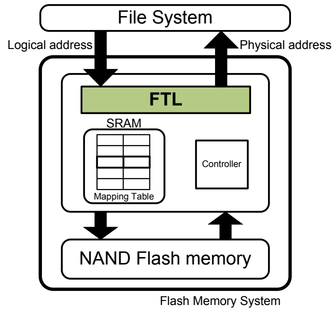


Figure 1: NAND Flash memory system architecture

Existing FTLs use various logical to physical page mapping strategies to solve the in-place update and wear-out problem. The mapping can be maintained at the page level, block level, or a combination of them (hybrid). In a page level mapping case, FTL maintains a mapping between each logical page number (LPN) and physical page number (PPN). Although a page level mapping scheme [11] has its merits in high block utilization and good read/write performance, but it requires very large memory space to store the entire page mapping table. To overcome this large memory limitation, a block level mapping FTL tries to map each logical block number (LBN) to physical block number (PBN). In addition, inside a block, page offsets are always fixed. An update to a page in a page level mapping may not have to trigger a block erasure, while an update to a page in a block level mapping will trigger the erasure of the block containing the corresponding page. 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 even with much less memory space requirement.

To take advantage of the both page level and block level mapping, various hybrid schemes [12, 13, 14, 15, 16, 17] have been proposed. Most of these schemes are fundamentally based on a block level mapping with an additional page level mapping restricted only to a small number of log blocks in order to delay the block erasure. However, for write intensive workloads, hybrid FTL schemes still suffer from performance degradation due to the excessive number of block erase operations. Recently, Gupta et al. proposed a pure page level mapping scheme called DFTL (Demand-based Flash Translation Layer) with a combination of small SRAM and flash memory [18]. In DFTL, the entire page level mapping is stored in the flash memory and temporal locality is exploited in an SRAM cache to reduce lookup overhead of the mapping table. However, DFTL incurs page mapping lookup overhead for the workloads with fewer temporal locality. In addition, it suffers from frequent updates in the pages storing the page mapping table in case of write intensive workloads and garbage collection.

Considering the advantages and disadvantages of the existing FTLs and the limited memory space, we have made the following observations: **(1)** Block level mapping has a good performance for read intensive data due to its fast direct address translations, **(2)** Page level mapping manages write intensive data well since its high block utilization and the reduced number of erase operations required, **(3)** Write intensive pages, which define as *hot* pages, will be benefited from the page level mapping, and **(4)** Read intensive pages, which we define as *cold* pages, will be benefited from the block level mapping. **(5)** Spatial locality in a workload can help to improve an FTL performance. Based on these observations, our goal is to design an FTL scheme that will follow the workload behavior. For a write intensive workload, it will provide the page level mapping-like performance, while for a read intensive workload, it will provide the block level mapping-like performance.

In this paper, we propose a novel FTL scheme named as CFTL, which stands for Convertible Flash Translation Layer. In CFTL, the core mapping table is a pure page level mapping. This mapping table is stored in the flash memory. The key idea is to force the page level mapping to provide performance comparable to a block level mapping, if the data is turned to cold (read intensive) from hot (write intensive) and vice versa. Since CFTL maintains core mapping table in the page level mapping, a logical page can be mapped to any physical page. Thus, to change from block level to page level mapping, no extra effort is needed. Contrarily, to switch from page level to block level mapping, we force CFTL to map consecutive logical pages to consecutive physical pages.

CFTL uses a simple hot/cold block detection algorithm for its addressing mode changes. If a logical block is identified as cold, then all consecutive logical pages in the block are stored in the consecutive physical pages of a new block. On the other hand, if logical block is identified as hot, CFTL does not force any restriction between the corresponding logical pages to physical page mapping for that block. According to the hotness of a block, the corresponding mapping is also dynamically changed. In CFTL, since mapping table is stored in the flash memory, there can be an overhead to lookup the mapping table. To speed up the mapping table lookup, CFTL maintains two mapping caches. First cache maintains the page level mappings, while second cache maintains block level mappings. These caches exploit both temporal and spatial localities to reduce the lookup overhead. The main contribution of this paper is as follows:

- ***A Convertible FTL Scheme:*** Unlike other existing FTLs, CFTL is adaptive to data access patterns. The main idea of CFTL is simple: a block level mapping deals with read intensive data to make the best of the fast direct address translation, and a page level mapping manages write intensive data to minimize erase operations thereby doing completely away with expensive full merge operations. Therefore, some parts of flash are addressed by a page level mapping, while some 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 underlying data.

- ***An Efficient Caching Strategy:*** For the fast address translation, CFTL employs two caches to store the mapping data. One cache is used to speed up the page level address translation, while another cache is used to speed up the block level address translation. In particular, the page level cache is specially designed to exploit the spatial locality of data. It uses hints to improve hit ratio. Consequently, by exploiting both temporal and spatial localities, these two caches make significant contribution to improve the overall performance thereby reducing address lookup overhead.

The remainder of this paper is organized as follows. Section 2 gives an overview of flash memory and describes existing FTL schemes. Section 3 explains the design and operations of CFTL scheme. Section 4 provides experimental results. Finally, Section 5 concludes the discussion.

2. BACKGROUND AND RELATED WORK

In this section, at first, we describe flash memory architecture. Next, we describe various address mapping schemes including page level, block level, and hybrid mappings.

2.1 Flash Memory Architecture

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 [19]. 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 1 gives an overview of an NAND flash-based storage device. In an NAND flash memory, data are organized as an array of blocks. Each block comprises either 32 pages (small block NAND flash) or 64 pages (large block NAND flash). In case of small block NAND flash, each page consists of sectors of 512 bytes and spare areas of 16 bytes. Each sector is employed to save data and each spare area is used to record LPN (Logical Page Number) or ECC (Error Correcting Code). Since NAND flash does not allow overwrite to the page already written, the whole block which the page belongs to must be erased before writing operation is allowed to the page due to the feature of Erase-before-Write in flash memory. In order to balance the number of erasures on physical blocks, an update to data in a logical page may migrate the updated data to a different physical page. Therefore, there must be a mapping between logical pages to physical pages. This mapping is managed by the Flash Translation Layer (FTL). FTL allows existing applications to use flash memory as a replacement for the magnetic disk without any code modifications. However, the performance of the flash memory is greatly impacted by mapping scheme used in FTL.

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 an empty location in order to avoid erase operations. After the overwrite operation, FTL updates the address mapping information and the outdated block can be erased later by a garbage collection process. FTL can maintains the mapping table information either in the page-level, or block-level, or in a hybrid manner. In addition, FTL can store mapping table either in the SRAM or in the

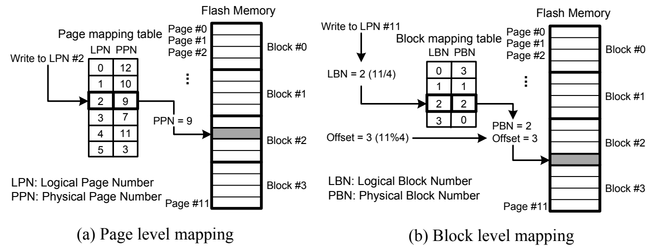


Figure 2: Page level and block level mapping

flash memory itself. In the rest of this section, first we describe SRAM based mapping schemes. Next, we describe flash memory based mapping schemes.

1) SRAM-based Mapping Table: According to the granularities with which the mapping information is managed, FTLs are largely classified into either a page level mapping [11] or a block level mapping [20] or hybrid.

• *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 2). In addition to this feature, since it does not require expensive full merge operations described in the next subsection, it shows a good overall performance for both read and write operations. This intrinsic merit, however, brings about its critical demerit–large size of memory requirements. That is, the size of mapping table may be too large to be resided in SRAM of FTL. For example, let us consider a flash memory of 4GB size and a page size of 2KB, this requires 2 million (2^{21}) numbers of page mapping information to be stored in SRAM. In this case, assuming each mapping entry needs 8bytes, 16MB memory space is required only for the mapping table. This may be infeasible for the economic reasons.

• *Block Level Mapping:* In a block level address mapping, a logical page address is made up of both a logical block number and its corresponding offset. Since the mapping table maintains only the mapping information between logical and physical blocks, the size of block mapping information is relatively smaller than that of a page level mapping. However, this approach also retains an inevitable disadvantage. When the overwrite operations to logical pages are issued, the corresponding block must be migrated and remapped to a free physical block as follows: The valid pages and the updated page of the original data block are copied to a new free physical block, and then the original physical block should 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 level and block level mapping approaches, a variety of hybrid schemes have been proposed [12, 13, 14, 15, 16, 17]. Most of these algorithms are based on a log buffer approach. In other words, there exists a page level mapping table for a limited number of blocks (log blocks) as well as a block level mapping table for the data blocks. The log blocks are used to temporarily record updates to 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 a merge operation. A merge operation can be expensive and cause a number of log block erasures.

Merge operations can be classified into three types: *switch merge*, *partial merge*, and *full merge* [18]. 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. Since this requires only one block erasure, this 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. Therefore, this costs additional page copies as well as one block erasure. A full merge requires the largest overhead among merge operations. An FTL allocates a free block and copies the all valid pages either from the data block or from the log block into the free block. After copying all the valid pages, the free block becomes the data block and the former data block and the log block are erased. Therefore, a single full merge operation requires as many read and write operations as the number of valid pages in a block and two erase operations [15].

A variety of hybrid mapping schemes have been proposed [12, 13, 14, 15, 16, 17]. We summarize core features of each scheme and discuss merits and demerits of each scheme.

BAST (Block Associative Sector Translation) [12] scheme classifies blocks into two types, namely, data blocks for data saving and log blocks for overwrite operations. Once an overwrite operation is issued, an empty log block is assigned and the data is written to it instead of direct calling a block erase operation which is very expensive. As a result of this, erase operation does 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 problem [14].

FAST (Fully Associative Sector Translation) [14] is based on BAST scheme but allows log blocks to be shared by all data blocks unlike BAST in which data blocks are associated to log blocks 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 accomplished better utilization of log blocks, it still remains in low 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 fully associative policy.

SuperBlock FTL [15] scheme attempts to exploit the block level spatial locality in workloads by allowing the page level mapping in a superblock which is a set of consecutive blocks. This separates hot data (frequently updated data) and non-hot data into different blocks within a superblock and consequently the garbage collection efficiency is achieved thereby reducing the number of full merge operations. However, this approach uses a three-level address translation mechanism which leads to multiple accesses of spare area to serve the requests. In addition, it also uses a fixed size of superblock explicitly required to be tuned according to workload requirements and does not efficiently make a distinction between cold and hot data.

LAST (Locality-Aware Sector Translation) [16] scheme adopts multiple sequential log blocks to make use of spatial localities in workload in order to supplement the limitations

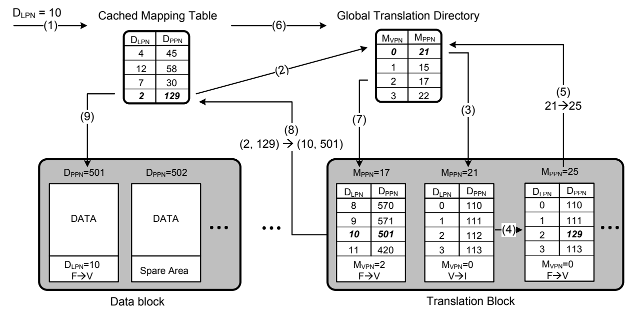


Figure 3: Address translation process of DFTL

of FAST. It classifies random log buffers into hot and cold partitions to alleviate full merge cost. LAST, as the 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 brings about the overall garbage collection overhead.

AFTL (Adaptive Two-Level Flash Translation Layer) [17] scheme maintains latest recently used mapping information with fine-grained address translation mechanism and the least recently used mapping information with coarse-grained mechanisms due to the limited source of the fine-grained slots. Notwithstanding this two-level management, even though there are the large amounts of hot data, they all cannot move to fine-grained slots due to the limited size of fine-grained mechanism. That is, coarse-to-fine switches incur corresponding fine-to-coarse switches, which causes overhead in valid data page copies. Additionally, only if all of the data in its primary block appear in the replacement block, both corresponding coarse-grained slot and its primary block can be removed, which leads to low block utilization.

2) Flash-based Mapping Table: Instead of storing the page mapping table in the SRAM, it can also be stored in the flash memory itself. Thus, the performance degradation due to expensive full merge operations experienced by the hybrid schemes can be avoided. To that end, Gupta et al. [18] proposed a two-tier, SRAM and Flash, pure page level mapping scheme called DFTL. Our proposed new mapping, CFTL (Convertible Flash Translation Layer), also stores the mapping table in the flash memory. Since DFTL is highly related to CFTL, we describe the DFTL scheme in detail.

• *DFTL Architecture:* DFTL maintains two types of tables in SRAM, namely, Cached Mapping Table (CMT) and Global Translation Directory (GTD). CMT stores only a small number of page mapping information like a cache for a fast address translation in SRAM. GTD keeps track of all scattered page mapping tables stored in flash since out-of-updates cause translation pages get physically dispersed over the entire flash memory. That is, GTD works like tier-1 mapping table in the two-tier address translation hierarchy. A complete translation pages are stored in flash due to their relatively big size. Translation pages retain the whole address mapping information and each page directly point to physical data pages in flash.

• *Address Translation Process in DFTL:* Figure 3 illustrates the address translation process of DFTL. If a read/write request hits the CMT in SRAM, it is served directly by using the mapping information stored in CMT.

Table 1: **Comparison between DFTL and CFTL**

	DFTL	CFTL
Mapping Table Stored in	Flash	Flash
Write Intensive Workload Performance	Good	Good
Read Intensive Workload Performance	Not Good	Good
Exploits Temporal Locality	Yes	Yes
Exploits Spatial Locality	No	Yes
Adaptiveness	No	Yes

Otherwise it goes through the two-tier address translation process and it may entail to choose a victim entry in CMT for eviction in case of the list full. If the victim information has been updated since it was loaded into it, DFTL checks GTD to find translation page including corresponding logical page address. Next, DFTL invalidates the translation page and assign one new page to reflect the updated mapping information. Then DFTL also needs to update GTD to reflect the newly assigned translation page. However, if the victim entry has not been updated since it was stored in CMT, the mapping information is simply removed from CMT. Now the incoming request is translated with the same processes above and the newly translated mapping information is stored in CMT. Finally, the requested operation is executed.

• *Advantages and Disadvantages of DFTL:* Since DFTL is a pure page level mapping scheme, not only does it achieve high block utilization, but also it completely remove full merge operations [18]. As a result, it improves overall performance and outperforms state-of-the-art hybrid FTLs in terms of write performance, block utilization, and the number of merge operations. However, DFTL suffers from frequent updates of translation pages in case of write dominant access patterns or garbage collection. To alleviate this problem, it uses delayed updates and batch updates in CMT with the aim of delaying the frequent updates. DFTL achieves a good write performance but cannot achieve as good read performance as hybrid FTLs under read dominant workloads due to its intrinsic two-tier address translation overhead. It costs an extra page read in flash when the request does not hit the CMT. Therefore DFTL cannot outperform hybrid mapping schemes using direct (i.e., one-level) address translation especially under randomly read intensive environments. DTFL considers temporal locality but leaves spatial locality unaccounted. In many cases, spatial locality is also an essential factor to efficiently access data [21].

Our novel mapping scheme, CFTL, addresses the shortcomings of the DFTL scheme. Table 1 provides a comparison between CFTL and DFTL schemes.

3. 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 so that it can overcome the innate limitations of existing FTLs described in Section 2.2. In addition, CFTL uses two mapping caches to speed up the address lookup performance. The rest of this section is organized as follows. Section 3.1 depicts the CFTL architecture and section 3.2 explains the addressing mode change scheme. The address translation process in CFTL is described in section 3.3 and section 3.4 describes how to efficiently manage mapping caches. Finally, Section 3.5 discusses the advantages of CFTL compared to other existing FTL schemes.

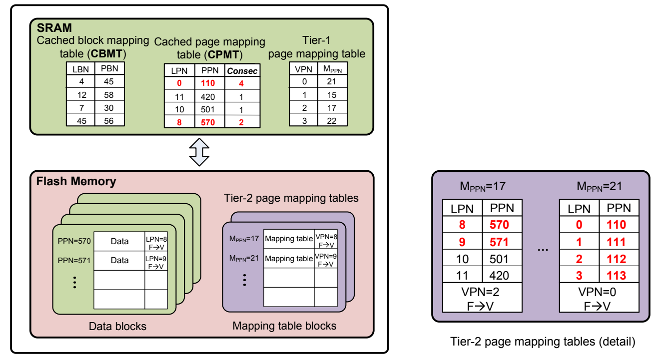


Figure 4: CFTL Architecture

3.1 Architecture

Figure 4 gives an overview of the CFTL design. CFTL entire page mapping table in the flash memory. We define it as *tier-2 page mapping table*. As mapping table is stored in the flash memory, therefore during the address lookup, first we need to read page mapping table to find the location of the orginal data. Next, from that location, we can find the data. Clearly, 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, 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 and served as a cache to make the best of temporal locality and spatial locality in a page level mapping. This table has an addition to a general address mapping table called *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 later in subsection 3.4. CBMT is also a small amount of a block mapping table and serves like a cache as well 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 data blocks in flash in conjunction with page offsets.

In addition to both CBMT and CPMT (as shown in Figure 4), there exists another mapping table. We define this SRAM based mapping table as *tier-1 page mapping table*. A tier-1 page mapping 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 on flash memory due to its large size limitation. Since tier-2 mapping tables compose the whole 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 each 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 reflecting the updated mapping information. This arises from such a peculiarity that both reads and writes to flash memory are performed in terms of pages. Thus we need to maintain a tier-1 mapping table so as to trace each tier-2 mapping table which can be scattered over the flash memory whenever it is updated. Actually each page (one tier-2 mapping table) can store 512 page mapping entries. For clarification, assuming one page size is 2KB and 4bytes are required to address entire flash memory space, then 2^9 (2KB/4bytes) logically consecutive address mapping entries can be saved for each data page. Therefore, 1GB flash memory device needs only 2MB ($2^{10}\times$2KB) space for all tier-2 mapping tables in flash: it totally requires 2^{10} (1GB/1MB per page) number of tier-2 mapping tables. Apart from these small amounts of tier-2 page mapping tables, all the other spaces are for data blocks to store real data in flash memory.

3.2 Addressing Mode Changes

The core feature of CFTL is that it is dynamically converted to either a page level mapping or a block level mapping to follow workload characteristics. Therefore, when and how CFTL converts its addressing mode are of paramount importance. In this subsection, we describe the addressing mode change policy in CFTL.

When any data block is frequently updated, we define it as *hot* data. On the other hand, it is accessed in a read dominant manner or has not updated for a long time, we define it as *cold* data. In CFTL scheme, hot and cold data identification plays an important role in decision making on mode changes from a block mapping to page mapping or vice versa. As a hot/cold identification algorithm in CFTL, we employ a simple counting method. In this approach, we maintain a counter for every logical page address and each time the logical page address is accessed, the corresponding counter is incremented. If the counter value for a particular logical address is more than a specified value (hot threshold value), we identify the data block as a hot block. Hot and cold data identification itself, however, is out of scope of this paper, other algorithms such as a hash-based technique [22] or LRU discipline [23] can also replace it.

• *Page to Block Level Mapping:* If a hot/cold data identifier in CFTL identifies some data as cold data, addressing mode of those data is switched to a block level mapping during garbage collection time. In particular, when the cold data pages in a logical block are physically dissipated in flash, we need to collect those pages into a new physical data block consecutively. Then we can register this block mapping information into CBMT for a block level mapping. Contrastingly, 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 such data that have not updated for a long time, CFTL does not convert such type of cold data blocks to a block level mapping. It converts only cold data blocks that frequently accessed in a read manner to a block level mapping.

• *Block to Page Level Mapping:* In case of write dominant access patterns, a block level mapping scheme does not demonstrate a good performance because frequent updates indispensably lead to expensive full merge operations. This inevitable full merge operation is the main cause to degrade, in particular, write performance in a block level mapping scheme including existing hybrid FTLs. Thus CFTL maintains write intensive data with a page level mapping schemes. On the contrary to the mode change from a page to a block level mapping, this mode switch originally does not require any extra cost because a page mapping table is always valid to all data in flash. Therefore, when a hot/cold data identifier in CFTL identifies some data in flash as hot data, CFTL only has to remove corresponding mapping entries from CBMT. Then those data can be accessed only by page mapping tables, not block mapping tables any more.

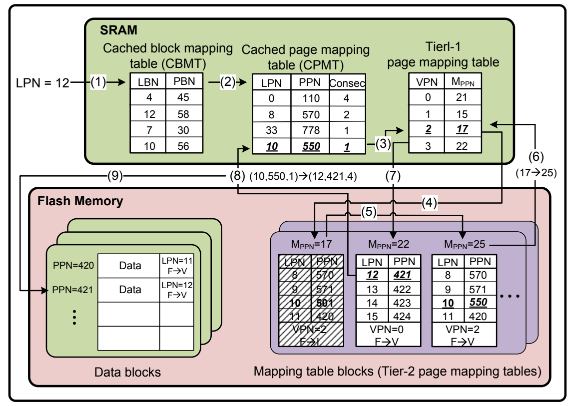


Figure 5: Address translation process of CFTL

3.3 Address Translation Process

Figure 5 describes the address translation process of CFTL for serving requests that are composed of read or write operations. As described in subsection 3.1, since both CPMT and CBMT in SRAM are served as a cache, if the requested mapping information for read or write requests is already stored in both tables in SRAM, those requests can be directly served with that information. Otherwise, CFTL fetches the corresponding mapping information into the tables from flash by using both tier-1 and tier-2 page mapping table. At this time we may need to choose a victim entry from those tables if the corresponding mapping tables are fully filled with the fetched mapping information. As an entry replacement algorithm, CFTL adopts Least Frequently Used (LFU) cache algorithm [24]. 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, we need to reflect the updated mapping information in CPMT to tier-1 and tier-2 page mapping table because CFTL uses delayed updates. Thus 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 5). Finally the corresponding tier-1 mapping table is updated to reflect the new tier-2 page mapping table (step 6 in Figure 5). Now the request is ready to be served with the fetched mapping information.

Algorithm: Algorithm 1 describes the address translation process in CFTL. If a request is issued from the system, CFTL first checks CBMT, and then CPMT. If it does not hit both CBMT and CPMT, CFTL follows regular address translation process by using both tier-1 and tier-2 page mapping tables.

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 1). In this case, if the request entails write operation in flash, we update the corresponding mapping information in CPMT (Line 8 in Algorithm 1). 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 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 (Lines 13-20 in Algorithm 1).

Third, if the request, however, does not hit CPMT as well as CBMT, CFTL follows its regular two-tier address translation process. CFTL first checks whether CPMT is full of mapping entries or not. In case that CPMT has any space available to store mapping information, the new mapping information can take the free slot in CPMT (Lines 35-39 in Algorithm 1). On the other hand, if CPMT is full of mapping entries, CFTL chooses a victim entry to evict it from CPMT. 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 to prevent frequent update of tier-2 page mapping tables. At this moment, we can get the PPN storing the corresponding tier-2 page mapping table from tier-1 page mapping 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 also need to update the corresponding tier-1 mapping table entry to reflect the newly assigned PPN of the new tier-2 mapping table (Lines 25-31 in Algorithm 1). Up to this point, 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.

Algorithm 1 CFTL Address Translation Algorithm

Function AddrTrans(*Request, RequestLPN*)

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

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, namely, Cached Page Mapping Table (CPMT) and Cached Block Mapping Table (CBMT). Although these are address mapping tables, they are served as a cache for a fast address translation. CPMT is employed to speed up the page level address translation, while CBMT is for the fast block level address translation. As shown in Figure 4, in addition to the existing logical to physical address mapping fields in CPMT, CFTL adds one more field named *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. Then it reflects the information on CPMT at the time it updates CPMT.

As a clear example, according to the CPMT in Figure 4, LPN = 0 corresponds to PPN = 110. Moreover, the consecutive field hints that 4 numbers of PPN from PPN = 110 are consecutive, namely, 110, 111, 112, and 113. These physical addresses also correspond to 4 logical addresses respectively 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 out of the consecutive pages is updated, we need to maintain both sides of consecutive pages because all corresponding pages are not consecutive any more due to the updated page. In this case, we divide the corresponding mapping information into two consecutive parts and update each mapping information accordingly. Algorithm 2 describes this algorithm especially regarding CPMT management.

In summary, 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 even with the same number of mapping table entries in the cache, which results in higher overall performance.

Algorithm: Algorithm 2 gives the pseudocode to manage CPMT containing consecutive field. When CFTL gets *Re-*

questLPN (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 2), *RequestLPN* can hit the CPMT even though there does not exist the exact mapping information in CPMT. CFTL first calculates the difference between *RequestLPN* and the corresponding *LPN* in CPMT. Then it can finally find out *PPN* (Physical Page Number) just by adding the difference to *PPN* in the corresponding CPMT entry (Line 4 in Algorithm 2). If the operation (Type of request) is *WRITE*, CFTL needs to carefully update the corresponding mapping information in CPMT because *WRITE* operation causes page update, which produces an effect on consecutive value in the corresponding mapping entry. The update process is as follows. First of all, CFTL updates the original consecutive value to difference value it already calculated. Next, it saves the new mapping information with *RequestLPN* and new *PPN* (Lines 5-10 in Algorithm 2). If the *the RequestLPN* is not the end of 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 consecutive value to *Consecutive–Diff–1*.

For clarification, a mapping entry in CPMT assumes the form of *(LPN, PPN, Consecutive)*. We additionally assume that *RequestLPN* is 105 and the mapping entry corresponds to *(100, 200, 10)*. Then 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 thereby 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 in between 100 and 109. Then, CFTL can derive the requested *PPN* (205) by adding 5 (*Diff*) to 200 (*PPN*). However, if this request is *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, then CFTL 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)*.

Algorithm 2 Cache Manangement in CFTL

```
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 or three entries. Then they will be updated or newly stored into CPMT accordingly
7:       Update Consecutive value of LPN in CPMT = Diff
8:       Store to CPMT with RequestLPN and new PPN
9:       if (Consecutive value of LPN–Diff–1 > 0) then
10:        // if the hitting Request LPN is not the end of consecutive addresses of LPN
11:        Store to CPMT with Request LPN+1, PPN+Diff+1, and Consecutive–Diff–1
12:      end if
13:    end if
14:  end if
15: end for
```

3.5 Discussions

In this subsection, we discuss the advantages of CFTL compared to other existing FTL schemes in several respects.

- *Read Performance:* DFTL shows a good read performance under the condition with a high temporal locality. However, under totally random read intensive patterns (i.e., low temporal locality), DFTL inevitably causes many cache misses in SRAM, which is the root cause to degrade 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 and accessed by the block mapping manner. Moreover, its efficient caching strategy improves its read performance further.
- *Temporal and Spatial Localities:* When it comes to data access, both temporal and spatial localities play a very important role in data access performance [21]. DFTL takes a temporal locality into consideration by residing a cached mapping table in SRAM but leaves spatial locality unaccounted. On the other hands, hybrid FTLs like superblock FTL [15] and LAST [16] take both localities into account. In addition to these schemes, CFTL also considers both thereby putting both CBMT and CPMT into SRAM.
- *Full Merge Operations:* Since CFTL is fundamentally based on a two-tier page level mapping like DFTL, this eradicates expensive full merge operations. However, existing hybrid FTLs are stick to the inborn limitations of a block level mapping. Even though each hybrid approach tries to adopt a page level mapping scheme, they are restricted only to a small amount of log blocks. Thus this kind of approach cannot be a basic solution to remove full merge operations but just to delay them.
- *Block Utilization:* Existing hybrid FTLs maintain relatively a small amount of log blocks to serve update requests. These ultimately lead to low block utilization. So even though there are many unoccupied pages in data blocks, garbage collection can be unnecessarily triggered to reclaim them. On the other hands, DFTL and CFTL based on a page level mapping scheme, resolve this low block utilization problem because updated data can be placed into any of the data block in flash.
- *Write Performance:* As far as hybrid FTLs maintain log blocks, they cannot be free from a low write performance. Many random write operations inevitably cause many full merge operations to them, which ultimately results in low 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 like a page level mapping scheme because all data in CFTL is

Table 2: Simulation parameters

Parameters	Values
Page Read Speed	25μs
Page Write Speed	200μs
Block Erase Speed	1.5ms
Page Size	2KB
Block Size	128KB
Entries in Mapping Tables	4,096 entries

Table 3: 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

fundamentally managed by a two-tier pure page level mapping and a block level mapping approach is adopted only for read intensive data for the better performance. Thus both CFTL and DFTL achieve good write performance.

4. EXPERIMENTAL RESULTS

There exist several factors that affect FTL performance such as the number of merge operations and block erasures performed, address translation time, SRAM size, and so forth. Although each factor has its own significance to FTL performance, FTL scheme exhibits its good performance only when all those factors are well harmonized. We, therefore, choose an average response time to compare each performance of a diverse set of FTL schemes. 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. We also make an attempt to compare the overall performance of CFTL with and without an efficient caching strategy in order to demonstrate the efficiency of our proposed caching strategy. Finally, memory requirements are another important factor to be discussed since SRAM size is very limited in flash memory.

4.1 Evaluation Setup

We simulate a 32GB NAND flash memory with configurations shown in Table 2. Our experiments of flash memory are based on the latest product specification of Samsung's K9XXG08UXM series NAND flash part [25][7]. We consider only a part of flash memory storing our test workloads for equitable comparison with other schemes. We additionally assume that the remainder of the flash memory is free or cold blocks which are not taken into account for this experiment. For more objective evaluation, various types of workloads including real trace data sets are selected (Table 3). Websearch3 [26] trace made by Storage Performance Council (SPC) [27] reflects well read intensive I/O trace. As a write intensive trace, we employ Financial1 [28] made from an OLTP application running at a financial institution. For the totally random performance measurements, we based random traces upon Financial2 [28] which is also made from an OLTP application. Three types of random trace workloads-read intensive, half and half, and write intensive workload-are employed for more complete and objective experiments of the random access performance.

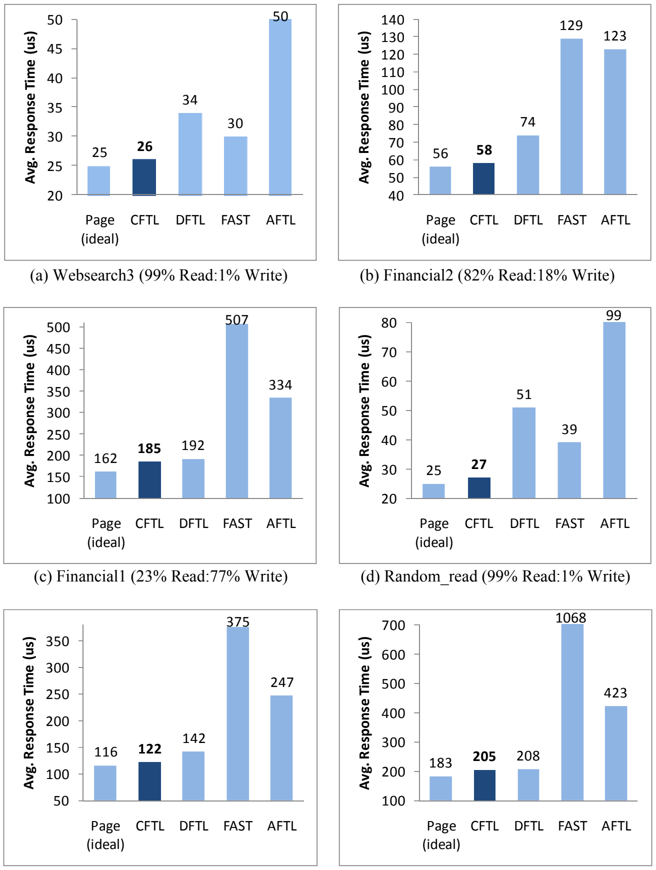
4.2 Results and Analysis

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: Figure 6 illustrates overall performance of diverse FTLs under a variety of workloads. We measure not only overhead of garbage collection and address translation time but also system service time in order to evaluate overall performance including both read and write performance. Afterwards we will call the summation of these average response time. An ideal pure page level mapping is selected as our baseline scheme.

As shown in Figure 6, overall performance of CFTL is very close to that of an ideal page level mapping. That is, CFTL outperforms all the other FTLs in the lists 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, in particular, not only its better read performance against FAST [14] having strong point in read performance, but also write performance against DFTL [18] displaying the excellent write performance.

Figure 7 depicts the overall performance change of each scheme as time flows and enables us to analyze these results in more detail. Under the read intensive access patterns (Figure 6(a) and 7(a), Figure 6(b) and 7(b), and Figure 6(d) and 7(d)), CFTL switches more and more data to a block level mapping as time goes by in order to make the best of its fast direct address translation. Moreover, the efficient caching strategy makes a considerable effect on read performance since almost 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 6(d), our proposed caching strategy exhibits its respectable effectiveness in read performance especially under randomly read intensive workloads. Compared to Figure 6(a), we can identify the random read performance of most of the FTLs is significantly degraded. However, CFTL still shows its good random read performance due to the efficient caching strategy. We will discuss this in more detail in the next subsection. As we expected, FAST which is hybrid mapping but primarily based on a block level mapping also shows a good performance under the read intensive workloads. However, it does not reach CFTL because of its intrinsic limitation such as merge operations even though there are not so many update operations in this workload. In addition to the expensive merge operations in FAST, extra read cost in log blocks is also another factor to deteriorate its read performance. DFTL is a two-tier pure page level mapping. If, however, the workload does not contain a high temporal locality, it frequently requires an additional overhead in address translation. This is the main cause that DFTL does not exhibit relatively good random read performance (Fig-



(a) Websearch3 (99% Read:1% Write)

(b) Financial2 (82% Read:18% Write)

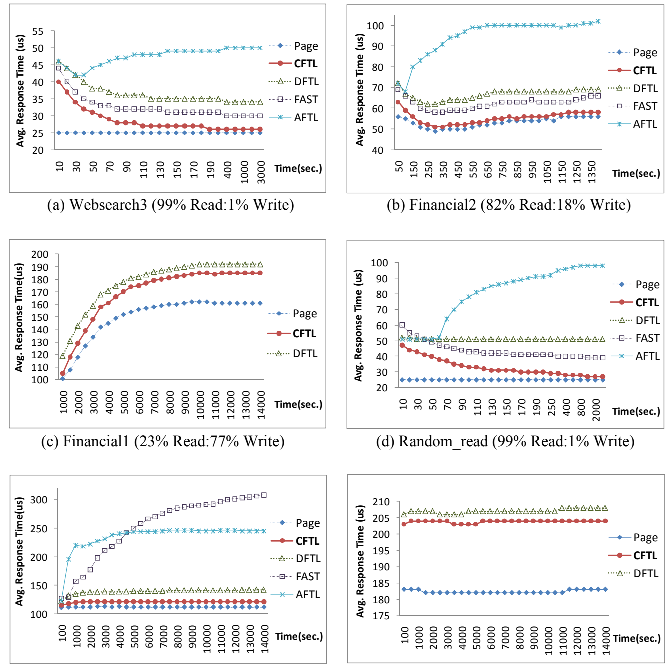
(c) Financial1 (23% Read:77% Write)

(d) Random_read (99% Read:1% Write)

(e) Random_even (50% Read:50% Write)

(f) Random_write (10% Read:90% Write)

Figure 6: Overall read and write performance under various workloads



(a) Websearch3 (99% Read:1% Write)

(b) Financial2 (82% Read:18% Write)

(c) Financial1 (23% Read:77% Write)

(d) Random_read (99% Read:1% Write)

(e) Random_even (50% Read:50% Write)

(f) Random_write (10% Read:90% Write)

Figure 7: Performance change in the time flow. Here, *page* stands for ideal page level mapping.

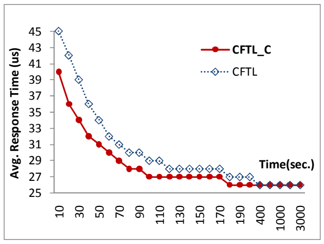
ure 6(d) and 7(d)). Unlike these three schemes, AFTL [17] does not show a good overall performance in read intensive workload because when fine-grained slots are full of mapping information, it starts to cause valid data page copies for a coarse-grained mapping. This brings about severe overhead so that it degrades its overall performance.

As the number of write requests grows in a workload, performance variations among each scheme are also increased rapidly since frequent write operations cause frequent updates in flash. As shown in Figure 6(c) and 7(c), Figure 6(e) and 7(e), and Figure 6(f) and 7(f), the overall performances of AFTL and FAST are severely degraded under half and half workload, not to mention write intensive workload. Thus we eliminate both schemes in Figure 7(c) and 7(f).

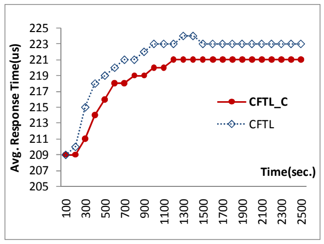
This fundamentally stems from frequent erase operations in both schemes. That is to say, frequent updates cause frequent merge operations in FAST and frequent both primary and replacement block erases in AFTL. However, since DFTL is a two-tier pure page level mapping and CFTL is fundamentally based on a page level mapping, both schemes shows a considerably better overall performance than AFTL as well as FAST, in particular, under write dominant access patterns (Figure 6(c) and 7(c) and Figure 6(f) and 7(f)). However, CFTL shows the 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 of switch from a block level mapping to a page level mapping and does not maintain log blocks under a block level mapping.

In summary, CFTL is adaptive to data access patterns to fully exploit the benefits of a page level mapping and a block level mapping whereby it achieves a good read performance as well as write performance under read intensive, write intensive, and totally random access patterns. These simulation results demonstrate well our assertion.

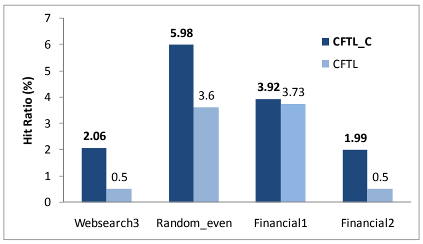
2) An Efficient Caching Strategy: CFTL maintains Cached Page Mapping Table (CPMT) in SRAM the same as Cached Mapping Table (CMT) in DFTL [18]. However, CPMT retains one more field called *consecutive field*. This simple field helps improve not only read performance but write performance in CFTL by judiciously exploiting spatial locality. As shown in Figure 8, our proposed caching strategy is effective to improve overall performance of CFTL further. Due to this simple field, we considerably ameliorate the hit ratio of CPMT even though the request does not hit the cache (Shown in Figure 8(c)). This yields a faster address translation and ultimately wields great influence on the enhancement of overall performance. Figure 8(c) supports our assertion on effectiveness of our proposed caching strategy. It illustrates the request hit ratios in CPMT under several workloads. As shown in the Figure 8(c), CFTL_C (CFTL with the efficient caching strategy) exhibits its dominant request hit count against CFTL (CFTL without the strategy) especially under read intensive workloads (Websearch3 and Financial2) since read operations does not affect the consecutiveness of data pages in a block. On the other hand, as write requests grow like Random_even (50% Read:50% Write) and Financial1 (23% Read:77% Write), the frequent updates hurt the consecutiveness so that the variation of hit



(a) Read performance



(b) Write performance



(c) Request hit ratios in CPMT

Figure 8: Performance improvement with efficient caching strategy

counts between CFTL_C and CFTL is relatively reduced.

Figure 8(a) illustrates that both schemes converge on the same average response time as time goes by. In other words, even though the CFTL does not have the efficient caching strategy, its read performance gradually approaches to that of CFTL_C as more and more requests come in. In our experiment, both read performances converge on the same performance after around 500 seconds corresponding to 104,420 intensive read requests in this workload. Note that the convergence time totally depends on the types of workloads. In other words, most of the read intensive data switch to a block level mapping so that the read performance of CFTL gradually approaches to that of CFTL_C after some time. This proves the adaptive feature of CFTL. However, CFTL_C exhibits its better read performance even before the read intensive data are converted to a block level mapping because it can considerably improve the hit ratio of CPMT even with the same number of mapping table entries in SRAM. This is the unique point of our proposed efficient caching strategy in CFTL scheme.

3) Memory Space Requirements: Most of the FTLs maintain their own address mapping information in SRAM for a faster address translation. Since SRAM size, however, is very limited to flash memory, memory space requirements are also another momentous factor among diverse FTL schemes. For simplification, we assume entire flash size is 4GB and each mapping table in SRAM consists of 2,048 mapping entries. We additionally assume that approximately 3% of the entire space is allocated for log blocks in hybrid FTLs (this is based on [15]).

An ideal page level mapping requires 8MB for its complete page mapping table. This is exceptionally large space compared to the other schemes; so we exclude this in Figure 9. Both AFTL [17] and FAST [14] also consume a lot of memory space. Almost over 90% of total memory requirements in AFTL are assigned for coarse-grained slots and almost of them in FAST are allocated for page level mapping tables. On the contrary, CFTL and DFTL [18] consume only about

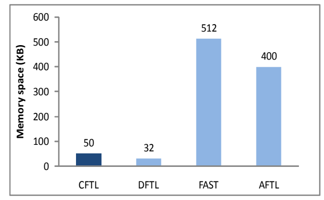


Figure 9: Memory space requirements

10% of the total memory space in FAST and AFTL since complete page mapping tables are stored in flash memory not in SRAM. CFTL consumes a little more memory than DFTL since it adds Consecutive Field in CPMT for a more efficient caching strategy and maintains one additional mapping table called CBMT for a block mapping which does not exist in DFTL. However, this extra small amount of memory empowers CFTL to make the best of a page level mapping and a block level mapping, with which CFTL achieves a good read performance as well as good write performance.

5. CONCLUSION

In this paper, we propose a novel FTL scheme named Convertible Flash Translation Layer (CFTL, for short) for flash memory 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. CFTL stores entire mapping table in the flash memory. Thus, there is an overhead to lookup the mapping table. To remedy this problem, we also propose an efficient caching strategy which improves the mapping table lookup performance of CFTL by leveraging spatial locality as well as temporal locality.

Our experimental results show that for the real-life read intensive workloads CFTL outperforms DFTL (which is the state-of-the-art FTL scheme) [18] by up to 24%, and for random read intensive workload CFTL outperforms DFTL by up to 47%, while for the real-life write intensive workloads CFTL outperforms DFTL by up to 4%. Our experiments also show about new caching strategy improves cache hit ratio by up to 400% for the read intensive workloads.

6. REFERENCES

[1] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," in *ACM Computing Surveys*, vol. 37, no. 2, 2005.

[2] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song, "A Log Buffer-Based Flash Translation Layer Using Fully-Associate Sector Translation," in *ACM Transactions on Embedded Computing System*, vol. 6, no. 3, 2007.

[3] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *FAST*, 2008.

[4] S. Nath and P. Gibbons, "Online Maintenance of Very Large Random Samples on Flash Storage," in *VLDB*, 2008.

[5] S. Lee and B. Moon, "Design of Flash-based DBMS: an In-page Logging Approach," in *SIGMOD*, 2007.

[6] S. Lee, B. Moon, C. Park, J. Kim, and S. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," in *SIGMOD*, 2008.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy, “Design Tradeoffs for SSD Performance,” in *Usenix*, 2008.

[8] M. Moshayedi and P. Wilkison, “Enterprise SSDs,” *ACM Queue*, vol. 6, no. 4, 2008.

[9] A. Leventhal, “Flash Storage Today,” *Queue*, vol. 6, no. 4, 2008.

[10] A. Caulfield, L. Grupp, and S. Swanson, “Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications,” in *ASPLOS*, 2009.

[11] CompactFlashAssociation, “http://www.compactflash.org.”

[12] Jesung Kim and Jong Min Kim and Noh, S.H. and Sang Lyul Min and Yookun Cho, “A space-efficient flash translation layer for compactflash systems,” *IEEE Transactions on Consumer Electronics*, 2002.

[13] 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, 2009.

[14] 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.

[15] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, “A Superblock-based Flash Translation Layer for NAND Flash Memory,” in *EMSOFT*, 2006.

[16] 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, 2008.

[17] C.-H. Wu and T.-W. Kuo, “An adaptive two-level management for the flash translation layer in eembedded systems,” in *ICCAD*, 2006.

[18] A. Gupta, Y. Kim, and B. Urgaonkar, “Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings,” in *ASPLOS*, 2009.

[19] M-systems, “Two Technologies Compared: NOR vs. NAND,” White Paper, July 2003.

[20] A. Ban, “Flash File System,” United States of America Patent 5 404 485, April 4, 1995.

[21] L.-P. Chang and T.-W. Kuo, “Efficient Management for Large-scale Flash Memory Storage Systems with Resource Conservation,” in *ACM Transactions on Storage*, vol. 1, no. 4, 2005.

[22] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, “Efficient Identification of Hot Data for Flash Memory Storage Systems,” *Trans. Storage*, vol. 2, no. 1, 2006.

[23] L.-P. Chang and T.-W. Kuo, “An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems,” in *RTAS*, 2002.

[24] R. Karedla, J. S. Love, and B. G. Wherry, “Caching Strategies to Improve Disk System Performance,” *Computer*, vol. 27, no. 3, 1994.

[25] SamsungElectronics, “K9XXG08XXM Flash Memory Specification,” 2007.

[26] “Websearch Trace from UMass Trace Repository,” http: //traces.cs.umass.edu/index.php/Storage/Storage.

[27] “SPC: Storage Performance Council,” http://www.storageperformance.org.

[28] “OLTP Trace from UMass Trace Repository,” http: //traces.cs.umass.edu/index.php/Storage/Storage.