A Hash-based Space-efficient Page-level FTL for Large-capacity SSDs

Fan Ni[†], Chunyi Liu[†‡], Yang Wang[⊤], Chengzhong Xu[⊤], Xiao Zhang[‡], Song Jiang[⊤†]
[†]University of Texas at Arlington, Texas, US
[‡]Northwestern Polytechnical University, Xi'an, China
[⊤]Shenzhen Institutes of Advanced Technology Chinese Academy of Sciences, China
fan.ni@mavs.uta.edu, chun.liu@uta.edu, yang.wang1@siat.ac.cn, cz.xu@siat.ac.cn, zhangxiao@nwpu.edu.cn, song.jiang@uta.edu

***Abstract*—With increasing demands on high-performance and large-capacity SSDs in the enterprise-scale storage, the concern about inefficient use of the DRAM space in SSDs rises, especially for those using page-level FTL (Flash Translation Layer). In such an FTL, the address mapping scheme allows a logical page address (LPA) to be mapped to any physical page address (PPA) on the disk. Though it provides flexible address management and minimizes internal data movements, it requires a large address mapping table whose size is proportional to the capacity of the disk. With the increase of SSD's capacity, the table can be too large to be held entirely in the DRAM buffer of the SSD, causing constantly accessing to the flash for the address translation. This performance penalty due to the buffer misses is particularly high with workloads of weak access locality and large working sets. In this paper, we propose a space-efficient page-level FTL using hash functions in the address translation, named Hash-based Page-level FTL, or HP-FTL in short, to address the concern. HP-FTL trades mapping flexibility with limited performance impact for high space efficiency allowing the entire table to fit in the buffer and eliminating translation misses. The experiment results show that HP-FTL can provide up to 2.6X throughput compared to DFTL, a representative page-level FTL, using the same amount of DRAM for buffering the table. Meanwhile, HP-FTL reduces the mapping table size to about 25% of the table space required by page-level mapping schemes, including DFTL, without having any buffer misses.**

I. Introduction

The Flash-based SSD is a high-performance storage device with an access speed much faster than the hard disk. It has become the mainstream storage device in both consumer and enterprise markets for its high throughput, low latency, and power efficiency. Increasingly more applications rely on high-end SSDs to provide performance-critical services. These applications include databases [1], [2], key-value stores [3], [4], [5], and OLTP applications [6], [7]. With the advances in semiconductor and 3D vertical NAND flash technology, the cost-per-bit of the flash memory keeps falling. SSD vendors currently provide SSD products of increasingly large capacity at acceptable prices.

Flash memory has unique characteristics which distinguish it from other storage devices, including hard disks. In particular, its erase-before-write feature precludes in-place update, or overwrite. That is, a page cannot be overwritten for updating until it is erased. In order to mask the peculiarity from the software, a layer of firmware, or flash translation layer (FTL), has been introduced within flash-based storage devices. When a write request to logical address(es) is served, the data will be directed to physical page(s) that have been erased on the flash. Meanwhile, the FTL sets up mapping(s) from the logical address(es) of the data to the physical address(es) on the flash where the data are stored. When a read request is received, the FTL translates the logical address(es) to their corresponding physical address(es) using the recorded address mappings. In addition to maintaining the logical-to-physical (L2P) address mappings, FTL must also conduct garbage collection (GC) operations for space reclamation. As the GC process can only be performed at the unit of erase block, which can consist of a large number of pages, the valid data pages in a victim block selected for recycling need to be copied to other blocks in the flash and correspondingly the address mappings are updated. The mapping flexibility, which describes severity of restriction imposed on selection of physical address corresponding to a given logical address, may significantly affects GC's performance. A less flexible mapping scheme, such as block-level FTL, requires GC more frequently migrate valid pages out of blocks to be erased.

In general, there are three FTL address mapping schemes. In addition to block-level FTL, there are also page-level and hybrid FTLs. These mapping schemes provide different tradeoffs between mapping flexibility and space overheads. With the page-level FTL [8], a logical page can be potentially mapped to any physical page, similar to the fully associative cache. In this mapping, each logical page requires an entry in the mapping table to specify its mapped physical page. While this FTL provides high mapping flexibility, it introduces a high space overhead. In contrast, a block-level FTL requires that a number of the least significant bits of address of a logical page and those of address of its mapped physical page be the same [9], [10], a mapping scheme similar to that in the direct-mapped cache. The block-level FTL reduces mapping flexibility to achieve high space efficiency as all pages in a block share a single mapping entry in the mapping table. In hybrid address mapping [11], both page-level and block-level address mappings are used to make a compromise between mapping flexibility and space-efficiency.

High-end SSDs favors page-level address mapping scheme over block-level and even hybrid mapping schemes to minimize the GC cost. This is due to the fact that it is the least restrictive on finding qualified erased pages for data writes,

while other mapping schemes may introduce significant write amplification to accommodate address mappings for multiple pages into one mapping table entry. However, the page-level mapping scheme requires a large L2P mapping table. Suppose each entry for mapping a 4KB page is 4-byte long, the size of the L2P table will be 0.1% of the SSD's capacity. That is, for a 4TB SSD, the table will be of 4GB.

Since the mapping table is accessed by every read/write request, for high performance it is desirable that the entire table resides in the DRAM of the SSD. However, keeping the whole mapping table into the DRAM is often not possible for a large-capacity SSD due to the limited size of DRAM. The size of the DRAM in an SSD is usually bounded by the 32-bit embedded processor, which can only access 4GB address space. While embedding a 64 processor may enable use of a larger DRAM, it will increase the cost, power consumption, complexity of the DRAM controller as well as the DRAM access latency. In this way, the large L2P mapping table has become a serious bottleneck for large-capacity SSDs and would compromise the devices' I/O performance if a large number of mapping entries were left un-cached in the DRAM.

To this end, FTL schemes that on-demand load mapping entries, such as DFTL [8], have been proposed to only cache recently used mapping entries in a small in-DRAM table. Meanwhile, the entire page-level L2P table is stored in reserved blocks on the flash memory dedicated for address translation. Once there is a miss in the in-DRAM table, the L2P table on the flash is consulted and the missed mapping entry as well as a number of contiguous entries next to it are loaded into the in-DRAM table. Meanwhile, some victim entries currently in the in-DRAM table are selected for replacement. They can be discarded or written back to the translation blocks on the flash depending on whether they have been modified. The effectiveness of the on-demand table loading schemes relies on strength of I/O access locality of workloads. With a strong locality, the scheme can provide performance similar to that of the page-level FTL keeping all mapping entries in the DRAM. However, for workloads with weak locality, or with many random accesses in a large table footprint, it will incur a large number of mapping misses and significant miss penalty. Unfortunately, many real-world applications, including databases and big data processing, show widespread random access patterns in large working sets [12], [13]. For these workloads, with DFTL and similar FTLs relying on L2P table caching, the SSD's performance can be severely degraded.

In this paper, we propose a Hash-based Page-level FTL, or HP-FTL, for high-performance large-capacity SSDs. In the FTL two page-level L2P mapping tables, a compact primary mapping table and a small supporting secondary table, are maintained. The L2P mappings in the primary table are encoded compactly with a series of hash functions for low space overhead and configurable mapping flexibility. The secondary table is used as a supplement to the primary mapping table for resolving hash mapping collisions and reducing GC cost. As most mapping information can be held in the primary table with a compact encoding scheme and the secondary table is small, the total size of the two mapping tables can be sufficiently small so that they can easily fit in the SSD's DRAM. In this way, HP-FTL supports high access performance with moderate DRAM space overhead.

The contributions of the paper are twofold:

- we propose a novel FTL design, HP-FTL, in which a high-performance page-level address mapping scheme with high mapping flexibility and space efficiency are provided. Different from proposals based on on-demand table mapping schemes, the efficiency of the address mapping scheme in HP-FTL does not require strong access locality in the workloads, which makes it efficient even for workloads weak locality.
- We evaluate HP-FTL extensively by experimentally comparing it with other FTL schemes. The results show that compared to on-demand table mapping schemes, HP-FTL can provide up to 2.6X throughput. Meanwhile, it reduces use of DRAM space to about 25% of that of a conventional page-level mapping scheme.

The rest of the paper is organized as follows. The design of HP-FTL is described in Section II. The experiments and analysis of the results are given in Section III. In Section IV, we present prior studies on similar efforts. We conclude the paper in Section V.

II. The Design

The major design goal of HP-FTL is to provide high-efficient address translation with a relative small address mapping table. Considering the significant overheads of serving write requests and garbage collection introduced by the block-level and hybrid mapping schemes, we choose a pure page-level mapping scheme for address translation. To achieve the goal with a page-level mapping scheme, a big challenge is how to limit the size of the mapping table while retaining high-efficient address translation. To this end, we devise a compact address mapping encoding scheme, which significantly reduces size of individual mapping entries. However, it is infeasible to maintain a smaller mapping table without compromising performance and mapping flexibility. In the design of HP-FTL, we provide a solution to make a careful trade-off between space overhead and mapping flexibility, which minimizes the performance penalty.

A. HP-FTL Architecture

Unlike on-demand table loading scheme, such as DFTL, the high efficiency of HP-FTL does not rely on the locality of the accesses issued to the SSD. Instead, a novel compact encoding approach is developed to encode each address mapping entry with much fewer bits. In this way, the total space used by mapping tables of HP-FTL can be much smaller and easily be fit in the DRAM of the SSD. However, the table reduction does not come for free. Although HP-FTL provides much higher mapping flexibility than block-level mapping schemes, the number of candidate positions a logical page can be mapped is limited and thus there is still a chance that a

logical page will not find a valid physical location under the compact mapping scheme. As a supplement, a small secondary table is maintained to admit the address mappings that cannot be recorded in the primary mapping table. Periodically, the mapping information in the secondary table will be merged into the primary mapping table for space-efficiency. We call this two-table-supported logical to physical address translation as *Dual-Table based L2P address translation* scheme. Besides the two L2P mapping tables, there are data structures, such as that recording the status of each physical page on the SSD, maintained in the DRAM. A physical page can be in any of the following three states: clean, valid, and invalid. A clean page can admit new data with a write operation and then turns into a valid page. When a valid page is updated out-of-place or discarded, it becomes invalid and cannot be written again until it is erased and turns into a clean page.

B. Dual-Table based L2P Address Translation

To serve a read/write request in an SSD, the L2P mapping table must firstly be accessed. To retrieve a page from the flash, the table is looked up to get the corresponding physical page number (PPN) mapped to the logical page number (LPN) of the request, and the flash page at the PPN is read and returned. To serve a write request, a clean page is found to admit the data and the physical address of the page is used to update the L2P table by either inserting a new entry to the table (for new write) or updating an existing entry in the table (for overwrite).

In the HP-FTL scheme, two tables, a primary table and a secondary table, are used to maintain the address mapping information. In the design, the mappings in the primary table cover the entire logical address space. Assuming the logical address space covers *N* pages, from page *0* to page *N-1*, the primary table will include *N* entries with each entry storing an encoded PPN indicating the location where the LPN is mapped as shown in Fig. 1. In a conventional page-level scheme, to cover 2^{32} pages each PPN in the table has to be 4-byte (32bit) long. Instead, HP-FTL uses much fewer bits to represent a PPN in the primary table. A set of hash functions are introduced to generate the PPN's new representation of reduced size, which we call encoded PPN. The encoded PPN is composed of two fields, HID (Hash ID) and PPID (Partial Page ID). The HID field records which hash function is used to select a physical block where a page is used to stored the data. If there are totally 2^h candidate hash functions, the HID field will occupy h bits. For example, a 4-bit HID field can specify one out of at most 16 hash functions, from F_0 to F_{15} as shown in Fig. 1. The PPID field is used to generate the offset of the page inside the block determined by the HID field. As p bits are needed to index a page in a block consisting of 2^p pages, the length of the PPID field, marked as m, should not exceed p. If m is smaller than p, some bits from LPN are borrowed to synthesize the p-bit page offset. For example, in an SSD where a block includes 32 pages, 5 bits are needed for indexing a page inside a block, that is, $p = 5$. There are two possible cases for the m value.

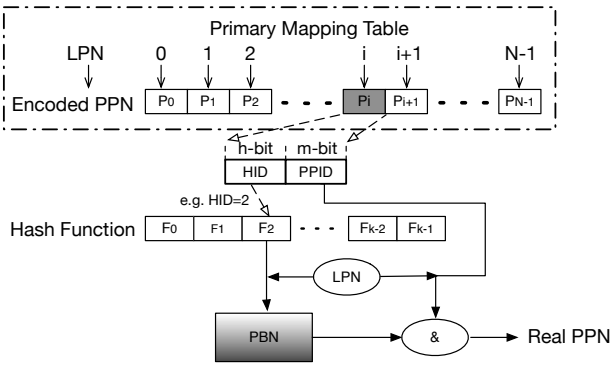


Fig. 1: *The primary mapping table tracking address mapping*

- $m = 5$: the page offset is indexed directly by the PPID field, and the logical page can be mapped to any physical page in a block.
- $m < 5$: concatenating the m bits with the least significant $p - m$ bits of LPN, we generate the p-bit page offset. In particular, when m is 0, the page offset is directly determined by the least significant p bits of the LPN, similar to that of the block-level mapping scheme.

Suppose h and m bits are used for HID and PPID fields, respectively, in which case $h + m$ bits are used to encode a PPN, there will be 2^{h+m} candidate locations for storing a logical page. For example, when $h = 3$ and $m = 5$, a logical page can potentially be mapped to one of 2^8, or 256, pages. By varying the length of *h* and *m*, different trade-offs between table size and mapping flexibility are achieved.

Although the primary mapping table provides a large number of candidate locations for a logical page, there is still a chance all these locations have been occupied by other logical pages, which we name as a mapping collision, or the locations have been invalidated and need to be erased before serving new writes. To service the write requests for which no valid address mapping slots can be found in the primary mapping table, another table, named secondary mapping table, is maintained in the DRAM. For each mapping in the table, the logical page can be mapped to any valid physical page in the disk as that in a fully associative cache. For the secondary table, each entry uses 8-byte to record the LPN and PPN of an address mapping, which otherwise may need only 1-byte if it were held in the primary table. The secondary table is expected to be very small as the mapping collision in the primary table is very low with a large number of (8 or more) hash functions evenly covering the physical address space and thus most L2P mappings are held in the primary table. In a very rare case when the secondary table is full, a garbage collection process is triggered as described in Section II-D to reorganize data layout so that entries in the secondary table can be accommodated in the primary table. An on-flash mapping table is maintained mainly for fast mapping table setup when the device is mounted normally.

We have tuned our dual-table design to make the L2P address translation as efficient as possible. First, as an LBN can be either unmapped, mapped in the primary table, or

mapped in the secondary table, we reserve two values of the HID field to represent the *unmapped* and *mapped-in-secondary-table* cases to avoid unnecessary accesses to the flash and the secondary table. In the description and evaluation of this paper hereafter, a value of 0 and $2^h - 1$ is used to indicate that the LPN is unmapped or mapped in the secondary mapping table, respectively, where h is the number of bits in the HID field. Second, considering the lookup into the two tables, the primary table is fast as it is indexed by the LBN directly. However, accesses to the secondary table can be inefficient if it is contiguously searched from the beginning. Keeping the table sorted can speed up the lookup but it can be too expensive. Organizing the table into a data structure with higher lookup performance, for example using hash tables or B-trees, will need extra space to store the metadata, which will compromise the space efficiency of the design. In HP-FTL design, we reuse the PPID field in the primary table to enable fast retrieval of the mapping entries in the secondary table. For a secondary table which can hold at most *n* entries, we divide it into 2^m segments, where *m* is number of bits of the PPID field. When a mapping entry for a logical page is added to the secondary table at an offset named *pos*, the segment id is calculated as $pos * (2^m)/n$ and stored in the PPID field of the entry in the primary table indexed by the LPN. The reuse of PPID will not introduce extra space overhead and will significantly speed up the lookup in the secondary table since at most $2^m - 1$ positions are checked to find a mapping entry.

C. Service of Read/Write Operations

HP-FTL provides efficient address translation for read and write requests by keeping all page-level mapping information in the DRAM with low space overhead.

For a read request, the logical page address, for example LP_1, is used to index into the primary table to get the encoded PPN, named *ePPN*. If the *ePPN*, whose HID is in the range of (0, $2^h - 1$), indicates a valid mapping entry in the primary table, the PPN is generated from LP_1 and *ePPN* as shown in Fig. 1. This PPN is then used to retrieve the data from the flash and the request returns with the data. If the HID field indicates that the mapping information is in the secondary table, when the HID field is $2^h - 1$, the PPID field of *ePPN* is extracted and acts as the start offset into the secondary table to search for the mapping entry. Otherwise, the LPN is unmapped and the read request returns immediately.

The serving of a write request needs to locate a clean (physical) page to store the data and establish/update an L2P mapping in either the primary or the secondary tables for a new write/overwrite to a logical page. This is achieved by first checking the primary table with the LPN (denoted as LPN_w) of the request and then the secondary table if necessary. The process to serve a write request is described in Algorithm 1. To successfully find a valid physical page to serve a write request and store the L2P mapping in the primary table, three conditions must be met. First, the id of the block where the physical page lies must be calculated by one of the predefined hash functions with LPN of the logical page as input. Second,the page offset inside the block must be able to be represented by the mechanism described in Fig. 1 and recorded with PPID field. Third, the write to the physical page should follow the rule of sequential programming within a block [14]. If the secondary table is used for storing the mapping information, it is much easier to find a valid page as it works like a fully associative cache.

Algorithm 1 Process to serve a write request

Require: LBN of the request (lpn)
  $ePPN \leftarrow primary\_table[lpn]$
  $hid \leftarrow ePPN[HID]$
  $ohid \leftarrow hid$
  while $hid < 2^h - 1$ **do**
    // H_{hid} *is the* hid^{th} *hash function*
    $pbn \leftarrow H_{hid}(LBN)$
    $flag \leftarrow$ found a valid clean page in block pbn?
    if $flag = found$ **then**
      // *ppid is the page id inside the block*
      Write data to flash page $ppid$ in block pbn
      $ePPN[HID] \leftarrow hid$
      $ePPN[PPID] \leftarrow ppid$
      $primary\_table[lpn] \leftarrow ePPN$
      $return$
    else
      $hid \leftarrow hid + 1$
    end if
  end while
  // *No valid mapping entry found in primary table, try secondary table...*
  Get a clean page ppn and write the data;
  if $ohid = 2^h - 1$ **then**
    pos $\leftarrow$ old offset of mapping for lbn in secondary table;
    Add lpn$\rightarrow$ ppn to secondary table at offset pos;
  else
    pos $\leftarrow$ offset of the next free slot in secondary table;
    Add lpn$\rightarrow$ ppn to secondary table at offset pos;
    $ppid \leftarrow pos * (2^m)/n$
    $ePPN[HID] \leftarrow hid$
    $ePPN[PPID] \leftarrow ppid$
    $primary\_table[lpn] \leftarrow ePPN$
  end if
  return

D. Garbage Collection

Since garbage collection in SSDs is expensive, the timing when it is triggered and the duration it lasts have a direct impact on the I/O performance of the device. In HP-FTL, two thresholds, *low-watermark* and *high-watermark*, are defined to control the start and termination of the garbage collection process, respectively. The garbage collection process is triggered whenever the number of free slots in the secondary table drops below the low-watermark and stops when the number reaches the high-watermark. The garbage collection is likely to

be triggered more frequently with a higher low-watermark and lasts longer when the difference between high-watermark and low-watermark is large. Low-watermark and high-watermark are defined regarding the the secondary mapping table instead of the primary table as the secondary table gets full usually after the primary table is almost full and no valid mapping entries can be found to serve write requests. A physical page allocation data structure is used to detect an extreme situation when the number of the clean pages in the disk drops to a low level while the number of free slots in the secondary table is higher than the low-watermark, triggering a garbage collection.

Once the garbage collection process is triggered, the following steps are performed to reclaim the physical blocks and free some mapping entries in the secondary table. First, blocks with the fewest valid pages are selected as candidates to erase. The valid pages in the candidate blocks are copied into other blocks and the mappings are recorded in the secondary table due to its mapping flexibility until the table is almost full. Second, the candidate blocks selected in the first step are erased and used to admit valid pages whose mapping information was in the secondary table and can be moved to the primary table after the relocation. This step will create some free slots in the secondary table but may not be enough to reach the high-watermark threshold. In this case the GC process repeats until the high-watermark is reached.

III. Evaluation

A. Experiment Setup

To evaluate HP-FTL's performance, we implement a virtual SSD device as a device mapper target [15] at the generic operating system block device layer in Linux kernel 4.7.1. The read and write requests issued by the applications are directed to the virtual device, where the FTL functionalities, such as address translation and garbage collection, are implemented, and the requests are then forwarded to a back-end SSD disk. The requests sent to the SSD access a limited range of the logical address space exposed by the SSD. We further 'trim' (using *fstrim* command) all blocks in the SSD before each experiment to minimize chance of triggering the garbage collection operations. In this way, we essentially treat the logical addresses provided by the real SSD as physical addresses in the FTLs we implement. In our experiments, we use direct I/O mode to minimize caching effect on the I/O operations.

In the evaluation, we also implement two other page-level FTL schemes. Among them, PFTL represents a page-level mapping scheme where the mapping table is entirely stored in the DRAM. It is an ideal FTL as it provides full address mapping flexibility without any performance penalty caused by its large mapping table. In contrast, DFTL is used to represent the on-demand page-level mapping scheme. Each time when there is a miss in the CMT (cached mapping table), the SSD is accessed to load the missed mapping entries and the victim entries are evicted to the SSD if they are dirty.

All our experiments were conducted on a Dell R630 server with two Xeon E5-2680v3 2.50GHz CPUs, each has twelve cores and 30MB last-level cache. The server equips with 128GB DDR4 memory and a Samsung 840 EVO 1TB SSD.

As the mapping table size in DRAM has a significant impact on the performance of address translation in DFTL and HP-FTL, we vary their mapping table sizes in the experiments and compare their performance with PFTL. PFTL's mapping table size is 0.1% of the disk's capacity, denoted as Base-Table-Size or *BTS*, which allows all mapping entries to fit in the DRAM.

In our experiments, the PFTL's table size is used as reference to specify the amount of DRAM space allocated for holding DFTL's or HP-FTL's table entries. For such a space whose size is less than the BTS, some mapping entries of DFTL have to stay out of the buffer. For HP-FTL, the size of the primary table is fixed once the lengths of the HID (h) and PPID field (m) in each mapping entry are given. For example, if 8-bit is used to record *HID* and *PPID* as shown in Fig. 1, 25% of BTS will be used for the primary table as the size of each mapping entry has been reduced to 25% of that in PFTL. Additionally a small amount of space is allocated for the secondary table.

For the benchmarks, we generate synthetic page write traces with random access pattern and the Zipfian distribution to represent workloads with weak-locality and strong-locality, respectively, and issue them continuously to the virtual SSD device with different FTL mapping schemes. The LPN footprints of the weak-locality workloads cover the capacity of the virtual disk. For the ZipFian distribution, the skewness parameter is set to 0.99. Detailed configuration parameters are given in Table I.

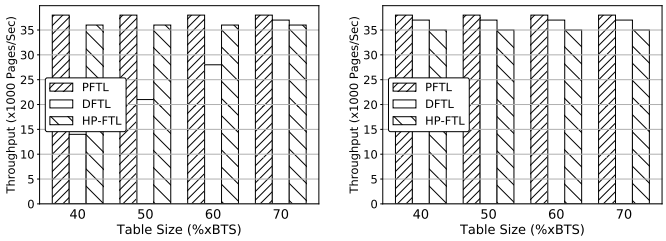
B. Experiment Results

We first compare the throughput and latency of HP-FTL to DFTL and PFTL and then evaluate how the performance of HP-FTL is affected by the values of h and m.

a) Throughput and Latency comparison: We vary the size of the mapping table(s) cached in the DRAM in DFTL and HP-FTL and measure the throughput. As shown in Figure 2a, HP-FTL shows much higher throughput than DFTL when the table size is small. For example, the improvement can be as high as 2.6X when the size of the mapping table(s) in DRAM is 40% of BTS. The significant performance difference between DFTL and HP-FTL is because DFTL can only cache 40% of the mappings in the DRAM, causing about 35% of the address translations to be missed in the DRAM, while HP-FTL stores all mappings in DRAM without any misses. When the table size increases, more mapping entries can be cached in the DRAM for DFTL, which helps improve its throughput, while for HP-FTL the throughput almost keeps unchanged, and thus the improvement decreases. When the table size increases to 70% of BTS, DFTL's throughput is a little bit higher than that of HP-FTL ($< 2\%$) as almost all mapping entries associated with the accesses are cached in the DRAM for DFTL, while HP-FTL's performance is affected by the cost of performing hashing operations. The figure reveals two performance trends. First, HP-FTL can provide a good performance even when the table size is small, which implies

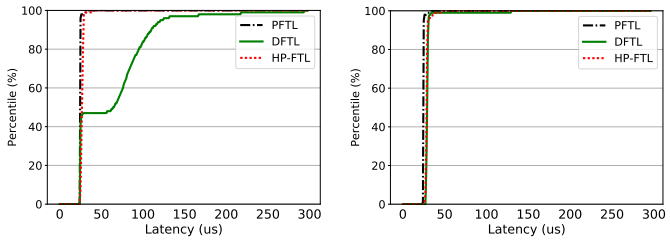
TABLE I: Experiment Configurations

Parameter	Value	Comments
Disk Size	256GB	Virtual SSD disk size
Page Size	4KB	Basic Read/Write Unit
Block Size	128KB	Basic size for Erase
BTS	256MB	Mapping table size of PFTL
h	3 unless stated	Number of bits to record HID
m	5 unless stated	Number of bits to record PPID
Hashing Function	MD5	Shifting MD5(LPN) 1 bit each time and keeping the least significant 21 bits to generate a hash function



(a) *Workloads of **weak** locality* (b) *Workloads of **strong** locality*

Fig. 2: *The throughput of PFTL, DFTL, and HP-FTL with different table sizes for **weak-locality** and **strong-locality** workloads. The size of the DRAM space used for caching address mapping in DFTL and HP-FTL is normalized against BTS. The value of h and m is 3 and 5, respectively.*



(a) *Workloads of weak locality* (b) *Workloads of strong-locality*

Fig. 3: *The CDF curves of PFTL, DFTL, and HP-FTL's latencies with different in-DRAM table sizes for weak- and strong-locality workloads. The size of the DRAM space used for caching address mapping in DFTL and HP-FTL is normalized against PFTL's table size. The value of h and m is 3 and 5, respectively.*

that its efficiency does not rely on a large secondary table and most write requests are served by the primary mapping table, which is highly space-efficient. Second, the performance of on-demand mapping schemes, such as DFTL, is significantly degraded for workloads with many random writes when the table size is small. Suppose a 4TB SSD is used, PFTL will need 4GB DRAM space to store the mapping table, while HP-FTL can provide over 94% of its throughput using only 1.6GB (40%*4GB) DRAM for its mapping tables. However, if the same amount of DRAM is used in DFTL, it can produce less than 40% of PFTL's throughput.

Figure 2b shows the throughput of the three mapping schemes for workloads with strong spatial locality. In all cases, DFTL achieves a higher throughput than HP-FTL, which is not surprising considering HP-FTL's extra cost of hashing. However, HP-FTL provides similar performance, about 95% of DFTL in all cases, which is about 92% of PFTL. Comparing Figs. 2a and 2b, we found that HP-FTL's performance decreases a little bit for workloads with strong locality accesses. This is because a large number of overwrites are involved in the workload and the mappings are recorded in the secondary table as all candidate slots are tried but no valid slot is found in the primary table due to a lazy garbage collection mechanism used in our current design. Overall, the results show that HP-FTL can constantly provide high performance whether the workload is of weak or strong access locality.

Besides throughput, we also measure the access latencies of the three mapping schemes for workloads with weak and strong locality. The results are shown in Figure 3 with the CDF curves. The latency of HP-FTL is similar to that of PFTL for both workloads. However, the latency of DFTL is significantly degraded for workloads with weak locality. For example, the 80th percentile write latency of DFTL is about 100μs while for HP-FTL it is only about 25μs.

b) Sensitive Study: In HP-FTL, two tables are used for storing mapping information with different trade-offs between space efficiency and mapping flexibility. The primary mapping table achieves high space efficiency at the expense of compromised mapping flexibility, while the secondary table provides highest mapping flexibility with compromised space efficiency. As the goal of HP-FTL is to provide high performance with low space overhead in DRAM, we need to make the secondary table small without compromising the performance. The *h* and *m* parameters determine number of candidate physical blocks a logical mapping can be mapped to, which will affect the percentage of accesses processed in the primary table and in secondary table. The more address mappings the primary table holds, the smaller the secondary table can be. We measure the space occupied by the address mappings in the secondary table and the corresponding throughput with different *h* and *m* values for workloads with weak or strong localities to study how they affect the performance and the space-efficiency of HP-FTL. The secondary table is allocated large enough to avoid overflowing.

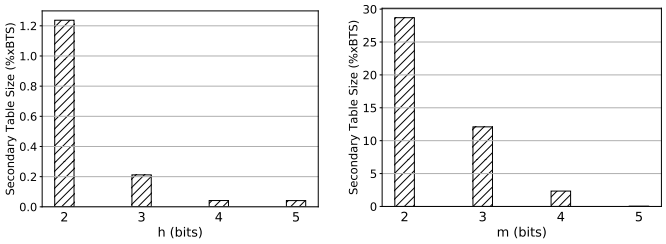
Figure 4 shows the secondary table size when different *h* and *m* values are selected and the accesses to the disk have weak locality. As shown in Fig. 4a, when *h* increases, the secondary table size decreases as a logical page can be mapped to more blocks reducing mapping conflicts in the primary table. However, even when *h* is small, such as 2, the secondary

table size is still small (about 3MB, or $1.2\% * 256$ MB). With a h value of 2, a logical page has 64 ($(2^2 - 2) * 2^5$) candidate physical locations. The results show that with a small number of hash functions, our primary table can provide sufficient mapping flexibility and avoid mapping conflicts to keep most L2P mappings in the DRAM with high space efficiency. Figure 4b shows the secondary table size when the m value varies. When m increases, the secondary table size decreases significantly. When m equals 5, the secondary table size is reduced to 512KB (0.2%*256MB). However, when m is 2, the table size is about 72MB (28%*258MB). This is because when m is small, some bits from the LPN are used to generate the page offset inside a block and the logical page can only be mapped to a subset of physical pages in a block, such as even pages. However, Writing to these pages may violate the sequential program feature inside a block and cause the logical pages to be remapped with the secondary table. The results in Fig. 4 reveal a small h and big m are helpful to reduce the size of the secondary table.

Comparing the results in Fig. 5 for strong-locality workloads, we have two observations. First, the secondary table size is not decreased when h grows from 3 to 5 as shown in Fig. 5a. For strong-locality workloads, a mapping entry is kept in the secondary table mainly because the associated data are frequently updated. The amount of frequently updated data is not sensitive to h. However, for weak-locality workloads a mapping entry has to be placed into the secondary table if it has mapping conflicts in all of its available slots in the primary table. The probability of having the conflicts depends on the h value. Second, the secondary table size is smaller with a workload of stronger locality when m is small (e.g., $m = 2$) as shown in Fig 5b. This is also because the secondary table mainly keeps address mappings associated with frequently updated data with small LBN footprints.

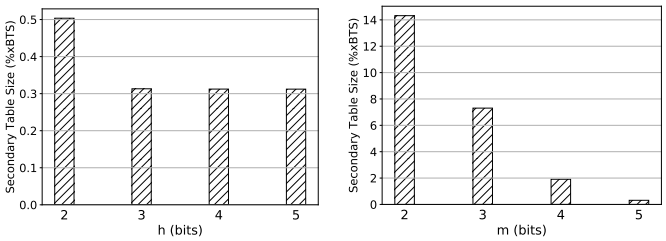
In our measurements, with h and m being 3 and 5, respectively, for the 256GB disk HP-FTL only use 64.5MB space, or 64MB for the primary table and 0.5MB for the secondary table, to cache the entire L2P mapping table while PFTL needs 256MB. Results in Figures 4 and 5 show that HP-FTL can significantly reduce the mapping table size, which is one of our design goals about space efficiency.

Figure 6 shows the throughout of HP-FTL with different h and m values. The results show that the throughput is not sensitive to h and m values. This is because no matter whether the mappings are kept in the primary table or the secondary table, the cost of address translation in DRAM is low relative to that of the flash page write. Overall, HP-FTL achieves about 93% of PFTL's performance with only 25% space overhead in the DRAM. If the saved DRAM space is used for data caching, it is likely that HP-FTL can provide even higher performance than PFTL. The experiments with strong-locality workloads show similar results as shown in Figs. 7.



(a) m=5 (b) h=3

Fig. 4: *Secondary table size when accesses with* ***weak locality*** *are issued to the disk and different h and m values are used. BTS is 256MB.*



(a) m=5 (b) h=3

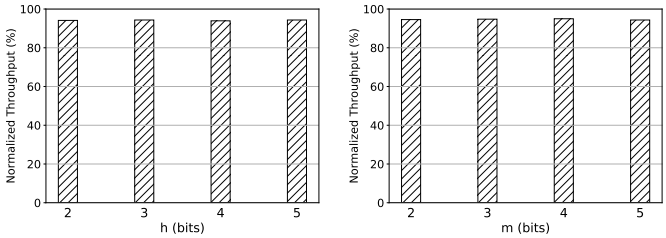
Fig. 5: *Secondary table size when accesses with* ***strong locality*** *are issued to the disk and different h and m values are used. BTS is 256MB.*

IV. Related Work

The importance of efficient address translation in FTL to provide fast accesses to SSDs has been well recognized and many efforts have been made. Studies related to the work include FTL designs for accelerating address translation and approaches for accelerating or avoiding the address translation.

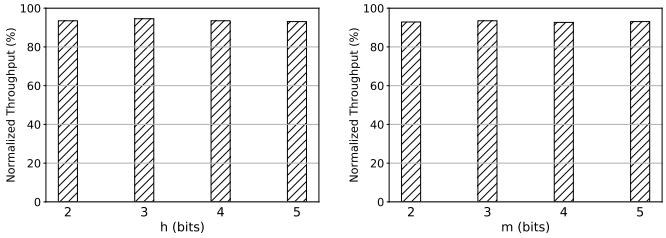
A. Efficient Hybrid FTL designs

Since block-level mapping schemes maintain a small mapping table while exhibit suboptimal performance, and page-level mapping schemes provide high performance at the expense of significant DRAM space consumption, hybrid mapping schemes, such as Adaptive FTL or AFTL [16], are proposed to make a tradeoff between these two groups of schemes. In AFTL, the flash space is divided into two areas, a small one which is maintained with a page-level mapping table and the other one managed with a block-level mapping scheme. With the hot data stored in the small area and the mapping table kept entirely in the memory, high performance is obtained. However, there are two issues in AFTL. First, pages in the two areas are swapped frequently based on their access frequency, which will introduce write amplification and compromise SSDs' endurance. Second, since hot data are stored in a small area, the AFTL scheme is highly likely to suffer from the wear-out issue. HFTL [17] adopts a similar idea as AFTL to manage hot data with a page-level mapping scheme while the cold data with a block-level mapping scheme, except that a technique based on bloom filter is used for hot data identification. HFTL shares similar issues as AFTL. Instead of detecting hot data in the device, Wu et al. proposed a file-system-aware FTL [18], in which page-level mappings are employed to manage the file system metadata,



(a) m=5 (b) h=3

Fig. 6: *Throughput of HP-FTL with different h and m values for* ***weak-locality*** *workloads. The throughput is normalized against that of PFTL, which is the optimal.*



(a) m=5 (b) h=3

Fig. 7: *Throughput of HP-FTL with different h and m configurations for* ***strong-locality*** *workloads. The throughput is normalized against that of PFTL. A value of 1 is the ideal throughput achieved with PFTL, which is the optimal.*

which are small and frequently accessed. HP-FTL does not have the aforementioned issues as it does not distinguish hot and cold data and the frequently-accessed logical pages can be mapped to any physical pages by either the primary table or the secondary table.

B. Efficient Page-Level FTL designs

An on-demand mapping scheme is proposed in DFTL [8], which provides page-level address translation. DFTL addresses the inefficiency of hybrid FTLs and reduces the space required by the mapping table in DRAM. Since DFTL only stores the recently accessed mapping entries in the DRAM and keeps the entire page-level mapping table in the flash memory, its efficiency depends heavily on the spatial locality of the workloads and its performance is sensitive to workloads' locality. S-FTL [19] exploits spatial locality to provide efficient address translation with sophisticated caching strategies. ZFTL [20] partitions the whole flash space into multiple zones, and only mapping information in the recently accessed zone is cached. The active zone is dynamically designated and its mapping information is loaded into the DRAM. CDFTL [21] is another on-demand page-level mapping FTL. It is friendly to a more diverse set of workloads and provides improved performance due to its workload-adaptive prefetching and optimization on reducing the writebacks of the replaced mapping entries. All these approaches take advantage of spatial locality of the workloads and thus share the same drawbacks as DFTL has. Moreover, an on-demand mapping scheme is complicated as it introduces additional operations, such as replacement and writebacks. HP-FTL stores all page-level mappings in DRAM, which avoids the overheads introduced by the caching mechanism. Also, its translation efficiency is less sensitive to access patterns and its I/O performance is resistant to weak locality of workloads.

C. Other approaches to reduce address translation overheads

The Nameless Writes interface [22] is proposed to remove the need for indirection in modern SSDs and expose physical flash addresses to the file system to improve performance. The removal of FTL from SSDs complicates the upper-layer software design as the functionalities provided by FTL, including space allocation, garbage collection, and wear-leveling have to be implemented by the software. Kim et al. [13] proposed an address reshaping technique named SHRD to transform random write requests into sequential write requests in the block device driver by assigning the address space of the reserved log area in the SSD, which improves demand-based mapping schemes, such as DFTL, and makes them more efficient for workloads with many random accesses. However, it may significantly complicate the design of the FTL as it involves extra remapping functions in SSDs as well as the caching mechanism. HP-FTL is not complicated to implement as it only works as a traditional page-level mapping scheme except that two tables with different space efficiency are maintained.

V. Conclusion

In this paper, we propose HP-FTL to enable high-efficient address translation with low space overhead inside SSDs. With HP-FTL, the L2P mapping information is compactly encoded and can fit entirely in the DRAM of the SSD, which is critical to achieving high performance for large-capacity SSDs. We conduct experiments to extensively evaluate its effectiveness. The results show that it can significant reduce mapping table size without compromising I/O performance in terms of latency and throughput. Moreover, the efficiency does not heavily depends on workloads' access locality.

References

[1] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory SSD in enterprise database applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1075–1086.

[2] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh, "Durable write cache in flash memory SSD for relational and nosql databases," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 529–540.

[3] B. Debnath, S. Sengupta, and J. Li, "Flashstore: high throughput persistent key-value store," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1414–1425, 2010.

[4] ——, "Skimpystash: Ram space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 25–36.

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 1–14.

[6] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki, "Evaluating and repairing write performance on flash devices," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. ACM, 2009, pp. 9–14.

[7] S.-H. Kim, D. Jung, J.-S. Kim, and S. Maeng, "Heterodrive: Re-shaping the storage access pattern of oltp workload using ssd," in *Proceedings of 4th International Workshop on Software Support for Portable Storage (IWSSPS 2009)*, 2009, pp. 13–17.

[8] A. Gupta, Y. Kim, and B. Urgaonkar, "*DFTL*: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508271

[9] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 138–163, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1089733.1089735

[10] S. Choudhuri and T. Givargis, "Performance improvement of block based nand flash translation layer," in *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '07. New York, NY, USA: ACM, 2007, pp. 257–262. [Online]. Available: http://doi.acm.org/10.1145/1289816.1289878

[11] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Trans. on Consum. Electron.*, vol. 48, no. 2, pp. 366–375, May 2002. [Online]. Available: http://dx.doi.org/10.1109/TCE.2002.1010143

[12] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "A file is not a file: understanding the i/o behavior of apple desktop applications," *ACM Transactions on Computer Systems (TOCS)*, vol. 30, no. 3, p. 10, 2012.

[13] H. Kim, D. Shin, Y. H. Jeong, and K. H. Kim, "Shrd: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, ser. FAST'17. Berkeley, CA, USA: USENIX Association, 2017.

[14] "Micron technical report (tn-29-07): Small-block vs. large-block nand flash devices." [Online]. Available: http://www.micron.com/products/nand/technotes

[15] "Linux device mapper," https://en.wikipedia.org/wiki/Device_mapper.

[16] C.-H. Wu and T.-W. Kuo, "An adaptive two-level management for the flash translation layer in embedded systems," in *Computer-Aided Design, 2006. ICCAD'06. IEEE/ACM International Conference on*. IEEE, 2006, pp. 601–606.

[17] H.-S. Lee, H.-S. Yun, and D.-H. Lee, "Hftl: hybrid flash translation layer based on hot data identification for flash memory," *IEEE Transactions on Consumer Electronics*, vol. 55, no. 4, 2009.

[18] P.-L. Wu, Y.-H. Chang, and T.-W. Kuo, "A file-system-aware ftl design for flash-memory storage systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 393–398.

[19] S. Jiang, L. Zhang, X. Yuan, H. Hu, and Y. Chen, "S-ftl: An efficient address translation for flash memory by exploiting spatial locality," in *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*. IEEE, 2011, pp. 1–12.

[20] W. Mingbang, Z. Youguang, and K. Wang, "Zftl: a zonebased flash translation layer with a two-tier selective caching mechanism," in *Proceedings of the 14th IEEE International Conference on Communication Technology (ICCT)*, 2011.

[21] Z. Qin, Y. Wang, D. Liu, and Z. Shao, "A two-level caching mechanism for demand-based page-level address mapping in nand flash memory storage systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011, pp. 157–166.

[22] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for flash-based ssds with nameless writes," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208462