Kim Y, Gupta A, Urgaonkar B. A temporal locality-aware page-mapped flash translation layer. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 28(6): 1025–1044 Nov. 2013. DOI 10.1007/s11390-013-1395-4

# A Temporal Locality-Aware Page-Mapped Flash Translation Layer

Youngjae Kim[1], Aayush Gupta[2], and Bhuvan Urgaonkar[3], *Senior Member, ACM, IEEE*

[1]*National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN 37831, U.S.A.*

[2]*IBM Almaden Research Center, San Jose, CA 95120, U.S.A.*

[3]*Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802, U.S.A.*

E-mail: kimy1@ornl.gov; guptaaa@us.ibm.com; bhuvan@cse.psu.edu

Received January 3, 2013; revised May 16, 2013.

**Abstract** The poor performance of random writes has been a cause of major concern which needs to be addressed to better utilize the potential of flash in enterprise-scale environments. We examine one of the important causes of this poor performance: the design of the flash translation layer (FTL) which performs the virtual-to-physical address translations and hides the erase-before-write characteristics of flash. We propose a complete paradigm shift in the design of the core FTL engine from the existing techniques with our Demand-Based Flash Translation Layer (DFTL) which selectively caches page-level address mappings. Our experimental evaluation using *FlashSim* with realistic enterprise-scale workloads endorses the utility of DFTL in enterprise-scale storage systems by demonstrating: 1) improved performance, 2) reduced garbage collection overhead and 3) better overload behavior compared with hybrid FTL schemes which are the most popular implementation methods. For example, a predominantly random-write dominant I/O trace from an OLTP application running at a large financial institution shows a 78% improvement in average response time (due to a 3-fold reduction in operations of the garbage collector), compared with the hybrid FTL scheme. Even for the well-known read-dominant TPC-H benchmark, for which DFTL introduces additional overheads, we improve system response time by 56%. Moreover, interestingly, when write-back cache on DFTL-based SSD is enabled, DFTL even outperforms the page-based FTL scheme, improving their response time by 72% in Financial trace.

**Keywords** flash memory, flash translation layer, storage system

## 1 Introduction

Hard disk drives have been the preferred media for data storage in enterprise-scale storage systems for several decades. The disk storage market totals approximately US$34 billion annually and is continually on the rise. Manufacturers of hard disk drives have been successful in ensuring sustained performance improvements while substantially bringing down the price-per-byte. However, there are several shortcomings inherent to hard disks that are becoming harder to overcome as we move into faster and denser design regimes. First, designers of hard disks are finding it increasingly difficult to further improve the revolutions per minute (RPM) (and hence the internal data transfer rate (IDR)) due to problems of dealing with the resulting increase in power consumption and temperature[1-2]. Second, any further improvement in storage density is increasingly harder to achieve and requires significant technological breakthroughs such as perpendicular recording[3-4]. Third, despite a variety of techniques employing caching, pre-fetching, scheduling, write-buffering, and those based on improving parallelism via replication (e.g., RAID (redundant array of independent disk)), the mechanical movement involved in the operation of hard disks implies that the performance of disk-based systems remains extremely sensitive to workload characteristics. Hard disks are significantly faster for sequential accesses than for random accesses — the IDR reflects and the gap continues to grow. This can severely limit the performance that hard disk based systems are able to offer to workloads with significant random access component or lack of locality. In an enterprise-scale system, consolidation can result in the multiplexing of unrelated workloads imparting randomness to their aggregate traffic[5].

Alongside improvements in disk technology, significant advances have also been made in various forms of solid-state memory such as NAND flash, magnetic

Regular Paper

This research was funded in part by the Natural Science Foundation of U.S. under Grant Nos. CCF-0811670, CNS-0720456, a gift from Cisco System, Inc. and partially through the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

The preliminary version of the paper was published in the Proceedings of the 14th ASPLOS.

RAM (MRAM)[6], phase-change memory (PCM)[①], and Ferroelectric RAM (FRAM)[7]. Solid-state memory offers several advantages over hard disks: lower and more predictable access latencies for random requests, smaller form factors, lower power consumption, lack of noise, and higher robustness to vibrations and temperature. In particular, recent improvements in the design and performance of NAND flash memory (simply *flash* henceforth) have resulted in it being employed in many embedded and consumer devices. Small form-factor hard disks have already been replaced by flash memory in some consumer devices like music players, PDAs, digital cameras.

The cost-per-byte for hard disks remains an order of magnitude lower than for flash memory and disks are likely to maintain this advantage in the foreseeable future. At the same time, however, flash devices are significantly cheaper than main memory technologies that play a crucial role in improving the performance of disk-based systems via caching and buffering. Furthermore, as an optimistic trend, their price-per-byte is falling[8], which leads us to believe that flash devices would be an integral component of future enterprise-scale storage systems. This trend is already evident as major storage vendors have started producing flash-based large-scale storage systems such as RamSan-500 from Texas Memory Systems, Symmetrix DMX-4 from EMC, and so on.

Before enterprise-scale systems can transition to employing flash-based devices at a large-scale, certain challenges must be addressed. SSDs have longevity and reliability concerns in particular for write intensive workloads because of the lifetime issues of NAND flash chips. Upon replacing hard disks with flash, certain managers of enterprise-scale applications are finding results that point to degraded performance. For example, the flash-based devices can be slow down for workloads with random writes[9-10]. Recent research has focused on improving random write performance of flash by adding DRAM-backed buffers[8] or buffering requests to increase their sequentiality[10]. However, we focus on an intrinsic component of the flash, namely the flash translation layer (FTL) to provide a solution for this poor performance.

The FTL is one of the core engines in flash-based SSDs that maintains a mapping table of virtual addresses from upper layers (e.g., those coming from file systems) to physical addresses on the flash. It helps to emulate the functionality of a normal block device by exposing only read/write operations to the upper software layers and by hiding the presence of erase operations, something unique to flash-based systems. Flash-based systems possess an asymmetry in how they can read and write. While a flash device can read any of its pages (a unit of read/write), it may only write to one that is in a special state called erased. This results in an important idiosyncrasy of updates in flash. Clearly, in-place updates would require an erase-per-update, causing performance to degrade. To get around this, FTLs implement out-of-place updates. An out-of-place update updates bring about the need for the FTL to employ a garbage collection (GC) mechanism. The role of the GC is to reclaim invalid pages within blocks by erasing the blocks (and if needed relocating any valid pages within them to new locations).

One of the main difficulties the FTL faces in ensuring high performance is the severely constrained size of the on-flash SRAM (static random-access memory)-based cache where it stores its mapping table. With the growing size of SSDs, this SRAM size is unlikely to scale proportionally due to the higher price/byte of SRAM. This prohibits FTLs from keeping virtual-to-physical address mappings for all pages on flash (page-level mapping). On the other hand, a block-level mapping, can lead to increased: 1) space wastage (due to internal fragmentation) and 2) performance degradation (due to GC-induced overheads). Furthermore, the specification for large-block flash devices (which are the norm today) requires sequential programming within the block[11] making such coarse-grained mapping infeasible. To counter these difficulties, state-of-the-art hybrid FTLs take the middle approach of using a hybrid of page-level and block-level mappings and are primarily based on the following main idea (we explain the intricacies of individual FTLs in Section 2): most of the blocks (called data blocks) are mapped at the block level, while a small number of blocks called "update" blocks are mapped at the page level and are used for recording updates to pages in the data blocks.

As we will argue in this paper, various variants of hybrid FTL fail to offer good enough performance for enterprise-scale workloads. As a motivational illustration, Fig.1 compares the performance of a hybrid FTL called FAST[12] with an idealized page-level mapping scheme with sufficient flash-based SRAM.

First, these hybrid schemes suffer from poor garbage collection behavior. Second, they often come with a number of workload-specific tunable parameters (for optimizing performance) that may be hard to set. Finally and most importantly, they do not properly exploit the temporal locality in accesses that most enterprise-scale workloads are known to exhibit.

[①]Intel, STMicroelectronics Deliver Industry's First Phase Change Memory Prototypes, 2008. http://www.intel.com/pressroom/archive/releases/2008/20080206corp.htm, Oct. 2013.

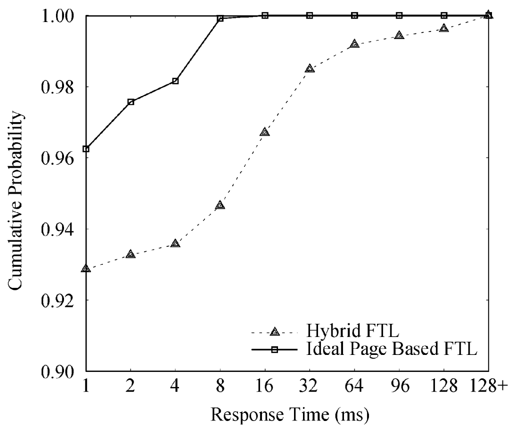


Fig.1. Comparison of the performance of a Financial trace employing an idealized page-level (assuming enough on-flash SRAM, an idealization), and a hybrid FTL scheme.

Fig.2 shows the extremely high temporal locality exhibited by two well-regarded workloads. Even the small SRAM available on flash devices can thus effectively store the mappings in use at a given time while the rest could be stored on the flash device itself.

Our thesis is that such a page-level FTL, based purely on exploiting such temporal locality, can outperform hybrid FTL schemes and also provide an easier-to-implement solution devoid of complicated tunable parameters. The specific contributions are list as follows.

We propose and design a novel flash translation layer called DFTL. Unlike currently predominant hybrid FTLs, it is purely page-mapped. The idea behind DFTL is simple: since most enterprise-scale workloads exhibit significant temporal locality, DFTL uses the on-flash limited SRAM to store the most popular (specifically, most recently used) mappings while the rest are maintained on the flash device itself.

Using an open-source flash simulator called FlashSim[13], we evaluate the efficacy of DFTL and compare it with other FTL schemes. FlashSim simulates the flash memory, controller, caches, device drivers and various interconnects. Using a number of realistic enterprise-scale workloads, we demonstrate the improved performance resulting from DFTL. As illustrative examples, we observe 78% improvement in average response time for a random write-dominant I/O trace from an OLTP application running at a large financial institution and 56% improvement for the read-dominant TPC-H workload.

We also show that DFTL can even outperform the ideal page-based FTL, reducing the system response time by utilizing the remaining memory space that could originally be used for storing mapping entries in the ideal page-based FTL for write-back cache. We also

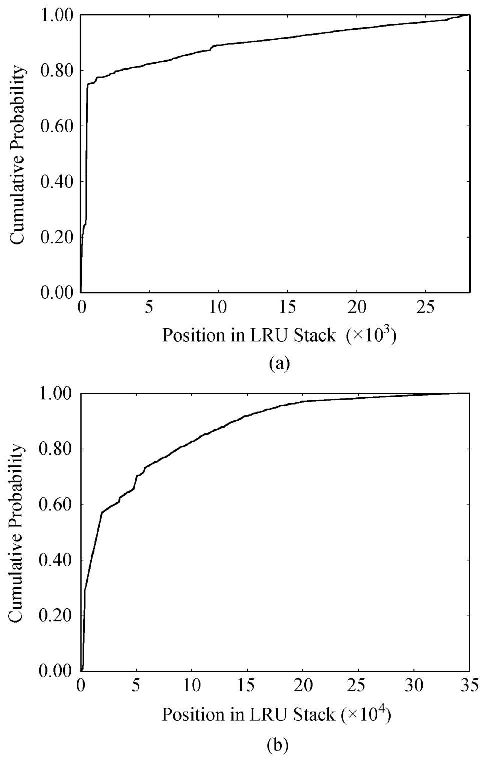


Fig.2. Cumulative distribution function (CDF) of virtual address access frequency obtained from (a) I/O trace from a financial institution and (b) TPC-C benchmark shows existence of significant temporal locality in I/O workloads. For the Financial trace, about 80% of the accesses belong to the first 5 000 requests in the LRU stack. The characteristics of workloads are described in Table 4.

present that SSDs that implement DFTL show predictable I/O response time as DFTL can do away full merge operations, minimizing GC overheads.

A preliminary version of the work was published in [14] and in the paper we enhance FlashSim and conduct more corresponding experimental comparison and of DFTL and other FTLs. Besides, the paper presents more details of our DFTL implementation.

## 2 Background and Motivation

The mapping tables and other data structures, manipulated by the FTL are stored in a small, fast SRAM. The FTL algorithms are executed on it. FTL helps in emulating flash as a normal block device by performing out-of-place updates which in turn helps to hide the erase operations in flash. It can be implemented at different address translation granularities. At two extremes are page-level and block-level translation schemes which we will discuss next. As has been stated, we begin by understanding two extremes of FTL designs with regard to what they store in their in-SRAM mapping table. Although neither is used in practice, these will help us understand the implications of various FTL design choices on performance.

As shown in Fig.3(a), in a page-level FTL scheme, the logical page number of the request sent to the device from the upper layers such as file system can be mapped into any page within the flash. This should remind the reader of a fully associative cache[15]. Thus, it provides compact and efficient utilization of blocks within the flash device. However, on the downside, such translation requires a large mapping table to be stored in SRAM. For example, a 16 GB flash memory requires approximately 32 MB of SRAM space for storing a page-level mapping table. Given the order of magnitude difference in the price/byte of SRAM and flash, having large SRAMs which scale with increasing flash size is infeasible.

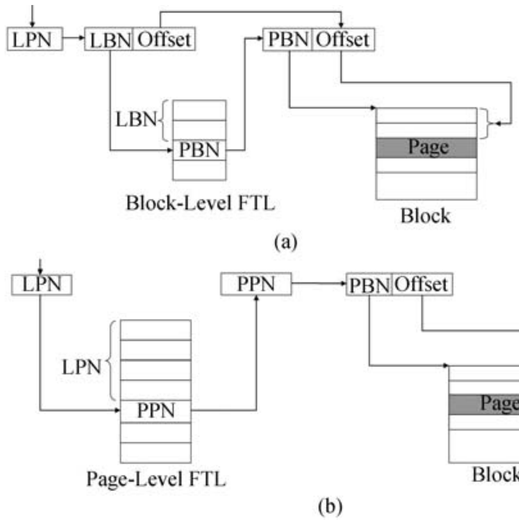


Fig.3. (a) Page-level FTL scheme. (b) Block-level FTL scheme. LPN: logical page number, PPN: physical page number, LBN: logical block number, PBN: physical block number.

At the other extreme, in a block-level FTL scheme, as depicted in Fig.3(b), page offset within a block is fixed. The logical block number is translated into a physical block number using the mapping table similar to set-associative cache design[15]. The logical page number offset within the block is fixed. Fig.3(b) shows an example of block-based address translation. The logical page number (LPN) is converted into a logical block number (LBN) and offset. The LBN is then converted to physical block number (PBN) using the block-based mapping table. Thus, the offset within the block is invariant to address translation. The size of the mapping table is reduced by a factor of block size/page size (128 KB/2 KB=64) as compared with page-level FTL. However, it provides less flexibility as compared with the page-based scheme. Even if there are free pages within a block except at the required offset, this scheme may require allocation of another free block, thus reducing the efficiency of block utilization. Moreover, the specification for large block-based flash devices requiring sequential programming within the block. This makes this scheme infeasible to implement in such devices.

To address the shortcomings of the above two extreme mapping schemes, researchers have come up with a variety of alternatives. Log-buffer based FTL scheme is a hybrid FTL which combines a block-based FTL with a page-based FTL as shown in Fig.4. The entire flash memory is partitioned into two types of blocks — data and log/update blocks. First write to a logical address is done in data blocks. Although many schemes have been proposed[12,16-19], they share one fundamental design principle. All of these schemes are a hybrid between page-level and block-level schemes. They logically partition their blocks into two groups — data blocks and log/update blocks. Data blocks form the

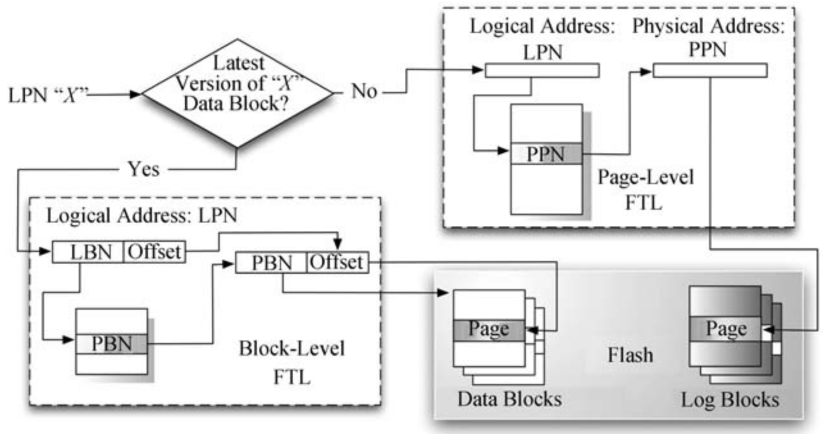


Fig.4. Hybrid FTL scheme, combining a block-based FTL for data blocks with a page-based FTL for log blocks. PPN: physical page number, LBN: logical block number, PBN: physical block number, LPN: logical page number.

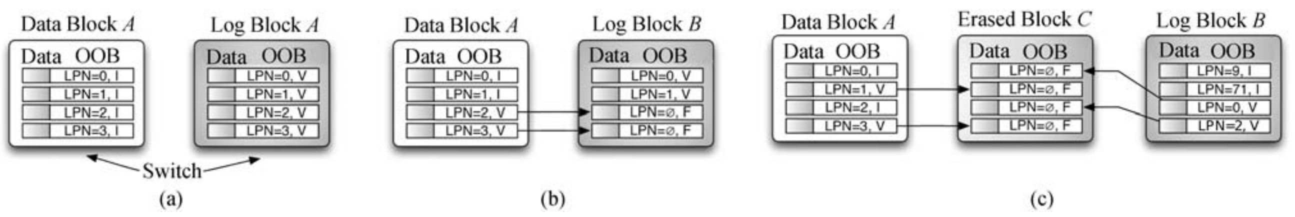


Fig.5. Various merge operations in log-buffer based FTL schemes. LPN: logical page number, V: valid, I: invalid, and F: free/erased. (a) Switch merge. (b) Partial merge. (c) Full merge.

majority and are mapped using the block-level mapping scheme. A second special type of blocks are called log blocks whose pages are mapped using a page-level mapping style. Fig.4 illustrates such hybrid FTLs. Any update on the data blocks are performed by writes to the log blocks. The log-buffer region is generally kept small in size (for example, 3% of total flash size[19]) to accommodate the page-based mappings in the small SRAM. Extensive research has been done in optimizing log-buffer based FTL schemes[12,16-19].

The hybrid FTLs invoke a garbage collector whenever no free log blocks are available. Garbage collection requires merging log blocks with data blocks. The merge operations can be classified into: switch merge, partial merge, and full merge. In Fig.5(a), since log block $B$ contains all valid, sequentially written pages corresponding to data block $A$, a simple switch merge is performed, whereby log block $B$ becomes new data block and the old data block $A$ is erased. Fig.5(b) illustrates partial merge between blocks $A$ and $B$ where only the valid pages in data block $A$ are copied to log block $B$ and the original data block $A$ is erased changing the block $B$'s status to a data block. Full merge involves the largest overhead among the three types of merges. As shown in Fig.5(c), log block $B$ is selected as the victim block by the garbage collector. The valid pages from the log block $B$ and its corresponding data block $A$ are then copied into a new erased block $C$ and blocks $A$ and $B$ are erased.

Full merge can become a long recursive operation in case of a fully-associative log block scheme where the victim log block has pages corresponding to multiple data blocks and each of these data blocks has updated pages in multiple log blocks. This situation is illustrated in Fig.6.

Log block $L1$ containing randomly written data is selected as a victim block for garbage collection. It contains valid pages belonging to data blocks $D1$, $D2$ and $D3$. An erased block is selected from the free block pool and the valid pages belonging to $D1$ are copied to it from different log blocks and $D1$ itself in the order shown. The other pages for $D1$ are copied similarly from log block $L2$ and $L3$. The valid page in $D1$ itself is

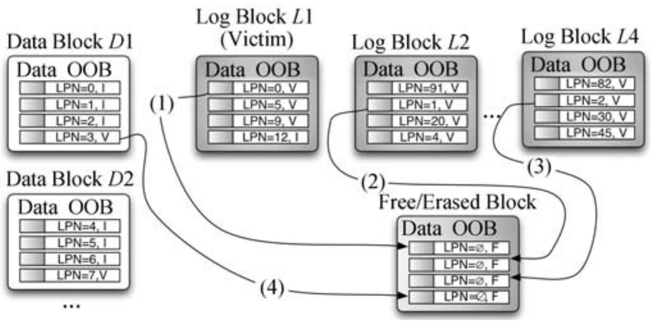


Fig.6. Expensive full merge.

then copied into the new data block. The data block $D1$ is then erased. Similar operations are carried out for data blocks $D2$ and $D3$ since $L1$ contains the latest version of some of the pages for these blocks. Finally, log block $L1$ is erased. This clearly illustrates the large overhead induced by full merge operations. Thus, random writes in hybrid FTLs induce costly garbage collection which in turn affects performance of subsequent operations irrespective of whether they are sequential or random. Recent log buffer-based FTL schemes[18-19] have tried to reduce the number of these full merge operations by segregating log blocks based on access patterns. Hot blocks with frequently accessed data generally contain a large number of invalid pages whereas cold blocks have least accessed data. Utilizing hot blocks for garbage collection reduces the valid page copying overhead, thus lowering the full merge cost.

## 3 Design of DFTL: Our Demand-Based Page-Mapped FTL

Demand-based page-mapped FTL (DFTL) is an enhanced form of the page-level FTL scheme described in Section 2. It does away completely with the notion of log blocks. In fact, all blocks in this scheme, can be used for servicing update requests. Page-level mappings allow requests to be serviced from any physical page on flash. However, as we remarked earlier, the small size of on-flash SRAM does not allow all these page-level mappings to be present in SRAM. However, to make the fine-grained mapping scheme feasible with the constrained SRAM size, a special address translation mechanism has to be developed. In the next subsections, we describe the architecture and functioning of DFTL and highlight its advantages over existing hybrid FTL schemes.

## 3.1 Architectural Design

DFTL makes use of the presence of temporal locality in workloads to judiciously utilize the small on-flash SRAM. Instead of the traditional approach of storing all the address translation entries in SRAM, it dynamically loads and unloads the page-level mappings depending on the workload access patterns. Furthermore, it maintains the complete image of the page-based mapping table on the flash device itself. There are two options for storing the image: the OOB area or the data area of the physical pages. We choose to store the mappings in the data area instead of OOB area because it enables us to group a larger number of mappings into a single page as compared with storing in the OOB area. For example, if four bytes are needed to represent the physical page address in flash, then we can group 512 logically consecutive mappings in the data area of a single page whereas only 16 such mappings would fit an OOB area. Workloads exhibiting spatial locality can benefit since this storage allows pre-fetching of a large number of mappings into SRAM by reading a single page. This amortizes the cost of this additional page-read as subsequent requests are hit within the SRAM itself. Moreover, the additional space overhead incurred is negligible compared with the total flash size. A 1 GB flash device requires only about 2 MB (approximately 0.2% of 1 GB) space for storing all the mappings.

*Data Pages and Translation Pages.* In order to store the address translation mappings on flash data area, we segregated data pages and translation pages. Data pages contain the real data which is accessed or updated during read/write operations whereas pages which only store information about logical-to-physical address mappings are called as translation pages. Blocks containing translation pages are referred to as translation-blocks and data blocks store only data pages. It should be noted that we completely do away with log blocks. As is clear from Fig.7, translation blocks are different from log blocks and are only used to store the address mappings. They require only about 0.2% of the entire flash space and do not require any merges with data blocks.

## 3.2 Logical to Physical Address Translation

A request is serviced by reading from or writing to pages in the data blocks while the corresponding mapping updates are performed in translation blocks. In the following subsections, we describe various data structures and mechanisms required for performing address translation and discuss their impact on the overall performance of DFTL.

*Global Mapping Table and Global Translation Directory.* The entire logical-to-physical address translation set is always maintained on some logically fixed portion of flash and is referred to as the global mapping table. However, only a small number of these mappings can be present in SRAM. These active mappings present in SRAM form the cached mapping table (CMT). Since out-of-place updates are performed on flash, translation pages get physically scattered over the entire flash

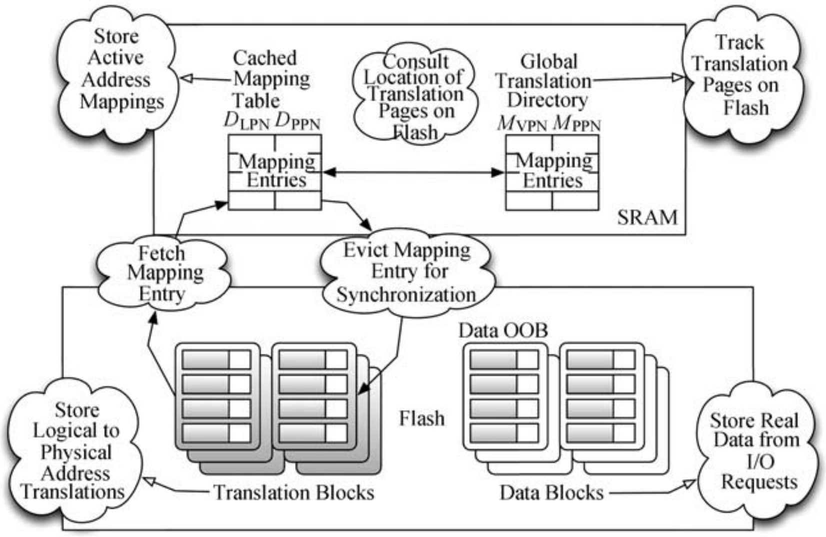

Store Active Address Mappings
Cached Mapping Table
$D_{\rm LPN}$ $D_{\rm PPN}$
Mapping Entries
Consult Location of Translation Pages on Flash
Global Translation Directory
$M_{\rm VPN}$ $M_{\rm PPN}$
Mapping Entries
Track Translation Pages on Flash
SRAM
Fetch Mapping Entry
Evict Mapping Entry for Synchronization
Data OOB
Flash
Store Logical to Physical Address Translations
Translation Blocks
Data Blocks
Store Real Data from I/O Requests

Fig.7. Schematic design of DFTL. $D_{\rm LPN}$: logical data page number, $D_{\rm PPN}$: physical data page number, $M_{\rm VPN}$: virtual translation page number, $M_{\rm PPN}$: physical translation page number.

memory. DFTL keeps track of all these translation pages on flash by using a global translation directory (GTD). Although GTD is permanently maintained in the SRAM, it does not pose any significant space overhead. For example, for a 16GB flash memory device, 16 384 translation pages are needed (each capable of storing 512 mappings), requiring a GTD of about 64 KB.

*DFTL Address Translation Process.* Algorithm 1 describes the process of address translation for servicing a request. If the required mapping information for the given read/write request exists in SRAM (in CMT), it is serviced directly by reading/writing the data page on flash using this mapping information. If the information is not present in SRAM then it needs to be fetched into the CMT from flash. However, depending on the state of CMT and the replacement algorithm being used, it may entail evicting entries from SRAM. We use the segmented LRU array cache algorithm[20] for replacement in our implementation. However, other algorithms such as evicting Least Frequently Used mappings can also be used.

**Algorithm 1.** DFTL Address Translation

```
Input: request's logical page number ($request_{\mathrm{lpn}}$),
       request's size ($request_{\mathrm{size}}$)
Output: NULL
while $request_{\mathrm{size}} \neq 0$ do
    if $request_{\mathrm{lpn}}$ miss in cached mapping table then
        if cached mapping table is full then
            /* Select entry for eviction using segmented
               LRU replacement algorithm */
            $victim_{\mathrm{lpn}} \leftarrow select\_victim\_entry()$
            if $victim_{\mathrm{last\_mod\_time}} \neq victim_{\mathrm{load\_time}}$ then
                /*$victim_{\mathrm{type}}$: translation or data block
                $Translation\_Page_{\mathrm{victim}}$: physical transla-
                tion-page number containing victim
                entry */
                $Translation\_Page_{\mathrm{victim}} \leftarrow consult\_GTD$
                ($victim_{\mathrm{lpn}}$)
                $victim_{\mathrm{type}} \leftarrow$ translation block
                $DFTL\_Service\_Request(victim)$
            end
            $erase\_entry(victim_{\mathrm{lpn}})$
        end
        $Translation\_Page_{\mathrm{request}} \leftarrow consult\_GTD(request_{\mathrm{lpn}})$
        /* Load map entry of the request from flash into
           cached mapping table */
        $load\_entry\ (Translation\_Page_{\mathrm{request}})$
    end
    $request_{\mathrm{type}} \leftarrow$ data block
    $request_{\mathrm{ppn}} \leftarrow CMT\_lookup(request_{\mathrm{lpn}})$
    $DFTL\_Service\_Request(request)$
    $request_{\mathrm{size}} - -$
end
```

If the victim chosen by the replacement algorithm has not been updated since the time it was loaded into SRAM, then the mapping is simply erased without requiring any extra operations. This reduces traffic to translation pages by a significant amount in read-dominant workloads. In our experiments, approximately 97% of the evictions in read-dominant TPC-H benchmark did not incur any eviction overheads. Otherwise, the global translation directory is consulted to locate the victim's corresponding translation page on flash. The page is then read, updated, and re-written to a new physical location. The corresponding GTD entry is updated to reflect the change. Now the incoming request's translation entry is located using the same procedure, read into the CMT and the requested operation is performed. The example in Fig.8 illustrates the process of address translation when a request incurs a CMT miss. Suppose a request to $D_{\mathrm{LPN}}$ 1 280 incurs a miss in cached mapping table (CMT) (1). A victim entry $D_{\mathrm{LPN}}$ 1 is selected, its corresponding translation page $M_{\mathrm{PPN}}$ 21 is located using global translation directory (GTD) (2), $M_{\mathrm{PPN}}$ 21 is read, updated ($D_{\mathrm{PPN}}130 \rightarrow D_{\mathrm{PPN}}$ 260) and written to a free translation page ($M_{\mathrm{PPN}}$ 23) (3)∼(4), GTD is updated ($M_{\mathrm{PPN}}$ 21 $\rightarrow M_{\mathrm{PPN}}$ 23) and $D_{\mathrm{LPN}}$ 1 entry is erased from CMT (5)∼(6). The original request's ($D_{\mathrm{LPN}}$ 1 280) translation page is located on flash ($M_{\mathrm{PPN}}$ 15) (7)∼(11). The mapping entry is loaded into CMT and the request is serviced. Note that each GTD entry maps 512 logically consecutive mappings.

*Overhead in DFTL Address Translation.* The worst-case overhead includes two translation page reads (one for the victim chosen by the replacement algorithm and the other for the original request) and one translation page write (for the victim) when a CMT miss occurs. However, our design choice is rooted deeply in the existence of temporal locality in workloads which helps in reducing the number of evictions. As discussed earlier, pre-fetching of mapping entries for I/O streams exhibiting spatial locality also helps to amortize this overhead. Furthermore, the presence of multiple mappings in a single translation page allows batch updates for the entries in the CMT, physically co-located with the victim entry. We later show through detailed experiments that the extra overhead involved with address translation is much less as compared with the benefits accrued by using a fine-grained FTL.

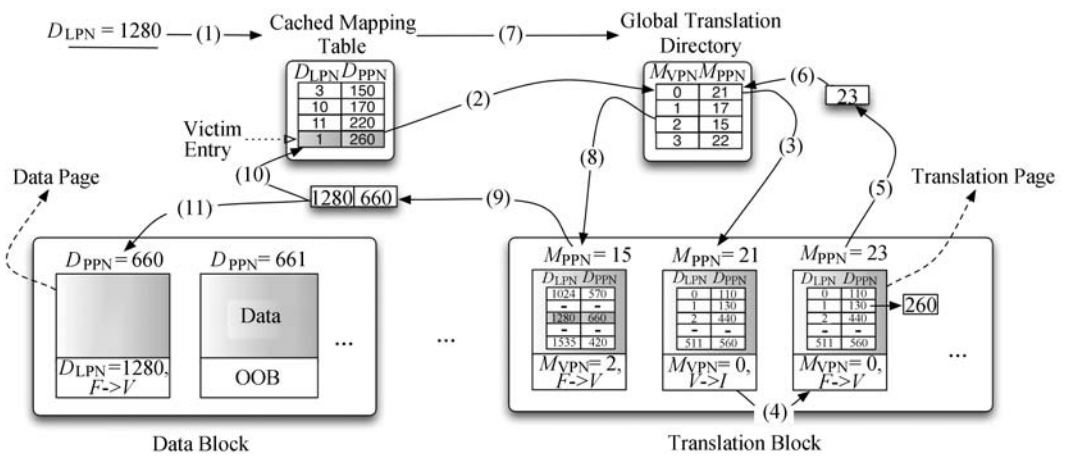


Fig.8. Example of DFTL address translation process.

## 3.3 Read/Write Operation and Garbage Collection

Till now our focus is on performing address translation to locate the page to be read or updated. In this subsection, we explain the actual data read and write operations along with the garbage collection mechanism involved. Read requests are directly serviced through flash page read operations once the address translation is completed. DFTL maintains two blocks, namely current data block and current translation block, where the data pages and translation pages are written, respectively. Page-based mappings allow sequential writes within these blocks, thus conforming to the large-block sequential write specification[11]. DFTL maintains pointers to the next free pages in the data and map blocks being currently written to. For write requests, DFTL allocates the next available free page in the current data block, writes to it and then updates the map entry in the CMT.

However, as writes/updates propagate through the flash, over a period of time the available physical blocks (in erased state) decreases. DFTL maintains a high watermark called $GC_{\text{threshold}}$, which represents the limit till which writes are allowed to be performed without incurring any overhead of garbage collection for recycling the invalidated pages. This threshold can be adjusted with changing workload characteristics to optimize flash device performance. If it is set to a high level, the garbage collector will be invoked more often but the system will be able to maintain a high percentage of erased blocks. On the other hand, a lower setting helps to improve block utilization in the flash device while making the system operate at a resource-constrained level. Thus a delicate balance must be maintained to optimize performance. This is one of the biggest advantages in DFTL as none of the other state-of-the-art hybrid FTL schemes provide this adaptability to changing workload environments. Once $GC_{\text{threshold}}$ is crossed, DFTL invokes the garbage collector. Victim blocks are selected based on a simple cost-benefit analysis that we adapt from [21]. In this analysis, cost represents the overhead involved in copying valid pages from the victim block and benefit is the amount of free space re claimed. These aspects of garbage collection are well studied and not the focus of our research. Any other garbage collection algorithm can be employed. However, we found empirically that minimizing the cost of valid page copying reduces the overall garbage collection overhead which in turn improves device performance during periods of intense I/Os by servicing the requests quicker and reducing the queuing delays in various storage subsystems.

Different steps are followed depending on whether the victim is a translation block or a data block before returning it to the free block pool after erasing it. If it is a translation block, then we copy the valid pages to the current translation block and update the GTD. However, if the victim is a data block, we copy the valid pages to the current data block and update all the translation pages and CMT entries associated with these pages. In order to reduce the operational overhead, we utilize a combination of lazy copying and batch updates. Instead of updating the translation pages on flash, we only update the CMT for those data pages whose mappings are present in it. This technique of lazy copying helps in delaying the proliferation of updates to flash till the corresponding mappings are evicted from SRAM. Moreover, multiple valid data pages in the victim may have their virtual-to-physical address translations present in the same translation-page. By combining all these modifications into a single batch update, we reduce a number of redundant updates. The associated global translation directory entries are also updated to reflect the changes. The examples in Figs. 9 and 10 display the working of our garbage collector when the $GC_{\text{threshold}}$ is reached. Fig.9 is when a translation block is selected as victim and Fig.10 is when a data block is selected as victim. Algorithm 2 shows the detailed description of read/write

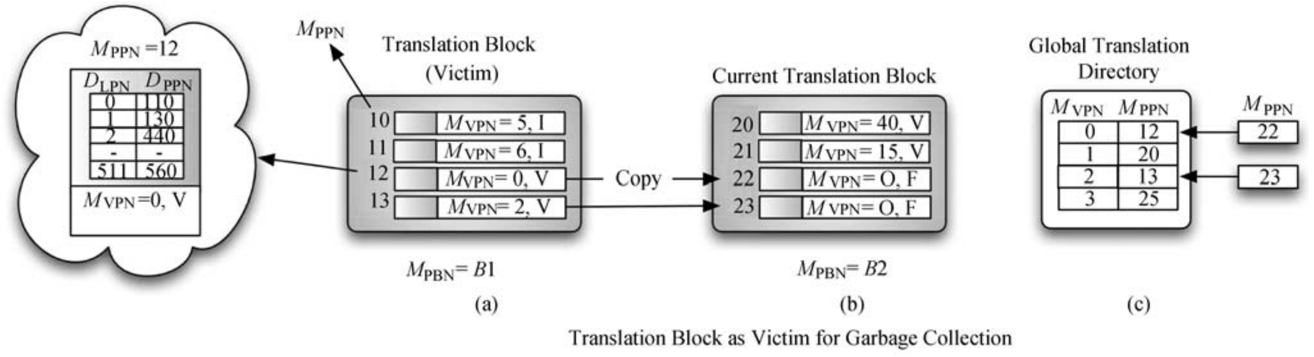


Fig.9. Example of translation block as victim for garbage collection. (a) Select victim block. Translation block ($M_{\text{PBN}}$ $B1$) is selected as victim for garbage collection. (b) Copy valid map pages. Valid pages $M_{\text{PPN}}$ 12 & $M_{\text{PPN}}$ 13 are copied to the current translation block ($M_{\text{PBN}}$ $B2$) at free pages $M_{\text{PPN}}$ 22 & $M_{\text{PPN}}$ 23. (c) Update global translation directory. Global translation directory entries corresponding to $M_{\text{VPN}}$ 0 & $M_{\text{VPN}}$ 2 are updated ($M_{\text{PPN}}$ 12$\rightarrow$ $M_{\text{PPN}}$ 22, $M_{\text{PPN}}$ 13 $\rightarrow$ $M_{\text{PPN}}$ 23).

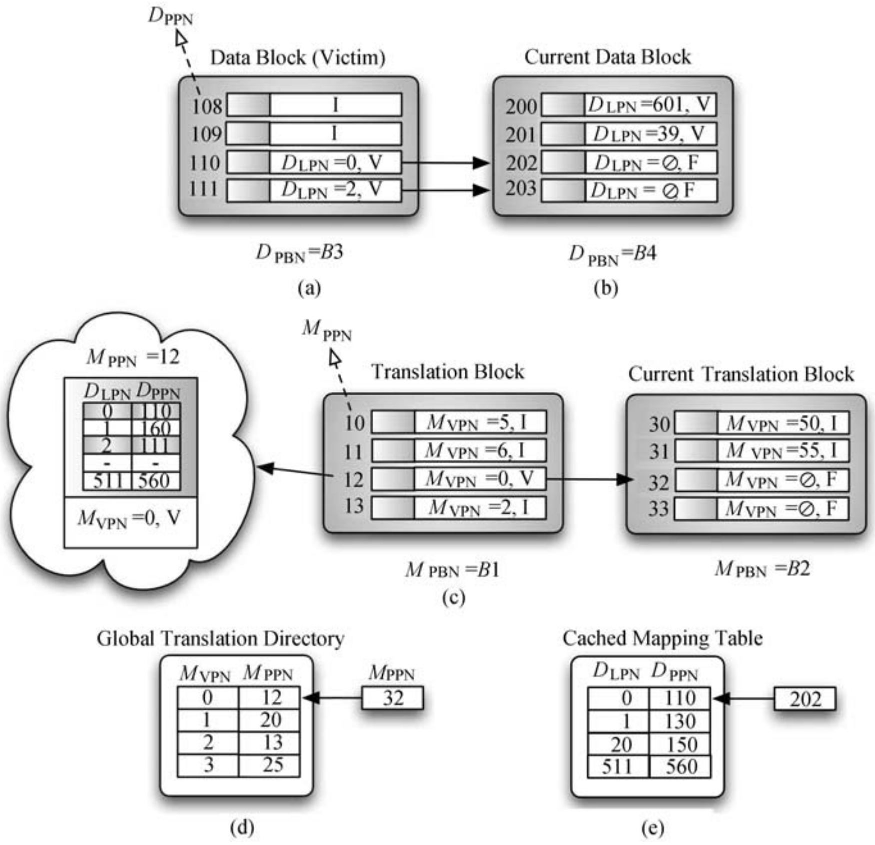


Fig.10. Example of data block as victim for garbage collection. (a) Select victim block. Data block ($D_{\text{PBN}}$ $B3$) is selected as victim for garbage collection. (b) Copy valid data pages. Valid pages $D_{\text{PPN}}$ 110 & $D_{\text{PPN}}$ 111 are copied to the current data block ($D_{\text{PBN}}$ $B4$) at free pages $D_{\text{PPN}}$ 202 & $D_{\text{PPN}}$ 203. (c) Update corresponding translation page. Translation page $M_{\text{PPN}}$ 12 containing the mappings for the valid pages $D_{\text{PPN}}$ 110 & $D_{\text{PPN}}$ 111 is updated and copied to the current map block ($M_{\text{PBN}}$ $B2$). (d) Update global translation directory. Global translation directory entry corresponding to $M_{\text{VPN}}$ 0 is updated ($M_{\text{PPN}}$ 12 $\rightarrow$ $M_{\text{PPN}}$ 32). (e) Update cached mapping table. Since $D_{\text{LPN}}$ 0 is present in cached mapping table, the entry is also updated ($D_{\text{PPN}}$ 110 $\rightarrow$ $D_{\text{PPN}}$ 202). Note: We do not illustrate the advantages of batch updates and lazy copying in this example.

**Algorithm 2.** Garbage Collection

```
Input: NULL
Output: NULL
$victim \leftarrow select\_victim\_entry()$;
/* Victim is translation block */
if $victim_{\rm type} \in TRANSLATION\_BLOCK\_SET$ then
    /* 1) Copy only valid pages in the victim block to
       the current translation block, 2) invalidate old
       pages, and update GDT */
    foreach $victim_{{\rm page}(i)}$ do
        if $victim_{{\rm page}(i)}$ is valid then
            $curr\_translation\_blk \leftarrow get\_curr\_translation\_blk()$;
            $copy\_page(victim_{{\rm page}(i)},\ curr\_map\_blk)$;
            $update\_GTD(victim_{{\rm page}(i)})$;
        end
    end
end
else
    /* Victim is data block */ foreach $victim_{{\rm page}(i)}$ do
        /* Copy only valid pages in the victim block to
           the current data block, invalidate old pages,
           and mark their corresponding translation pages
           for update */
        if $victim_{{\rm page}(i)}$ is valid then
            $curr\_data\_blk \leftarrow get\_curr\_data\_blk()$;
            $copy\_page(victim_{{\rm page}(i)},\ curr\_data\_blk)$;
            $translation\_page\_update\_set[] \leftarrow mark\_corr\_translation\_page\_for\_update\ (victim_{{\rm page}(i)})$;
        end
    end
    /* Perform batch update on the marked translation
       pages */
    foreach $translation\_page_i \in translation\_page\_update\_set$ do
        $curr\_translation\_blk \leftarrow get\_curr\_translation\_blk()$;
        $old\_translation\_page \leftarrow translation\_page_i$;
        $update\_translation\_page(translation\_page_i,\ curr\_translation\_blk)$;
        $invalidate(old\_translation\_page)$;
        $update\_GTD(translation\_page_i)$;
        if $translation\_page_i \in Cached\ Mapping\ Table$ then
            $update\_CMT(translation\_page_i)$;
        end
    end
end
$erase\_blk\ (victim)$; /* Erase the victim block */
```

operation including garbage collection mechanism in our implementation consideration.

Although flash is a non-volatile storage device, it relies on volatile on-flash SRAM which is susceptible to power failure in the host. Power-failure followed by data loss on cached mapping entries, are general problems for all FTL schemes where mapping entries are storage on the volatile memory. When power failure occurs, all logical-physical mapping information stored in the cached mapping table on SRAM will be lost. Note that we do not discuss how to improve the risk of data loss for power-failure, instead, we discuss how effectively and fast, DFTL can recover the loss of cached mappings in an even of power-failure with a small amount of battery-backup cache support. The traditional approach of reconstructing the mapping table utilizes scanning the logical addresses stored in the OOB area of all physical pages on flash[12]. However, the scanning process incurs high overhead and leads to long latencies while the mapping table is being recovered. In DFTL, the global translation directory stores the locational information corresponding to the global mapping table. Thus, storing the GTD on non-volatile storage (e.g., battery backed memory) resilient to power failure such as a fixed physical address location on flash device itself helps to bootstrap recovery. This can be performed periodically or depending on the required consistency model. Moreover, since GTD size is very small (4 KB for 1 GB flash), the overhead involved in terms of both space and extra operations is also very small. However, at the time of power failure there may be some mappings present in the cached mapping table, that have been updated but not written back to map pages on flash yet. If strong consistency is required then even the cached mapping table needs to be saved along with the GTD.

## 3.4 Comparison of Existing FTLs with DFTL

Table 1 shows some of the salient features of different FTL schemes. The DFTL architecture provides some intrinsic advantages over existing hybrid FTLs which are as follows.

Existing hybrid FTL schemes try to reduce the number of full merge operations to improve their performance. DFTL, on the other hand, completely does away with full merges. This is made possible by page-level mappings which enable relocation of any logical page to any physical page on flash while other hybrid FTLs have to merge page-mapped log blocks with block-mapped data blocks.

DFTL utilizes page-level temporal locality to store pages which are accessed together within same physical blocks. This implicitly separates hot and cold blocks as compared to LAST and Superblock schemes[18-19] require special external mechanisms to achieve the segre-

**Table 1**. FTL Schemes Classification

| | Replacement Block FTL | BAST | FAST | SuperBlock | LAST | DFTL | Ideal Page FTL |
|---|---|---|---|---|---|---|---|
| FTL type | Block | Hybrid | Hybrid | Hybrid | Hybrid | Page | Page |
| Mapping granularity | Block | DB-block LB-page | DB-block LB-page | SB-block LB/blocks within SB-page | DB/sequential LB-block Random LB-page | Page | Page |
| Division of update blocks ($M$) | – | – | 1 sequential + $(M-1)$ random | – | (m) sequential- $(M-m)$ Random (hot and cold) | – | – |
| Associativity of blocks (data:update) | $(1:K)$ | $(1:M)$ | Random LB-$(N:M-1)$ sequential LB-(1:1) | $(S:M)$ | Random LB-$(N:M-m)$ Sequential LB- (1:1) | $(N:N)$ | $(N:N)$ |
| Blocks available for updates | Replacement blocks | Log blocks | Log blocks | Log blocks | Log blocks | All data blocks | All blocks |
| Full merge operations | Yes | Yes | Yes | Yes | Yes | No | No |

Note: $N$: number of data blocks, $M$: number of log blocks, $S$: number of blocks in a super block, $K$: number of replacement blocks. DB: data block, LB: log block, SB: super block. In FAST and LAST FTLs, random log blocks can be associated with multiple data blocks.

gation. Thus, DFTL adapts more efficiently to changing workload environment as compared with existing hybrid FTL schemes.

Poor random write performance is argued to be a bottleneck for flash-based devices. As is clearly evident, it is not necessarily the random writes which cause poor flash device performance but the intrinsic shortcomings in the design of hybrid FTLs which cause costly merges (full and partial) on log blocks during garbage collection. Since DFTL does not require these expensive full-merges, it is able to improve random write performance of flash devices.

All hybrid log-buffer based schemes maintain a very small fraction of log blocks (3% of total blocks[19]) to keep the page-level mapping footprint small (in SRAM). This forces them to perform garbage collection as soon as these log blocks are utilized. Some schemes[17-18] may even call garbage collector even though there are free pages within these log blocks (because of low associativity with data blocks). DFTL, on the other hand, can delay garbage collection till $GC_{\text{threshold}}$ is reached which can be dynamically adjusted to suit various input streams.

In hybrid FTLs, only log blocks are available for servicing update requests. This can lead to low block utilization for workloads whose working-set size is smaller than the flash size. Many data blocks will remain unutilized (hybrid FTLs have block-based mappings for data blocks) and unnecessary garbage collection will be performed. DFTL solves this problem since updates can be performed on any of the data blocks.

## 4 Experimental Setup

In order to study the performance implications of various FTL schemes, we develop a simulation framework for flash-based storage systems called FlashSim[13]. FlashSim is built by enhancing Disksim[22], a well-regarded disk drive simulator. It was designed with a modular architecture with the capability to model a holistic flash-based storage environment. It is able to simulate different storage sub-system components including device drivers, controllers, caches, flash devices, and various interconnects. In our integrated simulator, we add the basic infrastructure required for implementing the internal operations (page read, page write, block erase, etc.) of a flash-based device. The core FTL engine is implemented to provide virtual-to-physical address translations along with a garbage collection mechanism. Furthermore, we implement a multitude of FTL schemes: 1) a block-based FTL scheme (replacement-block FTL[23]), 2) a hybrid FTL (FAST[12]), 3) LAST (LAST[19]), 4) our page-based DFTL scheme, and 5) an idealized page-based FTL. This setup is used to study the impact of various FTLs on flash device performance and more importantly on the components in the upper storage hierarchy. FlashSim has been validated against commercial SSDs in terms of performance behavioral similarity[13].

We simulate a 32 GB NAND flash memory device. The SSD simulator uses simulation parameters and device specification in Tables 2 and 3. To conduct a fair

**Table 2.** Simulation Parameters

| Default SSD Simulation Parameters | |
|---|---|
| Flash type | Large block |
| Page (data/OOB) | 2 KB/64 B |
| Block | (128 KB + 4 KB) |
| **Latency & Energy Consumption** | |
| Page read | (130.9 $\mu$s, 4.72 $\mu$J) |
| Page write | (405.9$\mu$s, 38.04$\mu$J) |
| Block erase | (1.5 ms, 527.68 $\mu$J) |

**Table 3.** SSD Device Specifications

| Firmware | |
|---|---|
| Garbage collection | Yes |
| Wear-leveling | Implicit/explicit |
| FTL | Page/DFTL/FAST/LAST |

comparison of different FTL schemes, we consider only a portion of flash as the active region which stores our test workloads. The remaining flash is assumed to contain cold data or free blocks which are not under consideration during the evaluation. We assume the SRAM to be just sufficient to hold the address translations for FAST FTL. Since the actual SRAM size is not disclosed by device manufacturers, our estimate represents the minimum SRAM required for the functioning of a typical hybrid FTL. We allocate extra space (approximately 3% of the total active region[18]) for use as log-buffers by the hybrid FTL.

We use a mixture of real-world and synthetic traces to study the impact of different FTLs on a wide spectrum of enterprise-scale workloads. Table 4 presents salient features of our workloads. We employ a write-dominant I/O trace from an OLTP application running at a financial institution② made available by the Storage Performance Council (SPC), henceforth referred to to as the Financial trace. We also experiment using Cello99③, which is a disk access trace collected from a time-sharing server exhibiting significant writes; this server was running the HP-UX operating system at Hewlett-Packard Laboratories. We consider two read-dominant workloads to help us assess the performance degradation, if any, suffered by DFTL in comparison with other hybrid FTL schemes due to its address translation overhead. For this purpose, we use TPC-H[24], which is an ad-hoc, decision-support benchmark (OLAP workload) examining large volumes of data to execute complex database queries. Also, we use a read-dominant Web search engine trace④ made available by SPC. Finally, apart from these real traces we also use a number of synthetic traces to study the behavior of different FTL schemes for a wider range of workload characteristics than those exhibited by the above real-world traces.

**Table 4.** Enterprise-Scale Workload Characteristics

| Workloads | Avg. Req. Size (KB) | Read (%) | Sequentiality (%) | Inter-Arrival Time (ms) |
|---|---|---|---|---|
| Financial (OLTP) | 4.38 | 9.0 | 2.0 | 133.50 |
| Cello99 | 5.03 | 35.0 | 1.0 | 41.01 |
| TPC-H (OLAP) | 12.82 | 95.0 | 18.0 | 155.56 |
| Web search | 14.86 | 99.0 | 14.0 | 9.97 |

The device service time is a good metric for estimating FTL performance since it captures the overheads due to both garbage collection and address translation. However, it does not include the queuing delays for requests pending in I/O driver queues. In this study, we utilize both 1) indicators of the garbage collector's efficacy and 2) response time as seen at the I/O driver (this is the sum of the device service time and time spent waiting in the driver's queue, we will call it the system response time) to characterize the behavior/performance of the FTLs. The garbage collection overhead is demonstrated through the impact of merges, the copying of valid pages, and the erasing of the blocks in these operations. In subsequent subsections, we highlight the cost of full merges, examine the performance of different FTL schemes, and evaluate their ability to handle overload conditions in different workloads.

## 5 Results

### 5.1 Analysis of Garbage Collection and Address Translation Overheads

As explained in Section 2, the garbage collector may have to perform merge operations of various kinds (switch, partial, and full) while servicing update requests. Recall that merge operations pose overheads in the form of block erases. Additionally, merge operations might induce copying of valid pages from victim blocks — a second kind of overhead. We report both these overheads and the different kinds of merge operations in Fig.11 for our workloads. As expected from Section 3 and corroborated by the experiments shown in Fig.11, read-dominant workloads (TPC-H and Web Search) — with their small percentage of write requests — exhibit much smaller garbage collection overheads than Cello99 or Financial trace. The number of merge operations and block erases are so small for the highly

②OLTP trace from UMass Trace Repository, http://traces.cs.umass.edu/index.php/Storage/Storage, May 2013.

③HP-Labs. Tools and Traces. http://tesla.hpl.hp.com/public_software/, May 2013.

④WebSearch trace from UMass Trace Repository, http://traces.cs.umass.edu/index.php/Storage/Storage, May 2013.

read-dominant Web Search trace that we do not show these in Figs.11(a), 11(b), and 11(c).

Hybrid FTLs can perform switch merges only when the victim update block (selected by garbage collector) contains valid data belonging to logically consecutive pages. DFTL, on the other hand, with its page-based address translation, does not have any such restriction. Hence, DFTL shows a larger number of switch merges for even random-write dominant Financial trace as seen in Fig.11(a).

For TPC-H, although DFTL shows a larger number of total merges, its fine-grained addressing enables it to replace full merges with less expensive partial merges. With FAST as many as 60% of the full merges involve more than 20 data blocks. As we will observe later, this directly impacts FAST's overall performance. Fig.11(b) shows the larger number of block erases with FAST as compared with DFTL for all our workloads. This can be directly attributed to the large number of data blocks that need to be erased to complete the full merge operation in hybrid FTLs. Moreover, in hybrid FTLs only a small fraction of blocks (log blocks) are available as update blocks, whereas DFTL allows all blocks to be used for servicing update requests. This not only improves the block utilization in our scheme as compared with FAST but also contributes in reducing the invocation of the garbage collector.

DFTL introduces some extra overheads due to its address translation mechanism (due to missed mappings that need to be brought into the SRAM from flash). Fig.11(c) shows the normalized overhead (with respect to FAST FTL) from these extra read and write operations along with the extra valid pages required to be copied during garbage collection. Even though the address translation accounts for approximately 90% of the extra overhead in DFTL for most workloads, overall it still performs less extra operations than FAST. For example, DFTL yields a 3-fold reduction in extra read/write operations over FAST for the Financial trace. Our evaluation supports the key insight behind DFTL, namely that the temporal locality present in workloads helps keep this address translation overhead small, i.e., most requests are serviced from the mappings in SRAM. DFTL is able to utilize page-level temporal locality in workloads to reduce the valid page copying overhead since most hot blocks (data blocks and translation blocks) contain invalid pages and are selected as victims by our garbage collector. In our experiments, we observe about 63% hits for address translations in SRAM for the financial trace even with our

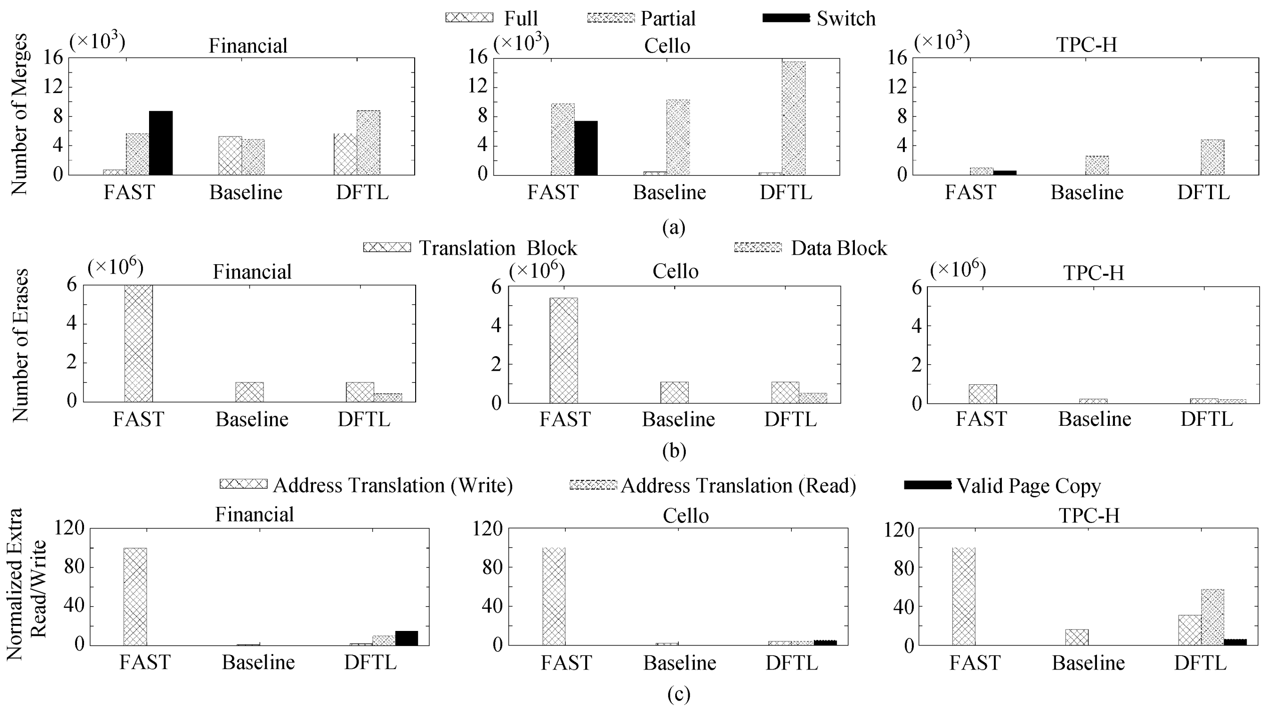


Fig.11. Overheads with different FTL schemes. (a) Merge operations. (b) Block erases. (c) Extra read/write operations. We compare DFTL with FAST and Baseline for three workloads: Financial, Cello99, and TPC-H. The overheads for the highly read-oriented Web Search workload are significantly smaller than the others and we do not show them here. In (c), address translation (read) and address translation (write) denote the extra read and write operations for address translations required in DFTL, respectively. All extra read/write operations have been normalized with respect to FAST FTL scheme.

conservatively chosen SRAM size. Furthermore, the relatively high address translation overhead can be attributed to the minimal size of SRAM that we have used in our experiments. All the values used in Fig.11 are in Table 5.

## 5.2 Performance Analysis

Having seen the comparison of the overheads of garbage collection and address translation for different FTLs, we are now in a position to appreciate their impact on the performance offered by the flash device. The performance of any FTL scheme deteriorates with increase in garbage collection overhead. The Baseline scheme does not incur any address translation overhead and is also able to prevent full-merges because of fine-grained mapping scheme. Thus, it shows the best performance amongst all other implementable FTL schemes. The cumulative distribution function of the average system response time for different workloads is shown in Fig.12.

DFTL is able to closely match the performance of Baseline scheme for the Financial and Cello99 traces, both random-write dominant workloads. In case of the Financial trace, DFTL reduces the total number of block erases as well as the extra page read/write operations by about three times, thus decreasing the overall merge overhead by about 76%. This results in improved device service time and shorter queuing delay (refer to Table 6) which in turn improve the overall I/O system response time by about 78% as compared to FAST.

For Cello99, the improvement is much more dramatic because of the high I/O intensity which increases the pending requests in the I/O driver queue, resulting in higher latencies. Readers should be careful about the following while interpreting these results: we would like to point out that Cello99 represents only a point within a much larger enterprise-scale workload spectrum for which the gains offered by DFTL are significantly large. More generally, DFTL is found to improve the average response time of workloads with random writes with the degree of improvement varying with the workload's properties.

For read-oriented workloads, DFTL incurs a larger additional address translation overhead and its performance deviates from the Baseline (Figs.12(c) and 12(d)). Since FAST is able to avoid any merge operations in the Web Search trace, it provides performance comparable to Baseline. However, for TPC-H, it exhibits a long tail primarily because of the expensive full merges and the consequent high latencies seen by requests in the I/O driver queue. Hence, even though FAST services about 95% of the requests faster than DFTL, it suffers from long latencies in the remaining requests, resulting in a higher average system response time than DFTL.

For FAST to match the performance of DFTL for random-write dominant workloads, it needs a faster flash device. Fig.13 shows the necessary flash device speed-up required for FAST to achieve the performance comparable with our FTL scheme. A four times faster flash will require more investment to attain similar results. Thus, DFTL even helps in reducing deployment costs for flash-based SSD devices in enterprise-servers.

**Table 5.** Analysis of Garbage Collection Overhead for Various FTLs

| Workloads | FTL | Erase (#) | | Merges | | | Read Overhead | | | | Write Overhead | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Type | Data | Map | *SM* | *P* | *F* | (1) | (2) | (3) | (4) | (5) | (6) | (7) | (8) |
| Financial | Baseline | 10 111 | - | 5 275 | 4 836 | - | 15 573 | - | - | - | 15 573 | - | - | - |
| | DFTL | 10 176 | 4 240 | 5 650 | 8 766 | - | 19 369 | 5 945 | 517 456 | 7 582 | 19 369 | 5 945 | 258 017 | 7 582 |
| | FAST | 151 180 | - | 374 | 5 967 | 8 865 | 1 508 490 | - | - | - | 1 508 490 | - | - | - |
| Cello | Baseline | 10 787 | - | 447 | 10 340 | - | 81 109 | - | - | - | 81 109 | - | - | - |
| | DFTL | 10 795 | 5 071 | 353 | 15 513 | - | 82 956 | 42 724 | 730 107 | 29 010 | 82 956 | 42 724 | 251 518 | 29 010 |
| | FAST | 134 676 | - | 1 | 9 763 | 7 694 | 3 149 194 | - | - | - | 3 149 194 | - | - | - |
| TPC-H | Baseline | 2 544 | - | 14 | 2 530 | - | 102 130 | - | - | - | 102 130 | - | - | - |
| | DFTL | 2 678 | 2 118 | 5 | 4 791 | - | 110 716 | 75 255 | 1 449 183 | 10 018 | 110 716 | 75 255 | 50 242 | 10 023 |
| | FAST | 19 476 | - | 5 | 949 | 568 | 618 459 | - | - | - | 618 459 | - | - | - |
| Web Search | Baseline | - | - | - | - | - | - | - | - | - | - | - | - | - |
| | DFTL | 15 | 350 | - | 365 | - | 480 | 6 391 | 1 588 120 | 51 | 480 | 6 391 | 16 390 | 51 |
| | FAST | - | - | - | - | - | - | - | - | - | - | - | - | - |

Note: (1): number of data page reads in GC, (2): number of map page reads in GC, (3): number of map page reads for address translation, and (4): number of map page reads when victim block is a data block. (5): number of data page writes in GC, (6): number of map page writes in GC, (7): number of map page writes for address translation, and (8): number of map page writes when victim block is a data block. "Baseline" in FTL type denotes a baseline FTL scheme. *SM*: number of switch merge operations, *P*: number of partial mergy operations, *F*: number of full merge operations.

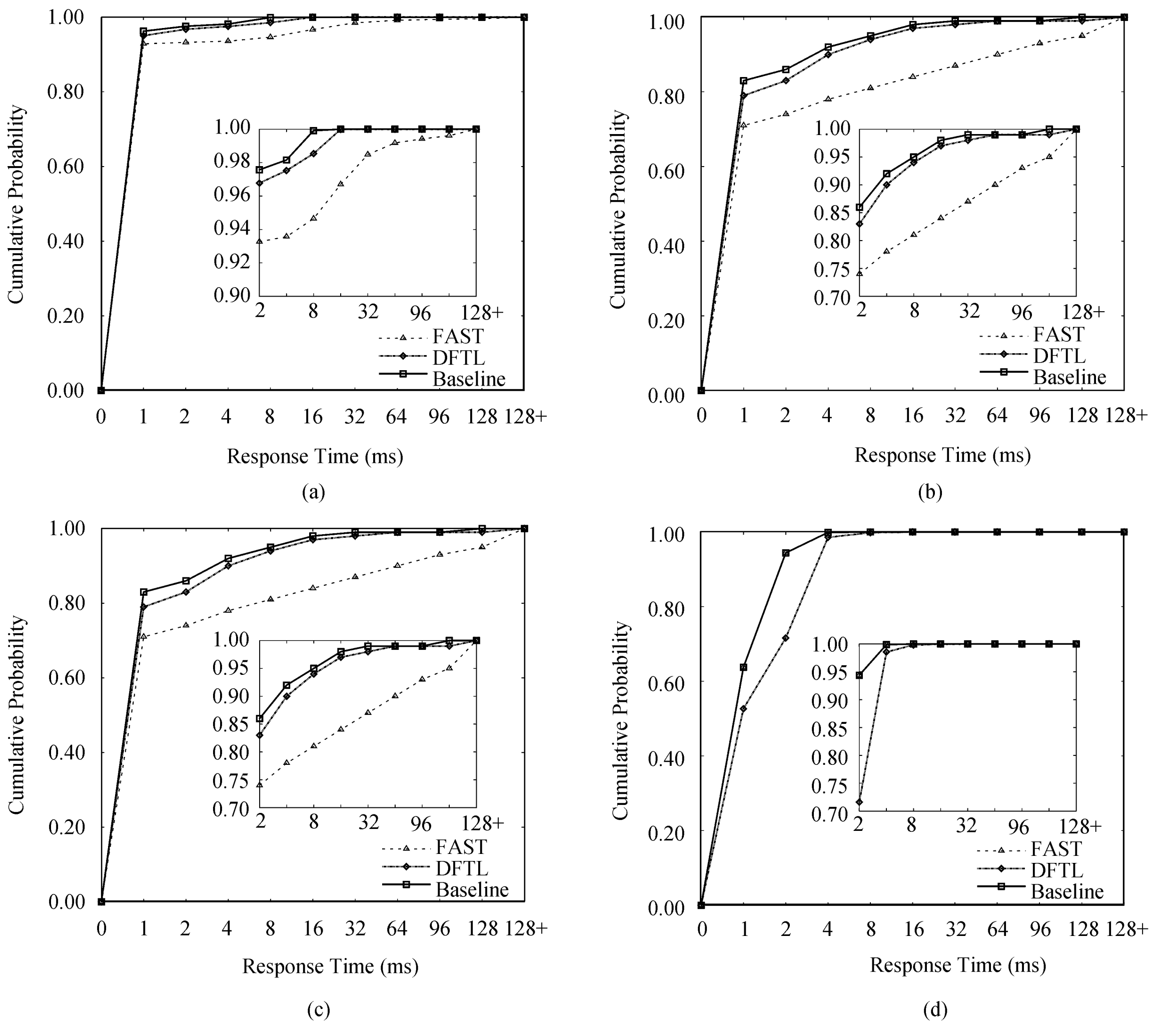


Fig.12. Each graph shows the cumulative distribution function (CDF) of the average system response time for different FTL schemes. (a) Financial trace (OLTP). (b) Cello99. (c) TPC-H. (d) Web Search.

**Table 6.** Performance Results for Different FTL Schemes

| Workloads | FTL Type | System Response Time (ms) | | Dev. Response Time (ms) | | I/O Queuing Delay (ms) | |
|---|---|---|---|---|---|---|---|
| | | Avg. | Std. Dev | Avg. | Std. Dev | Avg. | Std. Dev |
| Financial | Baseline | 0.43 | 0.81 | 0.39 | 0.79 | 0.04 | 0.19 |
| | FAST | 2.75 | 19.77 | 1.67 | 13.51 | 1.09 | 13.55 |
| | DFTL | 0.61 | 1.52 | 0.55 | 1.50 | 0.06 | 0.29 |
| Cello99 | Baseline | 1.50 | 4.96 | 0.41 | 0.80 | 1.08 | 4.88 |
| | FAST | 16.93 | 52.14 | 2.00 | 14.59 | 14.94 | 50.20 |
| | DFTL | 2.14 | 6.96 | 0.59 | 1.04 | 1.54 | 6.88 |
| TPC-H | Baseline | 0.79 | 2.96 | 0.68 | 1.78 | 0.11 | 2.13 |
| | FAST | 3.19 | 29.56 | 1.06 | 11.65 | 2.13 | 26.74 |
| | DFTL | 1.39 | 7.65 | 0.95 | 2.88 | 0.44 | 6.57 |
| Web Search | Baseline | 0.86 | 0.64 | 0.68 | 0.44 | 0.18 | 0.46 |
| | FAST | 0.86 | 0.64 | 0.68 | 0.44 | 0.18 | 0.46 |
| | DFTL | 1.24 | 1.06 | 0.94 | 0.68 | 0.30 | 0.78 |

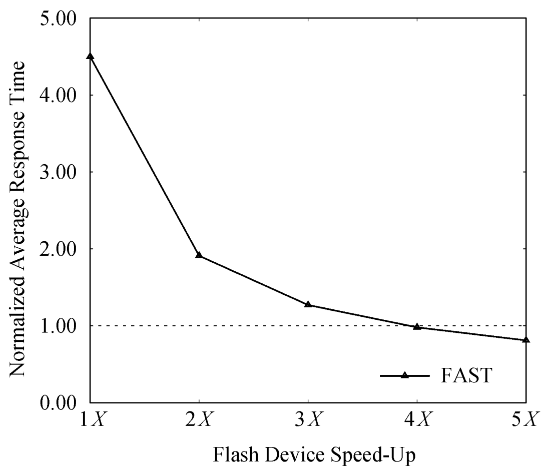


Fig.13. Performance improvement of FAST with flash device speed-up for the Financial. Average response time has been normalized with respect to DFTL performance without any speed-up (1$X$).

In the following subsection, we examine the various overheads associated with different FTL schemes including the cost imposed by garbage collection, especially full-merges in state-of-the-art hybrid FTLs and the address translation overhead in DFTL.

We also performance a microscopic analysis of the impact of garbage collection on instantaneous response time by installing probes within FlashSim to trace individual requests. Detailed results can be found in [14]. Also we study the impact of increased SRAM size on DFTL. We have seen that greater SRAM size improves the hit ratio, reducing the address translation overhead in DFTL, and thus improving flash device performance. However, increasing the SRAM size for holding address translations beyond the workload working-set size does not provide any tangible performance benefit. It would be more beneficial to utilize the extra SRAM for caching popular read requests, or buffering writes than for storing unused address translations. The results about this can be found in [14].

## 5.3 Impact of SSD Cache on SSDs

All the experiments in the preceding subsections were done by ignoring the effect of SSD cache. However, it will be interesting to see the effect of SSD cache on FTL performance. As mentioned earlier, DFTL and FAST FTLs require less SRAM space for mapping entries compared to ideal page-based FTL. We consider 32GB SSD. The ideal page-based FTL needs 16MB SRAM to maintain all mapping entries whereas DFTL and FAST FTLs require only 32KB SRAM for the mapping entries. Thus, the DFTL and FAST FTLs can utilize the remaining huge memory space except for the memory space for mapping entries from entire 16MB SRAM. However, there is no available memory space used for data cache in the ideal page-based FTL.

Fig.14 shows the impact of SSD cache on FTL performance when those remaining memory spaces are used for data cache. As expected, the cache improves flash device performance, reducing the amount of requests sent to the flash device (by about 85% for Financial trace and 58% for TPC-H). The results show the normalized average response time with respect to the baseline. DFTL scheme with cache even further outperforms the baseline, improving their response time by 72% in Financial trace. However, it is still worse than the baseline in TPC-H. It is because our cache is a write-back cache which is highly optimized for write-dominant workloads (note that TPC-H is a read dominant workload) and the DFTL still suffers from extra overhead which is inevitable for the management of mapping entries. Thus, in these workloads, it would be better to increase the SRAM area for mapping tables while allocating less SRAM for data cache when DFTL scheme is used. FAST FTL scheme with cache also improves their response time in both workloads. However they are still worse than the baseline schemes because the FAST FTLs still suffer from expensive full merge operations.

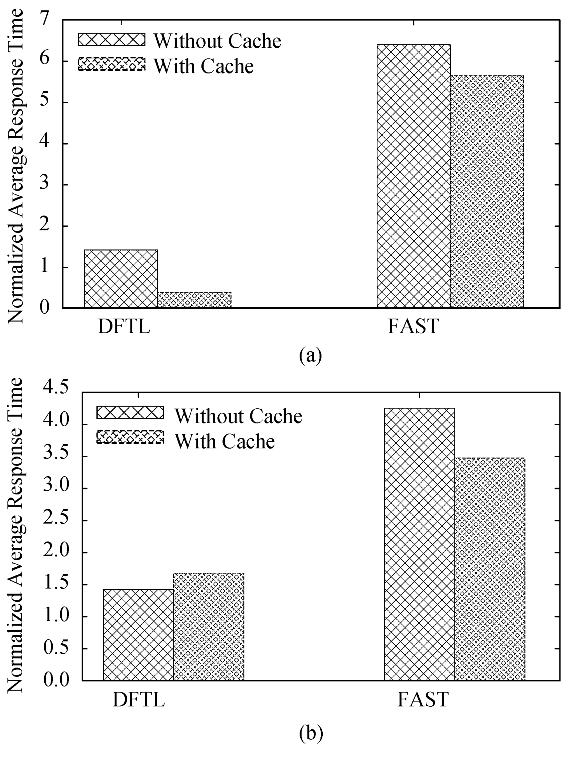
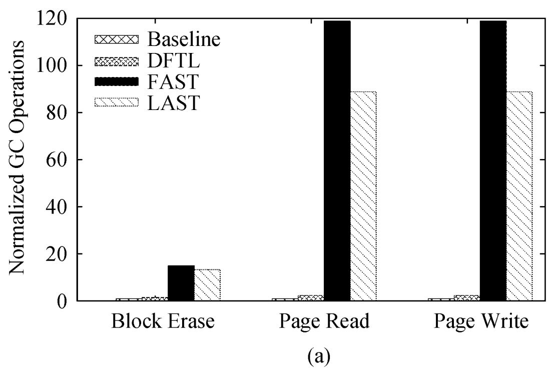


Fig.14. Performance improvement of DFTL and FAST FTLs with device caches. Average response time has been normalized with respect to baseline (ideal page-based FTL) performance without cache. (a) Financial trace. (b) TPC-H benchmark.

## 5.4 Performance Comparison of Various FTL Schemes

We have studied that DFTL could outperform FAST FTL scheme by completely doing away full merge operation in GC phases. In this subsection, we compare how effective DFTL is against the enhanced FTL over FAST FTL scheme, called LAST[19]. In particular, LAST improves the performance shortcomings of the FAST FTL scheme that was caused by the full merge operations, by employing multiple sequential log blocks to exploit spatial locality in workloads, and separating random log blocks into hot and cold regions to alleviate the full much operation cost. We have implemented the LAST FTL scheme by enhancing the FAST FTL scheme and we study the efficiency of DFTL against those hybrid FTL schemes. For performance comparison, we are particularly interested in the GC efficiency in terms of number of block erase operations and additional page read and write operations.

Fig.15 shows the results of our performance comparisons for different FTL schemes in the Financial and TPC-H traces. Fig.15(a) presents normalized block erase operations for FTL schemes, which are normalized with respect to block erase operations of the baseline (ideal page-based FTL scheme). We see that DFTL increases the number of block erase operations by 42% over the baseline whereas FAST and LAST increase them by 1 401% and 1 231% respectively over the baseline. Referring to the explanation that we made for DFTL in its design section, DFTL requires additional page read and writes due to address translation, for CMT misses, and page mapping entries synchronization between CMT and page table on flash, increasing additional address translation page read and write, followed by GC overhead over the baseline. For FAST, it is all attributed to high cost of full merge operation. However, we see that LAST could reduce the erase cost of FAST by 11.3%, which mostly benefits from the hot-cold separation technique for small random write blocks. In addition to block erase operations, we observe significantly increased additional page read and write operations by GC; DFTL increases those additional page read and write operation by GC by 130% however, FAST and LAST make huge increase in those operations against the baseline. However, again LAST could reduce those addition page read and write operations over the FAST by 25%. We have the similar observation from the TPC-H benchmark results in Fig.15(b) as we have from the financial trace results.



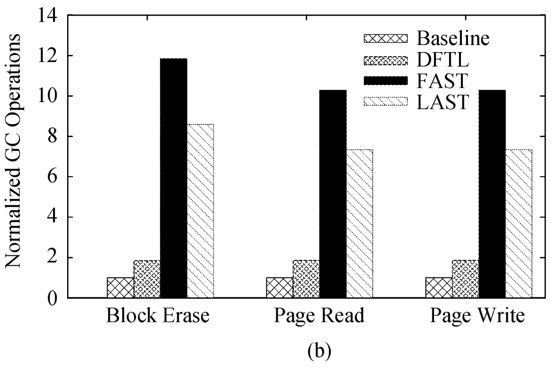


Fig.15. GC efficiency comparison by different FTL schemes. (a) Financial trace. (b) TPC-H benchmark.

## 5.5 Energy Efficiency Analysis of Various FTL Schemes

Even if the power consumption of the flash memory in the SSD may not be significant when compared with other components (CPU and Memory), erasing and writing in the SSD could affect the overall energy efficiency. Table 2 presents that erase operations consume significant power compared with read and write operations, and write operations also consume about an order of magnitude more power than read operations. We enhance our SSD simulator to be able to study energy efficiency with different FTL schemes on SSDs and we compare four different kinds of FTL schemes for their energy efficiency; ideal page-based mapping scheme (baseline), FAST[12], LAST[19] and DFTL.

Fig.16 shows the energy consumption by flash internal operations for different FTL schemes in the Financial and TPC-H traces. Note that the Financial trace is mostly random-write-dominant, while TPC-H is read-dominant (refer to Table 4). Thus, the energy consumption for the Financial trace is much higher than that of TPC-H due to the power consumptions caused by GCs. DFTL requires additional page read and write operations due to mapping table entry misses in the memory, causing additional energy consumption in both traces. As we expected, due to the high cost of full merge operations in the FAST FTL scheme, Fig.16 presents the most significant energy consumption for both erase and write operations during merge operation in the GC phase among compared FTL schemes. We also observe

LAST consumes less power than FAST. It is mainly due to the reduced merge operations than in FAST. In addition to high energy consumption by the merge operations in FAST and LAST FTL schemes, search operation to find a victim block every merge operation can consume the energy. This search operation takes time highly dependent on the search-count to find the victim block. We found that energy consumption almost linearly increases as it increases the number of the search-count, which is mainly because of the increased search time run at the device controller. Consequently we summarize our findings such that 1) minimizing merge operation cost in a hybrid FTL schemes, and 2) optimizing performance-energy efficiency of search operations in the merge operations in GC both are critical to make the hybrid-FTL based SSDs energy efficient.

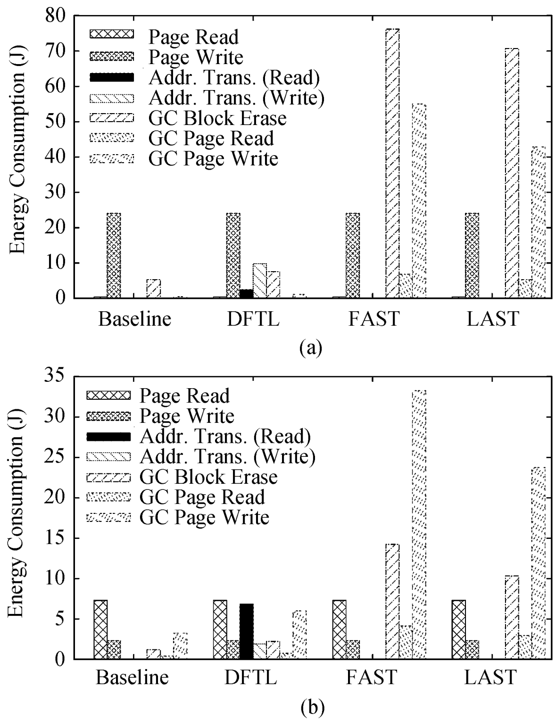


Fig.16. Energy consumption by different FTL schemes. (a) Financial trace. (b) TPC-H benchmark.

## 6 Related Work

An approach that uses log-buffers has been used to implement hybrid FTL schemes[12,17-19]. They try to address the problems of expensive full merges, which are inherent to any log-buffer based hybrid scheme, in their own unique way. However, all of these attempts are unable to provide the desired results.

Block Associative Sector Translation (BAST)[17] scheme exclusively associates a log block with a data block. In presence of small random writes, this scheme suffers from log block thrashing[12] that results in increased full merge cost due to inefficiently utilized log blocks.

Fully Associative Sector Translation (FAST)[12] allows log blocks to be shared by all data blocks. This improves the utilization of log blocks as compared with BAST. FAST keeps a single sequential log block dedicated for sequential updates while other log blocks are used for performing random writes. Thus, it cannot accommodate multiple sequential streams. Further, it does not provide any special mechanism to handle temporal locality in random streams.

SuperBlock FTL[18] scheme utilizes existence of block level spatial locality in workloads by combining consecutive logical blocks into a superblock. It maintains page-level mappings within the superblock to exploit temporal locality in the request streams by separating hot and cold data within the superblock. However, the three-level address translation mechanism employed by this scheme causes multiple OOB area reads and writes for servicing the requests. More importantly, it utilizes a fixed superblock size which needs to be explicitly tuned to adapt to changing workload requirements.

The recent Locality-Aware Sector Translation (LAST) scheme[19] tries to alleviate the shortcomings of FAST by providing multiple sequential log blocks to exploit spatial locality in workloads. It further separates random log blocks into hot and cold regions to reduce full merge cost. In order to provide this dynamic separation, LAST depends on an external locality detection mechanism. However, Lee *et al.*[19] themselves realized that the proposed locality detector cannot efficiently identify sequential writes when the small-sized write has a sequential locality. Moreover, maintaining sequential log blocks using a block-based mapping table requires the sequential streams to be aligned with the starting page offset of the log block in order to perform switch-merge. Dynamically changing request streams may impose severe restrictions on the utility of this scheme to efficiently adapt to the workload patterns.

Several recent studies were performed to study new FTL design and implementation for enterprise-scale SSDs. Park *et al.*[25] proposed a hybrid flash translation layer, called CFTL that exploits spatial and temporal localities in workloads. As the FTL maintains page-based and block-based FTL schemes, it could fully exploits the best benefits of each FTL scheme adaptive to workload changes. Budilovsky *et al.*[26] proposed an idea to use host memory to store the FTL mapping tables, which could not be entirely stored in SSD's memory due to small size of SRAM. For this, they developed a mechanism that a host can provide some hints on interfacing host memory to SSD device.

In addition to these studies on specific FTL design and implementation for improving random write performance, there has been a study on optimizing read throughputs for NAND flash based block devices[27]. Also, there has been a study on efficiently optimizing B-tree data structures over the FTL layer to optimize random write performance[28], which is independent of FTL optimization research. It will be interesting to study the performance study of their B-tree optimization techniques with various FTL schemes. Also Janus-FTL[29] has been proposed that FTL needs to be designed in a way that can choose a performance-cost wise optimal FTL among various FTLs (over the spectrum of page-based FTL to block-based FTL schemes) adaptive to dynamically changing workloads. On the contrary our paper proposed a specific FTL scheme that can be used as one of the FTL schemes on the two-end FTL design spectrum.

## 7 Conclusions and Future Work

We argued that existing FTL schemes, all based on storing a mix of page-level and block-level mappings, exhibit poor performance for enterprise-scale workloads with significant random write patterns. We proposed a complete paradigm shift in the design of the FTL with our Demand-Based Flash Translation Layer (DFTL) that selectively caches page-level address mappings. Our experimental evaluation using FlashSim with realistic enterprise-scale workloads endorsed DFTL's efficacy for enterprise systems by demonstrating that DFTL offered 1) improved performance, 2) reduced garbage collection overhead, 3) improved overload behavior and 4) most importantly unlike existing hybrid FTLs, it is free from any tunable parameters. As a representative example, a predominantly random write-dominant I/O trace from an OLTP application running at a large financial institution showed a 78% improvement in average response time due to a 3-fold reduction in garbage collection induced operations as compared with a hybrid FTL scheme. For the well-known read-dominant TPC-H benchmark, despite introducing additional operations due to mapping misses in SRAM, DFTL improved response time by 56%. Moreover, our DFTL scheme even outperformed the ideal page-based FTL scheme, improving the response time by 72% in OLTP trace. Ongoing research studies the feasibility of hybrid storage systems employing flash at appropriate places within the enterprise storage hierarchy along with hard disk drives. We have also compared the energy consumption of flash operations (page read, write, and block erase) for different FTL schemes for various enterprise-scale workloads. We enhanced our FlashSim[13] to include a write-back cache on an SSD for DFTL to study the benefits offered by the DFTL due to the increased cache space. Moreover we addressed the challenge of developing a performance model on SSDs and presented one of the possible methodologies that can be used to develop the prediction model. Our experimental results showed that SSD that implements DFTL can be predictable for performance.

We have identified several venues for future study. We discussed power-failure case in host; DFTL not only requires a minimal non-volatile memory to store global translation directory (GTD) but also enables a fast recovery by just scanning the GTD entries in it. However, current system does not support strong consistency between the cached mapping table and GTD. We plan to further investigate to support such strong consistency for power-failure issue on DFTL. In addition, we plan to extend our performance prediction model of SSDs to be able to build a model for lifetime prediction of SSDs to given workloads. It will be also interesting to study the effect of on-board cache on SSDs and develop cache algorithms that can work well for FTLs.

## References

[1] Gurumurthi S, Sivasubramaniam A, Natarajan V. Disk drive roadmap from the thermal perspective: A case for dynamic thermal management. In *Proc. the 32nd International Symposium on Computer Architecture*, June 2005, pp.38-49.

[2] Kim Y, Gurumurthi S, Sivasubramaniam A. Understanding the performance-temperature interactions in disk I/O of server workloads. In *Proc. the 12th International Symposium on High-Performance Computer Architecture* (*HPCA*), February 2006, pp.176-186.

[3] Mallary M, Torabi A, Benakli M. One terabit per square inch perpendicular recording conceptual design. *IEEE Transactions on Magnetics*, 2002, 38(4): 1719-1724.

[4] Chen J, Moon J. Detection signal-to-noise ratio versus bit cell aspect ratio at high areal densities. *IEEE Transactions on Magnetics*, 2001, 37(3): 1157-1167.

[5] Gulati A, Merchant A, Varman P J. pClock: An arrival curve based approach for QoS guarantees in shared storage systems. In *Proc. the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2007, pp.13-24.

[6] Tehrani S, Slaughter J M, Chen E, Durlam M, Shi J, DeHerren M. Progress and outlook for MRAM technology. *IEEE Transactions on Magnetics*, 1999, 35(5): 2814-2819.

[7] Shimada Y. FeRAM: Next generation challenges and future directions. In *Proc. the IEEE International Symposium on Applications of Ferroelectric*, May 2007.

[8] Leventhal A. Flash storage memory. *Communications of the ACM*, 2008, 51(7): 47-51.

[9] Lee S, Moon B. Design of flash-based DBMS: An in-page logging approach. In *Proc. the International Conference on Management of Data* (*SIGMOD*), August 2007, pp.55-66.

[10] Kim H, Ahn S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proc. the 6th USENIX Conference on File and Storage Technologies* (*FAST*), February 2008, pp.239-252.

[11] Small-block vs. large-block NAND flash devices. Technical

Report, TN-29-07, Micron. http://www.micron.com/products/nand/technotes, Jan. 2013.

[12] Lee S, Park D, Chung T, Lee D, Park S, Song H. A log buffer based flash translation layer using fully associative sector translation. *ACM Transactions on Embedded Computing Systems*, 2007, 6(3): Article No.18.

[13] Kim Y, Taurus B, Gupta A, Urgaonkar B. FlashSim: A Simulator for NAND flash-based solid-state drives. In *Proc. the International Conference on Advances in System Simulation* (*SIMUL*), September 2009, pp.125-131.

[14] Gupta A, Kim Y, Urgaonkar B. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. the 14th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, March 2009, pp.229-240.

[15] Hennessy J, Patterson D. Computer Architecture: A Quantitative Approach. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2006.

[16] Kim J, Kim J M, Noh S H, Min S, Cho Y. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 2002, 48(2): 366-375.

[17] Chung T, Park D, Park S, Lee D, Lee S, Song H. System software for flash memory: A survey. In *Proc. the International Conference on Embedded and Ubiquitous Computing*, August 2006, pp.394-404.

[18] Kang J, Jo H, Kim J, Lee J. A superblock-based flash translation layer for NAND flash memory. In *Proc. the 6th International Conference on Embedded Software* (*EMSOFT*), October 2006, pp.161-170.

[19] Lee S, Shin D, Kim Y, Kim J. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 2008, 42(6): 36-42.

[20] Karedla R, Love J S, Wherry B G. Caching strategies to improve disk system performance. *IEEE Transactions on Computer* (*TC*), 1994, 27(3): 38-46.

[21] Kawaguchi A, Nishioka S, Motoda H. A flash-memory based file system. In *Proc. the Winter 1995 USENIX Technical Conference*, Jan. 1995, pp.155-164.

[22] Bucy J S, Ganger G R. The DiskSim simulation environment version 3.0 reference manual. CMU, January 2003.

[23] Ban A. Flash file system. United States Patent 5404485, April 4, 1995.

[24] Zhang J, Sivasubramaniam A, Franke H, Gautam N, Zhang Y, Nagar S. Synthesizing representative I/O workloads for TPC-H. In *Proc. the 10th International Symposium on High Performance Computer Architecture* (*HPCA*), Feb. 2004, pp.142-151.

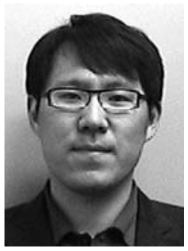
[25] Park D, Debnath B, Du D H C. A workload-aware adaptive hybrid flash translation layer with an efficient caching strategy. In *Proc. the 19th IEEE Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, July 2011, pp.248-255.

[26] Budilovsky E, Toledo S, Zuck A. Prototyping a high-performance low-cost solid-state disk. In *Proc. the 4th Annual International Conference on Systems and Storage*, May 30-June 1, 2011, Article No. 13.

[27] Choudhuri S, Givargis T. Performance improvement of block based NAND flash translation layer. In *Proc. the 15th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis* (*CODES+ISSS*), September 30-October 3, 2007, pp.257-262.

[28] Wu C H, Kuo T W, Chang L P. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Trans. Embed. Comput. Syst.*, 2007, 6(3): Article No. 19.

[29] Kwon H, Kim E, Choi J, Lee D, Noh S H. Janus-FTL: Finding the optimal point on the spectrum between page and block mapping schemes. In *Proc. the 10th ACM International Conference on Embedded Software*, Oct. 2010, pp.169-178.



**Youngjae Kim** is a computer science R&D staff member for the National Center for Computational Sciences at Oak Ridge National Laboratory, USA. He received the Ph.D. degree in computer science and engineering from the Department of Computer Science and Engineering of the Pennsylvania State University, USA, in 2009. His research interests include operating systems, parallel I/O and file systems, and storage systems.

**Aayush Gupta** is a research staff member at IBM Almaden Research Center. He received the Ph.D. degree in computer science and engineering from the Department of Computer Science and Engineering of the Pennsylvania State University in 2012. His research interests include operating systems, file and storage systems, and emerging storage technology.

**Bhuvan Urgaonkar** received the BTech (Honors) degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1999, and the Ph.D. degree in computer science at the University of Massachusetts, USA, in 2005. He is currently an associate professor in the Department of Computer Science and Engineering at the Pennsylvania State University. His research interests are in the modeling, implementation, and evaluation of distributed systems, operating systems, and storage systems.