S-FTL: An Efficient Address Translation for Flash Memory by Exploiting Spatial Locality

Song Jiang*, Lei Zhang†, XinHao Yuan†, Hao Hu† and Yu Chen†
*The ECE Department
Wayne State University Detroit, MI, 48202, USA
Email: sjiang@ece.wayne.edu
†Department of Computer Science and Technology
Tsinghua University
Beijing, China
Email: {sosilent.lzh,xinhaoyuan,haohu.th,chyyuu}@gmail.com

***Abstract*—The solid-state disk (SSD) is becoming increasingly popular, especially among users whose workloads exhibit substantial random access patterns. As SSD competes with the hard disk, whose per-GB cost keeps dramatically falling, the SSD must retain its performance advantages even with low-cost configurations, such as those with a small built-in DRAM cache for mapping table and using MLC NAND. To this end, we need to make the limited cache space efficiently used to support fast logical-to-physical address translation in the flash translation layer (FTL) with minimal access of flash memory and minimal merge operations. Existing schemes usually require a large number of overhead accesses, either for accessing uncached entries of the mapping table or for the merge operation, and achieve suboptimal performance when the cache space is limited. In this paper we take into account spatial locality exhibited in the workloads to obtain a highly efficient FTL even with a relatively small cache, named as *S-FTL*. Specifically, we identify three access patterns related to spatial locality, including sequential writes, clustered access, and sparse writes. Accordingly we propose designs to take advantage of these patterns to reduce mapping table size, increase hit ratio for in-cache address translation, and minimize expensive writes to flash memory.**

We have conducted extensive trace-driven simulations to evaluate S-FTL and compared it with other state-of-the-art FTL schemes. Our experiments show that S-FTL can reduce accesses to the flash for address translation by up to 70% and reduce response time of SSD by up to 25%, compared with the state-of-the-art FTL strategies such as FAST and DFTL.

I. Introduction

The flash-memory based solid-state disk (SSD) is rapidly gaining ground in the mainstream storage market traditionally held by the hard disk. By using semiconductor chips instead of rotating disk platters to store data, SSD offers strong resistance to extreme shock, vibration, and temperature, as well as low power consumption, which help it widely adopted in military and aerospace industries running mission-critical applications. Furthermore, while SSD can potentially achieve a high I/O throughput, especially for random access, which has been devastating hard disk's performance, SSD becomes increasingly popular in the enterprise computing environment as a performance booster, such as in Sun Storage 7000 Unified Storage Systems [1] and the SSD-based computing platforms to be built by Google and Baidu. However, as the hard disk keeps its trend of rapidly falling price, the significantly uncompetitive price of SSD makes it to hold only a niche industrial and high-priced mobile computing markets. Today's per-GB price of a low-end or middle-level flash SSD is around 20 - 100 times higher of that of the hard drive. For high-end flash SSD, the ratio can be 500 or even higher [8]. To contain the cost of flash SSD and make its price more competitive, manufacturers need to squeeze higher performance from SSDs with configurations of relatively low cost, such as using multi-level cell chips and small DRAM cache size [9].

The NAND flash memory, which is used in SSD for data storage, may use single-level cell (SLC) or multi-level cell (MLC) technologies. The SLC flash stores only one bit of data in one memory cell, while the MLC flash stores two or more bits in one memory cell. The increased density of the MLC flash significantly reduces the production cost and increases the capacity of SSD, making it popular with the low-end and middle-level SSDs. While for a flash memory the write operation is usually 5-10 times slower than the read operation in terms of page access time, the write time of MLC flash is additionally more than three times higher that of the SLC flash, making write a very expensive operation for the MLC flash. In addition to the writes directly requested by users, SSD's internal management schemes, especially those for the address translation, can produce additional write operations. To maintain a high performance for the MLC SSD, these write operations should be minimized.

As in the hard disk, SSD has a DRAM memory as buffer cache for caching data pages and mapping table. Because DRAM is much faster than the flash, especially for writes, it is critical to maintain a high hit ratio in the buffer cache for a high SSD's performance. To contain the cost of SSD to make it competitive to the hard disk, manufacturers prefer to provide a smaller DRAM cache. The challenge is whether we can still maintain a high hit ratio with a reduced cache. While the issue on achieving a high hit ratio for data caching in a buffer of limited size is well studied [13], [12], [14], it is not yet clear how to minimize the number of flash memory access associated with address translation with limited cache space for mapping table. Like the hard disk, SSD provides

a logical address abstraction for external software to specify locations of their requested data. Internally SSD maintains a flash translation layer (FTL) to translate a logical address to its corresponding physical address on the flash. SSD is a block device, in which the read and write operations are conducted in the unit of page, whose size usually ranges from 0.5KB to 4KB. Accordingly, FTL is used to translate a logical page number (LPN) to a physical page number (PPN). If there are not any restrictions on the way of mapping LPNs to PPNs, the size of the table recording the LPN-to-PPN mapping for facilitating the translation is proportional to the number of pages in the SSD. This is referred to as the *page-level FTL*, in which the so-called page-level mapping is used. The table of the page-level FTL is almost impossible to be fully held in the cache of any large-size SSDs. An an example, a 128GB SSD can require 256MB cache for the table, in addition to a usually larger cache space for caching data. If only a fraction of mapping table is cached, a substantial number of access misses on the table and writebacks of dirty table entries could add to the overhead of the flash access. As we know, some mapping tables used in the system software, such as page table in the operating system for translating virtual address to physical address and inodes in the file system for translating in-file logical address to on-disk logical address, do not have any mapping restrictions either. However, these tables do not have similar performance concern, because they are cached in the main memory of much larger size, instead of in-device cache.

To contain the size of a mapping table, basically there are two methods that impose restriction on the mapping of a source address to a destination address. The first one is to allow a source address to be mapped to only one or a small set of destination addresses. This is the method used in the direct mapped or set-associative processor cache or TLB. It can effectively eliminate the mapping table. The second method is to exploit or create mapping regularity between the source addresses and destination addresses. This method is most suitable to the hard disk when it maps the logical block addresses (LBAs) exposed to the external to its physical addresses, because it intentionally maps contiguous logical addresses to contiguous physical addresses to allow programmer to exploit the hard-disk's high throughput associated with sequential access. By exploitation of this regularity, the translation can be done using a small mapping table about segments of contiguous mappings and some simple calculations.

Unfortunately, flash memory has the unique "erase before write" requirement, i.e., a page that has been programmed, or has stored data, must first be erased before new data can be written into it. In addition, the erase operation is carried out in a much larger granularity, named as block, than page. A block usually contains 32-128 pages. Because in-place write is not allowed, the *block-level FTL* in SSD, which attempts to maintain the mapping regularity similar to that for the hard disk, is prohibitive, though the size of the mapping table is small. In the block-level FTL, its mapping table records only the mapping from logical block number (LBN) to physical

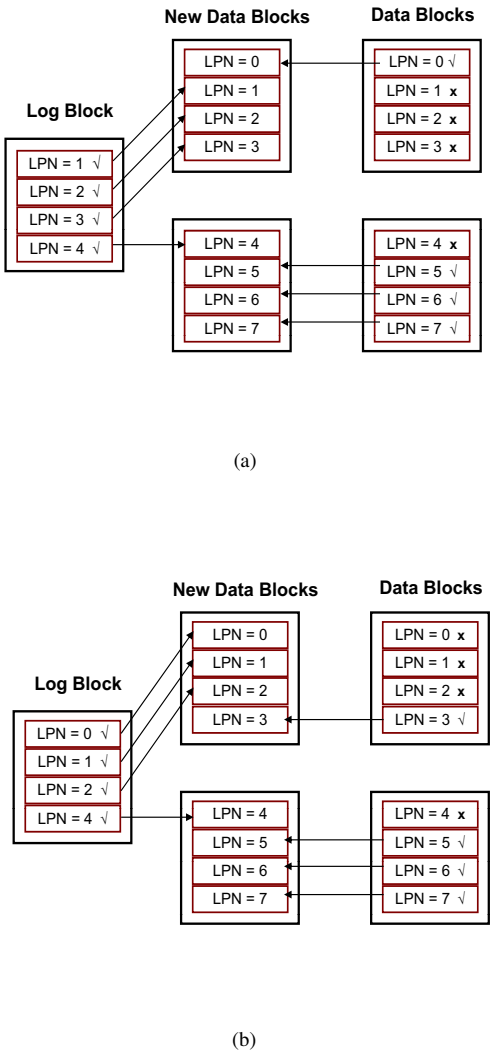


Fig. 1. Illustration on how the block-level FTL incurs high merge operation cost due to its inability on exploiting readily available contiguity in the page mapping. In the example, we assume each block has four pages. For each mapping entry, "$\surd$" indicates that the corresponding physical page is valid, or contains valid data, while "$\times$" indicates an invalidated physical page. (a) Eight pages have to be copied to reclaim the log block, though the LPNs in the block are contiguously laid out. Note that if the four LPNs in the block are 0, 1, 2, and 3, instead of 1, 2, 3, and 4, the log block can be simply switched into a data block without any page copying. (b) Eight pages have to be copied to reclaim the log block, though most of the LPNs in the block are contiguously laid out.

block number (PBN) by ensuring that the offset of an LPN in the logical block is the same as that of its corresponding PPN in the physical block, or, within a block contiguous logical pages are mapped to contiguous physical pages. Once one page in a block is to be updated, it will be written into an erased block (instead of an in-place overwriting), and the original copy of the page is invalidated. To enforce the mapping regularity required by the block-level FTL, all the

other pages in the block have to be copied into the erased block, introducing overhead flash access. Though its mapping table is proportional to the number of blocks and is small enough to be held in the cache, its overhead flash accesses (reads and writes due to the copying) are unaffordable. To alleviate the high cost, a hybrid FTL was proposed to use a small number of blocks, called log blocks, as a buffer to receive page writes. Log blocks are managed with the page-level mapping. The pages in the log blocks are moved into data blocks managed with the block-level mapping in a process called merge operation, when erased pages for new writes are running out. While the hybrid FTL can delay the copying of pages from their residing block to an erased block, its merge operation can be of high cost. As an example, if the pages of a log block belong to multiple logical blocks, they will be copied into multiple erased blocks, and valid pages in the corresponding physical blocks are all needed to be copied into the erased blocks.

Apparently FTL cannot use the first aforementioned method, which is used in the processor cache, because in-place writes are not allowed. Because both the block-level FTL and the hybrid FTL require rigid mapping regularity and use a large number of page copying or high-overhead merge operation to enforce the regularity, they cannot be efficiently used. For example, even if contiguous pages in a log block are also logically contiguous but the first page's LPN is not of multiples of block size, all the pages need to be copied into an erased block, which is illustrated in Figure 1 (a). In another example, if only a few contiguous pages in a log block are not logically contiguous, all the pages need to be copied (see illustration in Figure (b)). This deficiency and its consequent high cost are due to the fact that the block-level FTL takes efforts only to create its predefined regularity, instead of exploiting available regularity that has been naturally existing in the write request stream as much as possible.

In this paper, we propose a design that exploits this readily available contiguity to reduce mapping table size. Though we cannot guarantee that the table will be always small enough to be entirely held in cache, our design maximizes the contents of the table cached in the buffer to improve table hit ratio without requiring merge operations for the block-level FTL. In other words, any workloads that contain sequential writes would help reduce mapping table and improve address translation efficiency. Sequential write is a special spatial locality exploited in our work. In general, spatial locality, which refers to the access pattern in which when a page is accessed, its logically neighboring pages are likely to be accessed soon, is a commonly observed programs' access behavior. Considering the spatial locality, and the fact that the flash memory is accessed at the page granularity, we pack mapping information for neighboring LPNs in the same flash page, called mapping page, and manage the mapping table with the mapping page as a unit to improve hit ratio in the cache for address translation. In case where the spatial locality for write is very weak, there would be dirty table entries sparsely dispersed among many cached mapping pages. When one of the pages has to be evicted from the cache, it must be written back to the flash even though it may contain only a few dirty entries and write is a very expensive operation. To address the issue arising from weak spatial locality on write, we choose to cache the small number of dirty entries to avoid the expensive writebacks. As our FTL scheme is designed around spatial locality, we name the scheme as spatial-locality-aware FTL, or *S-FTL* in short.

In summary, we made the following contributions on the design of efficient FTL in the paper.

- We designed a scheme to eliminate the need for merge operations and to reduce mapping table size by exploiting access regularity exiting in the write request stream.
- Our scheme is designed both to take advantage of strong spatial locality by using mapping page as caching unit and to address the overhead issue with weak spatial locality.
- We extensively evaluate the *S-FTL* scheme with representative traces showing that it can reduce the number of accesses to the flash for address translation by up to 70% and reduce access response time of SSD by up to 20%, compared with the state-of-the-art FTL strategies such as FAST and DFTL.

The remaining of the paper is organized as follows. Section II discusses the related work. Section III describes the design and implementation of *S-FTL*. Section IV describes and analyzes experiment results, and Section V concludes.

II. Related Work

As flash-memory-based SSD is built with semiconductor chips without using any mechanical moving parts, users may expect SSD to have superior random-access performance. The fact is that SDD's random write performance is much worse than its sequential access performance (usually around 10% of the sequential access throughput [2]). For the low-end SSD, its performance can even be as low as that of the hard disk [8]. This performance observation is mainly attributed to the flash's "erase-before-write" characteristic and the FTL scheme designed to accommodate it. Realizing the critical role played by FTL for the SSD's promised high performance to be fully delivered, researchers and practitioners have done much work to improve FTL so that it would not be a performance bottleneck.

An intuitive method to implement FTL is to use page-level mapping in which a mapping table similar to the page table for virtual memory in OS is adopted. Because it is infeasible to hold such a big table in the cache, the block-level FTL was proposed, in which an LPN's offset in the logical block is the same as PPN's offset in the physical block and the PPNs in a block are contiguous [20], [10]. As discussed in Section 1, it is very expensive to maintain such a rigid mapping regularity defined by the block-level FTL for a logical page immediately after it is written. Therefore, the hybrid FTL was proposed to set aside a small number of log blocks to hold newly written pages using the page-level mapping. While majority of pages are still expected to stay in the data blocks managed by the block-level mapping scheme, a log block can be reclaimed by merging its pages with pages in other blocks to recover

data blocks that follow block-level-FTL's mapping regularity in a garbage collection operation. As there are critical issues associated with the hybrid FTL, several of its variants have been proposed.

If pages from any logical blocks can be written into the same log block, a log block may contain pages mapped from different logical blocks. When the log block is to be reclaimed, its pages will be copied into multiple erased blocks, and valid pages in other physical blocks are also needed to be copied into the erased blocks to turn them into data blocks. To reduce the copying cost, the Block Associative Sector Translation (BAST) scheme allows a log block dedicated to one data block [16]. If a data block has its dedicated log block, write requests to the data block will be fulfilled in the log block. Accordingly, only one data block is involved when the log block is reclaimed. However, if a write request is sent to a data block to which a log block is not yet assigned, it has to reclaim a log block assigned to another data block. If there are many small random writes, this scheme suffers from "log block thrashing" [18], where under-utilized log blocks are transferred frequently among data blocks with high garbage collection cost. To address this issue, the Fully Associative Sector Translation (FAST) scheme [18], [17] is proposed to allow a log block to be shared by all data blocks to increase the utilization of the log block, which unfortunately increases garbage collection cost.

In addition, the FAST scheme designates a log block dedicated for sequential writes with the hope that the log block can be filled with pages following the regularity required by the block-level FTL. Recognizing that there could be multiple sequential streams mixed in a workload, Lee et al. proposed the Locality-Aware Sector Translation (LAST) scheme, which uses multiple log blocks to exclusively receive sequential requests attempting to preserve the spatial locality [19]. Use of these dedicated log blocks can reduce garbage collection cost if they can turn into data blocks without copying their pages to other erased blocks (*switch merge*). To make this design effective, one has to ensure that pages in a sequential write meet three conditions: (1) The pages must be written to the physical pages (in one log block) whose offsets equals to their respective offsets in the logical block; (2) The writing must start from the first page of a log block, and carry out contiguously according to the rule of sequential programming within a block, required by today's popular large-block flash [3]; and (3) Significant number of pages in a log block, whose size is usually 64 pages or larger, must be filled by sequential requests before the log block is reclaimed. Although sequential requests are common in many workloads, it would be rare to have all these conditions met. In fact, the deficiency with these schemes are due to their adherence to the block-level-FTL's mapping regularity. In contract, The S-FTL scheme gives up the adherence, and is able to exploit any sequentiality available in the write requests to reduce mapping table size. For example, if the size of every request is two (or four) contiguous pages, the table size can be reduced to its half (or its quarter, respectively) in S-FTL. However, this weak sequentiality would be of little value in the LAST scheme.

As long as pages are in the same logic block, they are mapped to the same physical data block in the hybrid FTL regardless of their difference on update frequency. This raises another issue with block-level-FTL's mapping regularity. That is, frequently updated pages (hot pages) can be significantly mixed with infrequently updated pages (cold pages) in the same blocks. This makes garbage collection expensive because it has to copy valid cold pages of a block each time the invalidated hot pages in the block are collected. To address this issue, the *Superblock* scheme combines consecutive blocks into a superblock, in which page-level mapping is applied [15]. Without enforcing the block-level-FTL's mapping regularity within a superblock, hot and cold pages can be separated into different blocks. Instead of distinguishing hot and cold pages within a superblock, another scheme, Locality-Aware Sector Translation (LAST), distinguishes the log blocks for holding either non-sequentially-written hot pages or non-sequentially-written cold pages [19]. However, because both the number of pages in a superblock and the number of log blocks for non-sequential-written pages are relatively small, their effectiveness is limited. In contrast, S-FTL allows any page to be mapped into any block in the flash memory. Therefore, hot pages tend to move into the same blocks together during their re-writes.

Recently the DFTL scheme was proposed to use page-level mapping to avoid the high cost required for maintaining the mapping regularity in hybrid FTL [11]. As the mapping table of a page-level FTL is too large to be held entirely in the cache, DFTL exploits temporal locality to cache recently used mapping entries, each entry specifying which PPN an LPN is mapped to. The full mapping table is stored on the flash, where the mapping entries are packed into translation pages in the order of their LPNs. A small set of blocks are designated as translation blocks to hold the translation pages. Like the blocks holding user data, the translation pages are also managed with page-level mapping, with in-cache global translation directory recording where the translation page for a range of LPNs is. While S-FTL also adopts page-level mapping to avoid merge and garbage collection operations, it caches entire translation pages so as to benefit from the effect of mapping-entry prefetching. Meanwhile, by exploiting sequential writes, S-FTL does not necessarily have to spend a page-size space to hold a translation page in cache. In contrast, DFTL does not take any efforts on efficient use of cache space for storing mapping information – it simply disregards any mapping regularity exploited by the block-level or hybrid FTL.

III. The Design of S-FTL

There are several objectives to achieve in the design of spatial-locality-aware FTL (S-FTL). First, it should not impose the mapping regularity that is required by the block-level and hybrid FTLs for reduction of the mapping table size but carries a high garbage collection cost. Second, it should be able to benefit from sequential writes to reduce mapping table and make limited cache space well utilized. Third, it

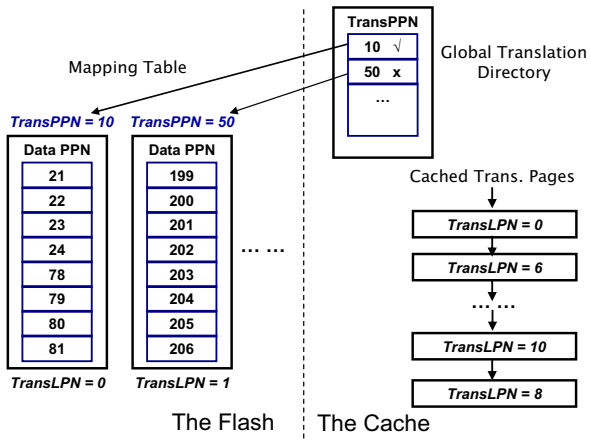


Fig. 2. Illustration of management of the mapping table in S-FTL. The complete table is stored in the flash. It consists of multiple translation pages, each recording m PPN entries. The LPN mapped to the nth PPN in a translation page with its TransLPN of k is $m * k + n$. For example the first mapping entry in the second translation page (TransLPN = 1) states that logical data page 8 ($m = 8$, $k = 1$, and $n = 0$) is mapped to physical page 199. The nth entry in the Global Translation Directory (GTD) records the transPPN for the translation page whose TransPPN is n. Cached translation pages are linked in the LRU list. Each entry of the GTD also indicates whether the corresponding translation page is cached: "$\surd$" for *cached* and "$\times$" for *uncached*. As an example, when logical page (LPN 9) is requested, we know its TransLPN is 1 ($9 \div 8 = 1$). The GTD shows that the translation page is not cached and its TransPPN is 50. Then the physical page 50 is retrieved and cached, and PPN 200 is obtained at the offset 1 (*9 modulo 8 = 1*) of the translation page. Note that we intentionally use a very small m ($m = 8$) for easy illustration. The actual number of entries in a translation page can be 512 or more.

should be able to minimize the impact of sparse writes on the efficiency of writing back mapping-table pages. To this end, S-FTL adopts the page-level mapping and uses page as unit to manage mapping entries. It exploits sequentiality in the write requests to create a concise representation of in-cache mapping table pages. In addition, S-FTL holds a small number of dirty mapping entries in the cache to avoid writing mostly clean table pages to the flash memory.

A. Caching Mapping Table

While S-FTL is a page-level FTL, in which any logical page can be mapped to any physical page, we need to record the mapping for any logical page in a table, whose size is usually proportional to the flash memory size and too large to be entirely held in the cache. Therefore, the table is resident in the flash with its entries packed into pages that are called translation pages, in contrast to the data pages that hold users' data. The entries in a translation page are placed in the ascending order of the LPNs of data pages represented by the entries. Similar to DFTL, we set a small number of translation blocks to hold translation pages. A translation page also has its logical page number (TransLPN) and physical page number (TransPPN), and their mappings are also managed at the page level. The table holding these mappings, or the Global Translation Directory (GTD), is small and entirely resident in the cache. The TransLPN of a translation page is determined by LPNs of the data pages whose mappings are recorded in the translation page. Recently used translation pages are cached and organized in an LRU list (or LRU stack). The formats of the mapping table and GTD are illustrated in Figure 2.

When a request is received with the LPN of requested data, the LPN is divided by the translation page size in terms of number of entries in the page, to get TransLPN of the translation page containing the entry for this LPN. The TransLPN is used to check the GDT to see if the translation page is cached. If yes, the PPN of the requested page is obtained by accessing the corresponding translation page. If it is a miss, the TransPPN of the translation page is obtained from the GTD. Using the TransPPN, the translation page can be read from the flash and placed at the top of the LRU stack. From this page, the PPN of requested data is found. By choosing an entire translation page as caching object, instead of the mapping entries that have been actually used, S-FTL exploits spatial locality by enabling prefetching of mapping entries. There are two advantages for the choice. First, a page is the minimal access size on the flash. Caching an entire translation page does not increase the cost of accessing the page on the flash. Second, when SSD is used as a storage device to support file I/O, cold misses, which are due to first-time access of pages, can be of significant percentage. This is because increasingly large DRAM buffer cache in the host computer can satisfy many requests for reused data. Prefetching is the only mechanism that can turn a cold miss into a hit. Apparently, caching an entire page may consume more space than caching only requested translation entries in the page. S-FTL needs to reduce this space consumption.

B. Exploiting Sequential Writes to Shrink Translation Pages

With limited cache space, a translation page can be considered too large to be held entirely in cache, especially when this strategy is compared with the block-level FTL (or the hybrid FTL). For an SSD whose block has 64 pages and whose page can hold 512 mapping entries, the block-level FTL needs only eight entries in its mapping table to represent the 512 entries in a translation page. As we have known, the space efficiency of the block-level FTL is achieved by artificially creating sequential writes to form the predefined mapping regularity, which incurs high garbage collection cost. In contrast, S-FTL takes advantage of sequentiality existing in the users' requests to reduce translation page size.

We have the following observation. For k pages with LPN_i (*i = 0, 1, ..., k-1*) continuously mapped to k contiguous physical pages of PPN_i (*i = 0, 1, ..., k-1*), we have $LPN_i - LPN_0 = i$, if these k pages are logically sequential, and $PPN_i - PPN_0 = i$ (*i = 1, 2, ..., k-1*). So we have $PPN_i = PPN_0 + i = PPN_0 + (LPN_i - LPN_0)$ (*i = 0, 1, 2, ..., k-1*). Therefore, if we record the mapping entry about LPN_0 and PPN_0, and know the distance between LPN_0 and LPN_i, we can obtain corresponding PPN_i for any LPN_i (*i = 1, 2, .., k-1*) without actually recording these $k-1$ mapping

entries. There are two methods to know the distance between LPN_0 and LPN_i. One is to directly record this distance. However, it can consume substantial space. Assume there are 512 entries in a translation page and four bytes are used to represent a PPN, it will use 9 bits to record the distance for each entry or use about 25% of a page space to record the distance for every entry in the page. Another problem is that once the sequence is broken because of rewriting to page PPN_0, all the recorded distances measured from PPN_0 have to be invalidated or updated, which is expensive and inflexible. The second method is to use a bitmap to record contiguity in their PPNs between two neighboring entries. Specifically, we associate one bit with each mapping entry (LPN_i, PPN_i). If $PPN_i = PPN_{i-1} + 1$, the bit with entry (LPN_i, PPN_i) is 0. Otherwise, it is 1. Note that for any two neighboring entries, we always have $LPN_i = LPN_{i-1} + 1$ and actually only PPN_i is recorded in the entry (see Figure 1). Then a sequential write of k pages would probably produce a bitmap *100...0* (the number of *0*s is $k-1$). As examples, the bitmaps for two translation pages, whose TransLPNs are 0 and 1, shown in Figure 1 are *10001000* and *10000000*, respectively. If we name the entry whose bit is 1 as head entry of all its immediately following entries whose bits are 0, then the distance of an entry to its head entry is number of 0s between the entry and the head entry, including this entry. In particular, the distance for head entry itself is 0. As the largest possible distance is limited, which is one less than the number of entries in a translation page, the circuitry in the SSD controller can use bit-wise operations on the bitmap to efficiently calculate the distance by counting the number of '0's.

As we maintain the bitmap for a translation page, only the head entries need to be kept and all other entries can be dropped without losing any mapping information. The larger the distance can be, the more space can be saved. Assume there are 512 entries in a translation page, and four bytes for a PPN, a translation page would be shrunk to 70Bytes, about 3.4% of its original size, including the bitmap, if there is only one head entry in a translation page. Compared with 24Bytes, or about 1.2% of a page size, needed for the block-level FTL to cover the mapping information in a translation page, S-FTL effectively produces a very concise representation for the translation page. It is not necessary to require that LPN of the first page of a data block have a zero in-block offset so that S-FTL can reduce the mapping table. As each '0' in a bitmap implies that a mapping entry can be dropped, the number of '0's in a bitmap indicates how much a translation page can be shrunk by. As any sequential write involving n pages produces $n-1$ '0's ($n > 1$), the sequentiality becomes opportunities for S-FTL to take for reducing mapping table size. In contrast to the rigid mapping regularity required by the block-level FTL as well as the high cost of garbage collection to restore such regularity, S-FTL is flexible and of light weight.

While a long write sequence may help reduce the number of head entries (see Figures 3 (a) and (b)), a small write into a previously written long sequence can produce additional head entries by splitting the long sequence into small sequences (see

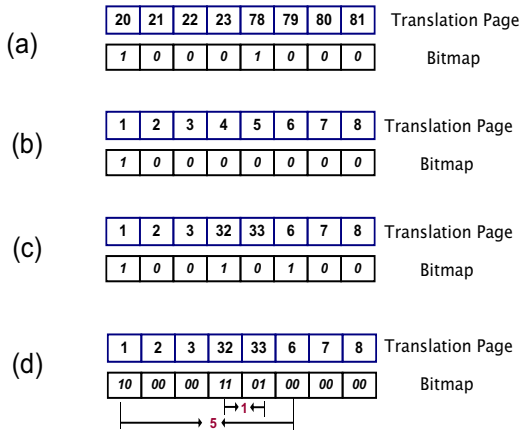


Fig. 3. Illustration on how the bitmap of a translation page is updated in response to new writes. (a) contents of current page and its bitmap. (b) updated after all pages are sequentially written. (c) further updated after another write to the fourth and fifth entries. Note that the number of head entries is increased from 1 to 3 when only one bit is used in the bitmap for an entry. (d) If two bits are used for each entry, only two head entries are needed. The distance of the fifth entry to its head entry is 1 and the distance of the sixth entry is 5.

Figure 3 (c)). The longer a sequence is, the more likely the sequence is to be broken by small writes and the harder for S-FTL to achieve a significant reduction of translation page. To address the issue, we can associate two bits with each mapping entries, instead of one bit. In the two bits, the first one indicates whether the corresponding entry is a head entry ('1' for yes), and the second bit indicates whether the entry belongs to the broken long sequence or to the local short sequence ('0' for the broken long sequence). That is, even if a long sequence is broken by a short sequence, the entries that follow the short sequence but belong to the long sequence can still be removed and their corresponding pages can keep using the head entry of the long sequence to obtain their PPNs (see Figure 3 (d)). Specifically, in the search of head entry for a page, the entries whose bits are '01' or '11' are passed if the second bit of this page is '0'. However, if a sequence of pages whose second bits are '1's is broken by a new write, this sequence has to be shortened, or split into smaller sequences, each with its own head entry. This problem could be addressed by further increasing the number of bits associated with each page. However, the increased space cost is usually not justified.

With the translation page shrinking technique, a translation page has two representations: its *in-flash form* with fixed page size and its *bitmap form* whose size depends on the contiguity of the entries in the page and can change with new writes to the page. Note that these two forms are fully convertible to each other. When a translation page is loaded into the cache, S-FTL calculates the size of its bitmap form. If the size is smaller than 80% of the page size, S-FTL caches it in its bitmap form. Otherwise, its in-flash form is used. S-FTL monitors the size of

a cached page's bitmap form, no matter if the form is actually used. Whenever a write occurs to a page, the size of its bitmap form is updated. If it is smaller than 80% of the page size and currently in the in-flash form, S-FTL converts it into its bitmap form. If it is larger than 90% of the page size and currently in the bitmap form, S-FTL converts it into its in-flash form. We use these thresholds to accommodate the additional cost for searching head entries in the bitmap form and to avoid frequent conversions between these two forms.

All cached translation pages, regardless of which forms they take, are managed by the LRU replacement policy. When a translation page is loaded into cache, it is placed at the top of the LRU stack and not recently used translation page is replaced from the bottom of the stack. For the page to be replaced, the space it holds would be variable if it has been in the bitmap form. Understandably among pages of similar temporal locality, it is preferred to replace the ones consuming larger cache space and leave the ones with smaller space in cache for a longer period of time [12]. For the management of SSD cache, we take a simple strategy to address the issue. We set up two yardsticks in the LRU stack: the first one is at the 1/3 stack size from the stack top, and the second one is at the 2/3 stack size from the top. When a page to be replaced is in the bitmap form and its size in the form is less than 30% of a page size, S-FTL re-inserts this page into the stack at the position indicated by the first yardstick. Similarly, if the page is less than 60% but more than 30% of the page size, S-FTL re-inserts this page into the stack at the position indicated by the second yardstick. A re-inserted page will be flagged, and the flag is removed if the page is used in the address translation. A flagged page will not be given the second chance to be inserted.

C. Improving Efficiency of Translation Page Writeback

If a translation page to be replaced is dirty, it needs to be written back to the flash and the corresponding GTD entry is updated, instead of being simply discarded. As flash write is much more expensive than flash read, the write efficiency, which is measured as the percentage of dirty entries in a translation page, becomes an important factor affecting address translation performance. When write requests exhibit very weak locality and writes are sparsely dispersed over many pages, each translation page may contain only a few dirty entries. It would not be cost effective to write the whole page back only because of the few dirty entries. To improve the writeback efficiency, we take out the dirty entries of a translation page, keep them in the cache and simply discard the page, if the page to be replaced contains dirty entries less than 5% of total entries in the page. When this translation page is loaded from the flash next time, these cached dirty entries are moved into the page. When the page is loaded from the flash next time, these cached dirty entries can be written back when the SSD is idle. In our experiments, we suspend caching of dirty entries once the cache space allocated for the purpose is full. In the experiments, we found that a minimal amount of allocation, space for 50 entries by default, serves the purpose in most cases.

IV. Performance Evaluation

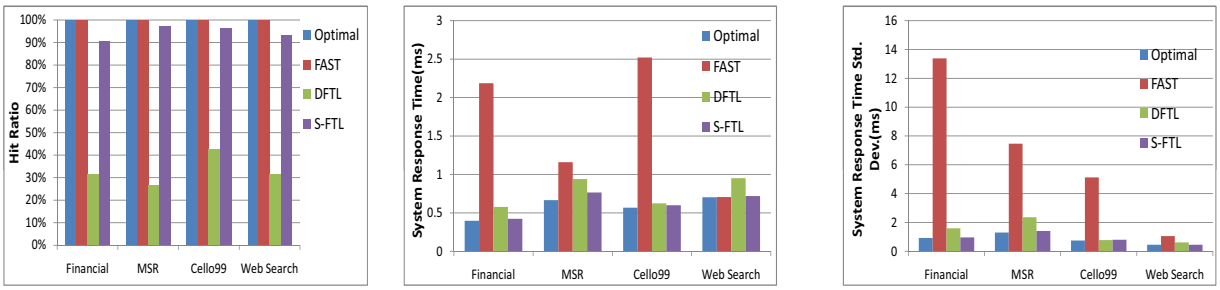
We used trace-driven simulation to evaluate S-FTL for managing a large-block NAND SSD of 32GB, and compared it with DFTL, FAST, and the optimal FTL. For the optimal FTL, we assumed a page-level mapping and an infinitely large cache for holding its mapping table, allowing it to represent the minimal overhead any FTL can possibly have. We use the ideal FTL to show how far S-FTL is close to the optimal performance. The simulator is obtained by adding an S-FTL module into the FlashSim simulator [11], which has been used for evaluating DFTL and FAST. For the large-block SSD, the page size is set as 2KB, and block size is 128KB [11]. The cache size for address translation is set as what is needed for the block-level FTL, which is 64KB. We leave 3% of the total SSD capacity used as log buffers to accommodate out-of-place writes. In the simulator, the page read time is set as 0.12ms, page write time is 0.41ms, and the block erase time is 2.0ms [3]. In the evaluation, we use one bit for each page to record its sequentiality in S-FTL for experiments except that presented in Section 4.2.4, where the option of using two bits is evaluated.

A. Workloads and Evaluation Metrics

Workloads	Request Size (KB)	Read (%)	Seq. Read (%)	Seq. Write (%)
Financial	9.85	23.16	0.71	0.40
MSR	9.31	18.45	13.86	2.60
Cello99	9.87	57.34	2.6	0.92
Websearch	32.29	99.98	6.47	0.94

TABLE I
CHARACTERISTICS OF SELECTED REAL-WORLD WORKLOADS, INCLUDING THEIR AVERAGE REQUEST SIZES, PERCENTAGE OF READ REQUESTS, PERCENTAGE OF READ REQUEST CONTIGUOUS TO ITS PREVIOUS ONE, AND PERCENTAGE OF WRITE REQUEST CONTIGUOUS TO ITS PREVIOUS ONE.

In the evaluation we choose several real-world and synthetic traces to study performance impact of different FTLs. As shown in Table I, we employ four read-world traces that represent different application domains and a broad range of enterprise-scale workloads. Among the traces, the *Financial* trace is a write-dominant I/O trace from an OLTP application running at a financial institution provided by the Storage Performance Council (SPC) [4]. *MSR* is also a write-dominant trace collected on servers at Microsoft Research Cambridge [6]. The *Celllo99* trace is mixed with substantial reads and writes and is collected from a time-sharing server running the HP-UX operating system at HP Laboratories [5]. The *Websearch* trace is a read-dominant web search engine trace from SPC [4]. The major statistics on access behaviors exhibited in the traces are included in Table I. In addition to the real-world traces, we generate a synthetic trace to comprehensively study the impact of S-FTL's prefetching



(a) Hit Ratio (b) System Response Time (c) System Response Time Std. Dev

Fig. 4. Average hit ratio, average system response time, and standard deviation of response times with different FTL schemes and workloads.

effect on the SSD's performance. The details of the trace is described in Section 4.B.5, and experimental results on the trace are reported and analyzed.

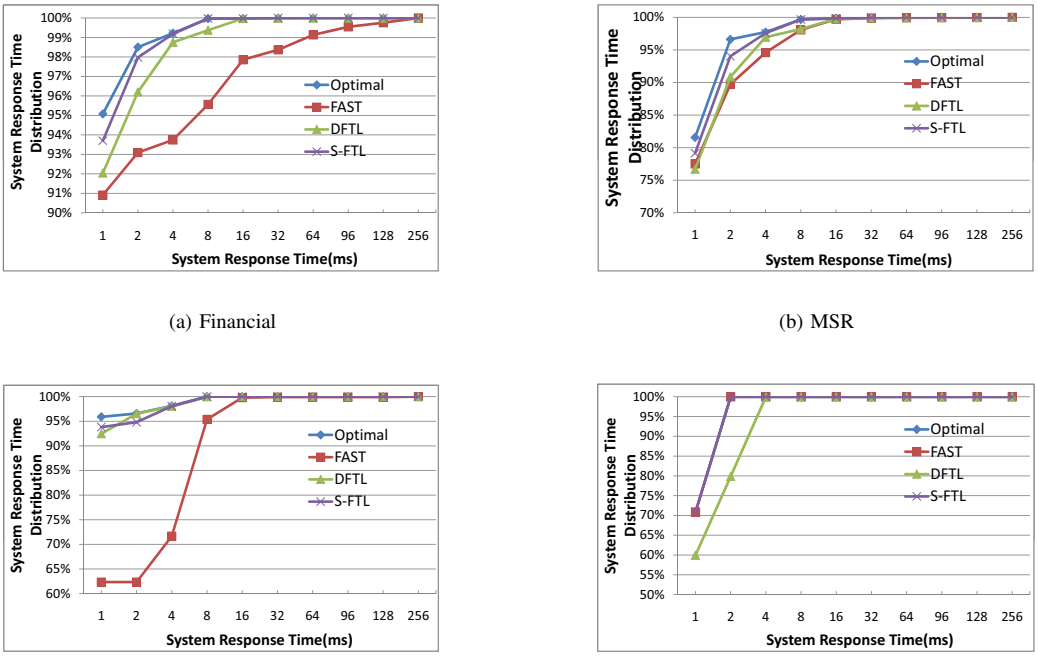
In the evaluation we use average request hit ratio, average system response time, and standard deviation of the system response times as performance metrics. The request hit ratio refers to the ratio of requests whose address translation can be completed in the cache without loading missing entries of the mapping table. Thus, the ratio has a direct impact on users' observed performance. System response time of a request refers to the time period between its arrival at the device and the completion of its requested operation (read or write). This time may include the time for address translation and garbage collection. Here we neglect the queuing delay for the requests to be pending in the host system as it is also affected by the host system and not directly concerned with FTL. Standard deviation of system response time measures variability of system response time, showing how much variation it has from the average system response time. The smaller the standard deviation is, the higher consistency in an SSD's service quality.

B. Performance Analysis

1) Hit Ratio and Response Time: Figure 4 shows average hit ratio, average response time, and standard deviation of response times with the FTL schemes for the four real-world workloads. We see that the hit ratios of S-FTL are close to that of the optimal FTL for all the four workloads, though the cache size for S-FTL is only 64KB while the optimal FTL has a sufficiently large cache (32MB) to hold the entire mapping table. This is because S-FTL can make an efficient use of the smaller cache by storing a table essentially much larger than the actually cache size. Among the workloads, *Financial* has the least sequentiality (or weakest spatial locality) among its requests (sequential reads and writes account for only 0.7% and 0.4% of all requests, as shown in Table I). Accordingly, S-FTL cannot fully take advantage of its compressed table and achieve a hit ratio as high as those with other workloads. In the meantime, it has considerable sequentiality within individual requests. Note that average size of requests in the *Financial* trace is 9.85KB, or there are almost five pages in a request by average. This allows S-FTL to show its effectiveness. As a result, a 90.7% hit ratio is achieved. In comparison, the hit ratios of DFTL is much worse with the small cache size, as the access locality exhibited in the workloads cannot be effectively captured in the cached portion of the mapping table. By compressing the table, S-FTL essentially caches a much larger portion of the table and makes locality effectively exploited. In the experiment we set the cache as large as the mapping table for the block-level mapping, which is adopted for data pages in FAST. Therefore, the hit ratios for FAST are always 100%.

Figure 4(b) shows the average response times for the four workloads. Generally, as S-FTL has higher hit ratios than DFTL, it produces lower response times than DFTL by 25% in average. However, its improvements on response times are not as dramatic as those on hit ratio, this is because garbage collection takes a major proportion in the response time and S-FTL uses a garbage collection method similar to that of DFTL. It is no surprise to see that FAST has significantly higher response times than other FTL schemes for the three workloads with substantial write requests, namely, *Financial*, *MSR*, and *Cello99*. FAST has to maintain a rigid regularity of address mapping for the block-level FTL, which greatly increases overhead of garbage collection. For the read-dominant workload *Websearch*, there is not any garbage collection cost and FAST achieves response times as low as the optimal FTL. DFTL has a higher response time because of its high miss ratio on the in-cache mapping table. In contrast, S-FTL consistently produces response times that are close to their counterparts for the optimal FTL.

Figure 4(c) shows standard deviation of response times for the workloads under the FTL schemes. The optimal FTL has the smallest standard deviation. Each of its request address translations in the FTL costs the same and garbage collection cost is evenly distributed in most cases, leading to the least variations in its request response times. The standard deviations for S-FTL are also small as most of request translations are hits. The standard deviations for DFTL are moderately



(a) Financial

(b) MSR

(c) Cello99

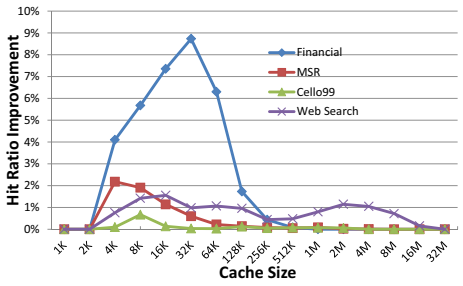
(d) Websearch

Fig. 5. Distribution of system response times represented in the CDF curves.

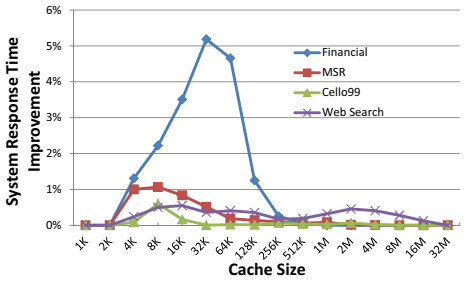
larger because a substantial number of the translations require loading translation pages from the flash and become more expensive than those hitting in the cache. FAST has much worst standard deviations for write-dominant workloads, as it can incur very expensive full merges in the garbage collection from time to time.

2) Distribution of System Response Time: To obtain more insights on the response time, we plot the cumulative distribution function (CDF) curves for response times of different workloads to show their distributions. As shown in Figure 5, the CDF curves of S-FTL are the closest to those for the optimal FTL. For the write-dominant *Financial* trace, the number of accesses to the translation pages on the flash and the number of translation block erases for S-FTL are smaller than those for DFTL by 72% and 28%, respectively. This helps S-FTL produce more response times that are smaller than those for DFTL. For *Cello99*, S-FTL does not show clear performance advantage over DFTL. S-FTL even has a slightly smaller number of requests whose system response time is larger than 4ms. That is because writes of *Cello99* are distributed over a large data space and S-FTL uses transaction pages as units for caching the mapping table. This can cause many dirty translation pages in the cache. Though S-FTL allocates a certain amount of cache space for mapping entries sparsely distributed over various translation pages to avoid write-backs of the pages, the size of the cache allocation for this purpose is small (50 entries by default). When this allocation is used up, the mechanism is disabled and many cached entries need to be flushed back to the flash. Consequently, more translation page write-backs would occur for S-FTL than for DFTL. As serious shortage of the cache allocation is only found for this workload, we did not increase the allocation to address the issue. As a future work, we plan to study how to dynamically determine the size adapting to the observed access behaviors. For the read-dominant workload *Websearch*, FAST provides a response-time curve overlapped with those for the optimal FTL as it does not have any cache misses and does not have any garbage collections. The curve for S-FTL is also almost overlapped with that for the optimal FTL, as S-FTL has a hit ratio of 92.8%. With a hit ratio of only 31.5% (see Figure 4(a)), DFTL incurs a much larger address translation overhead and produces a smaller percentage of requests of small response times. For workloads of dominant write requests, FAST demonstrates much worse performance (see the curves for *Financial*, *MSR*, and *Cello99* in Figure 5) as the writes cause expensive merges required by the block-level mapping FTL.

3) Impact of Cache Size on S-FTL's performance: As we have shown the performance of S-FTL with a cache of the same size as required by the block-level FTL, we would like to see the impact of the cache size on the workloads' performance. Figure 6 shows the hit ratios and response times for each of the four read-world workloads with cache sizes varying in a large range. As S-FTL uses an LRU list to record

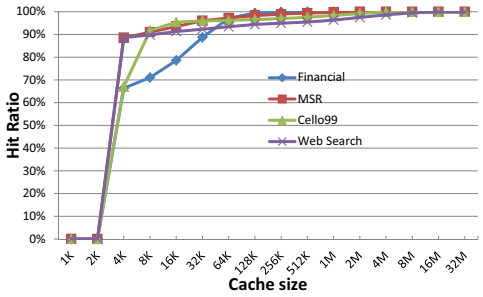


(a) Hit Ratios

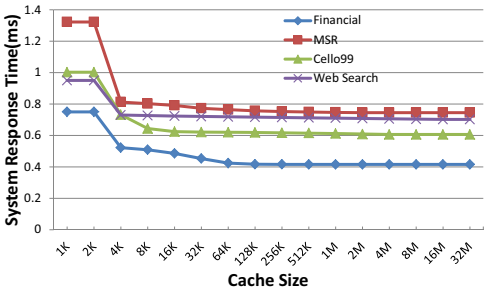


(b) System Response Times

Fig. 7. Impact of S-FTL's cache page shrinking technique on the workloads' performance in terms of hit ratio and response times at different cache sizes.



(a) Hit Ratio



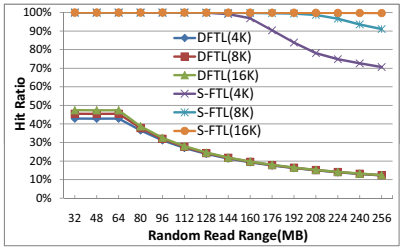
(b) System Response Time

Fig. 6. Impact of cache size on S-FTL, showing average hit ratios and response times for four real-world workloads with increasingly large cache for storing the mapping table.

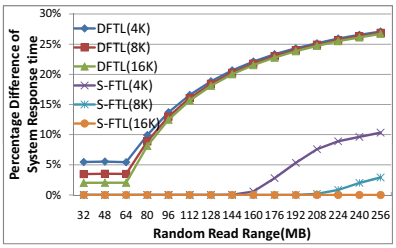
history of translation-page access, a larger cache would help hold a larger portion of a workload's working set in the cache until the working set has been entirely cached. As shown in Figure 6(a), when the cache size is very small (less than 2K), only one translation page can be cached and the hit ratio is almost 0%. When the cache size is increased to 8KB, the ratio grows rapidly to around 90% except for workload *Financial*. When the cache size increases beyond 64KB, further increase of the size produces diminishing return on either hit ratio or response time (see Figure 6(b)). This demonstrates that S-FTL can easily hold a compressed mapping table with a very small cache. The relatively lower hit ratio for the *Financial* workload is due to its large working set. The advantage of S-FTL of being able to achieve high hit ratios with a very small cache is especially desired on the economically-configured systems, including consumer electronics such as cellular phones and digital cameras.

4) Impact of S-FTL's Page Shrinking Technique: By using two bits for recording sequentiality of adjacent pages, the S-FTL's page shrinking technique can recognize more and longer sequences, or record fewer head entries for sequences. In other words, with a given cache size, the technique is expected to hold a larger portion of a mapping table in the cache so as to improve hit ratio. In other experiments, we use the default version of S-FTL, which employs only one bit. In this experiment we investigate how the technique can make a difference. Figure 7 shows the improvements made in terms of hit ratio and response time by applying the page shrinking technique over the default S-FTL version. As shown in the figure, the *Financial* workload receives the most significant improvements (up to 9.0% on hit ratio and up to 5.1% on response time). As *Financial* has many random writes, which can easily break long sequences, leading to relatively low hit ratios, when only one bit is used to link pages in a sequence (see Figure 6(a)). By using two bits, the existing sequences are less vulnerable to the writes and S-FTL can keep maintaining a large portion of the table in the cache. For the other three workloads, one-bit S-FTL is sufficient to provide high hit ratios with a small cache. Using two bits results in very limited improvements.

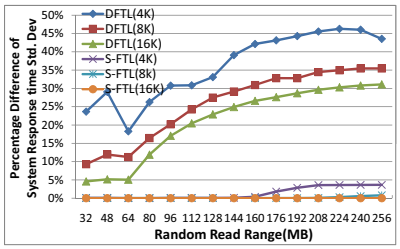
5) Impact of S-FTL's Prefetching Effect: The access unit of the flash is page, rather than entries of the mapping table. Accordingly, S-FTL uses translation page as caching unit, and all mapping entries contained in a page are cached after the page is loaded. For those entries that are not required for the current translation, they are essentially prefetched. To investigate how the prefetching effect would interact with



(a) Hit Ratio



(b) System Response Time



(c) System Response Time Std. Dev

Fig. 8. Hit ratios and system response times with different read ranges and upper bounds of request size.

different access patterns and impact workloads' performance, we designed a program that generates synthetic traces of different access patterns. In the traces, one read request is issued in every millisecond. The program generates a trace by accessing each page of the 32GB SSD for once. The SSD is divided into a number of ranges of constant size and the program accesses the ranges one by one sequentially. However, data are randomly requested within each range, with requests uniformly distributed between 2KB and N KB, where N is the upper bound of request size and configured as 4, 8, or 16 (or 2, 4, or 8 pages, respectively) in different traces.

Figure 8 shows the average hit ratio, the average degradation of response times over those of the optimal FTL in percentage, and standard deviation of the percentages for traces with different range sizes and upper bounds of request sizes in the SSD under S-FTL and DFTL. We can see from the figure that S-FTL consistently achieves high hit ratios across different range sizes from 32MB to 256MB and different request size distributions. When the upper bound of request sizes is 16KB (8 pages), S-FTL's hit ratio stays at almost 100%. In addition, as shown in Figure 8(b) S-FTL produces response times as low as those of the optimal FTL (approximately 0% degradation) in this scenario. This is because large request sizes make the prefetched mapping entries well utilized. As we know, one translation page contains 512 mapping entry, providing a translation coverage of a 1MB data. When the request sizes are small and random access ranges are large (more than 160MB), the prefetched mapping entries may not be used before the corresponding translation pages are evicted out of the cache. That is why we see reduced hit ratios for smaller requests with the increase of range sizes in Figure 8(a). Compared to S-FTL, DFTL caches individual mapping entries and does not fully exploit the prefetching opportunity. Therefore, its hit ratios are significantly lower than those for S-FTL.

In terms of response times, DFTL has dramatic degradation over the optimal FTL with the increase of random access ranges (see Figure 8(b)). The degradation can be as high as 27%. When the random read range is only about 64MB, its hit ratio and response time start to deteriorate. In contrast, S-FTL is sensitive to the range size only when the upper bound of request size is less than 16KB. In such a case, it is not affected by the increasing randomness until the range size increases to around 160MB. From Figure 8(c) we can see that variations of S-FTL performance are much smaller than those of DFTL, echoing the trends on the improvements of response time shown in Figure 8(b).

V. Conclusions

This paper proposes a new FTL mapping scheme (S-FTL) that can exploit readily available spatial locality in the SSD's workloads and significantly reduce page mapping table size without imposing any restriction on page mapping method. By doing so, S-FTL can benefit from the page-level mapping with a low garbage collection cost. It effectively addresses (1) the challenge of high garbage collection cost experienced by the block-level mapping and hybrid mapping such as BAST and FAST; and (2) the challenge of limited cache size experienced by the page-level mapping such as DFTL. The extensive

evaluation with real-world and synthetic traces shows that S-FTL can significantly improve hit ratio with limited cache size, reduce garbage collection overhead, and provide consistently improved response times across workloads of a variety of access patterns.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments that helped us to improve the paper. This research is partially supported by U.S. NSF CAREER award CCF-0845711 and partially supported by China National High-Tech Research and Development Plan under Grant No. 2009AA011906.

REFERENCES

[1] Sun Storage 7000 Unified Storage Systems. "http://www.sun.com/storage/disk_systems/unified_storage/"

[2] MTRON, Solid State Drive MSD-SATA3035 Product Specification, "http://mtron.net/Upload_Data/Spec/ASiC/ MOBI/SATA/MSD-SATA3035_rev0.4.pdf", 2008.

[3] Micron, "Small-Block vs. Large- Block NAND Flash Devices. Technical Report (TN-29-07), "http: //www.micron.com/products/nand/technotes", 2007.

[4] UMass, "UMass Trace from UMass Trace Repository", "http://traces.cs.umass.edu/index.php/Storage/Storage", 2002.

[5] HP Labs, "Tools and Traces", "http://www.hpl.hp.com/research/ssp/software/", 1999.

[6] Microsoft Research, "MSR Cambridge Traces", "http://iotta.snia.org/traces/list/Subtrace?parent=MSR+Cambridge+Traces", 2007.

[7] The DiskSim Simulation Environment(v4.0), Parallel Data Lab, *URL: http://www.pdl.cmu.edu/DiskSim/*

[8] F. Chen, D. Koufaty, and X. Zhang. "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives". In Proc. of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09), 2009.

[9] T. Chung, D. Park, S. Park, D. Lee, S. Lee, and H. Song. "System Software for Flash Memory: A Survey". In Proc. of International Conference on Embedded and Ubiquitous Computing (ICEUC'06), 2006.

[10] P. Estakhri and B. Iman. "Moving sequential sectors within a block of information in a flash memory mass storage architecture", United States Patent, No. 5,930,815, 1999.

[11] A. Gupta, Y. Kim, and B. Urgaonkar. "DFTL: a Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In Proc. of the 14th international Conference on Architectural Support For Programming Languages and Operating Systems (ASPLOS'09), 2009.

[12] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. "DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities". In Proc. of USENIX Confernece on File and Storage Technoologies (FAST'05), 2005.

[13] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance", in Pro. of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'02), 2002.

[14] H. Kim and S. Ahn, "BPLRU: Buffer Management Scheme for Improving Random Writes in Flash Storage". In Proc. of 6th USENIX Conference on File and Storage Technologies (FAST'08), 2008.

[15] J. Kang, H. Jo, J. Kim, and J. Lee. "A Superblock-based Flash Translation Layer for NAND Flash Memory". in Proc. of 6th ACM and IEEE International conference on Embedded software, 2006.

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

[17] S. Lim, S. Lee, and B. Moon. "FASTer FTL for Enterprise-Class Flash Memory SSDs". in Proc. of 6th IEEE International Workshop on Storage Network Architecture and Parallel IOs (SNAPI), 2010.

[18] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. "A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation". in ACM Trans. in Embedded Compututing Systems, Vol. 6, Issue 3, 2007.

[19] S. Lee, D. Shin, Y. Kim, and J. Kim. "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems". in Proc. of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability, 2008.

[20] T. Shinohara. "Flash memory card with block memory address arrangement". United States Patent, no. 5,905,993, 1999.