A Log-aware Synergized Scheme for Page-level FTL Design

Chu Li*, Dan Feng*✉, Yu Hua*, Fang Wang*, Chuntao Jiang†, Wei Zhou*
*Wuhan National Lab for Optoelectronics
School of Computer, Huazhong University of Science and Technology, Wuhan, China
†Illinois Institute of Technology, Chicago, IL, United States
Email: {lichu,dfeng,csyhua,wangfang,mmbl}@hust.edu.cn, chuntjiang@gmail.com
Corresponding author: Dan Feng (dfeng@hust.edu.cn)

***Abstract*—NAND flash-based Solid State Drives (SSDs) employ the Flash Translation Layer (FTL) to perform logical-to-physical address translation. Modern page-level FTLs selectively cache the address mappings in the limited SRAM while storing the mapping table in flash pages (called translation pages). However, many extra accesses to the translation pages are required for address translation, which decreases the performance and lifetime of an SSD. In this paper, we propose a Log-aware Synergized scheme for page-level FTL to reduce the extra overheads, called LSFTL. The contribution of LSFTL consists of two key elements: (i) By exploiting the partial programmability of SLC flash, *in-place logging* decreases garbage collection overhead via reserving a small portion of each translation page as a logging area to hold multiple updates to the entries of that translation page. (ii) *Log-aware flush back* reduces the number of translation page updates by evicting multiple dirty cache lines that share the same translation page in a single transaction. Extensive experimental results of trace-driven simulations show that LSFTL decreases the system response time by 39.40% on average, and up to 58.35%, and reduces the block erase count by 37.55% on average, and up to 39.99%, compared to the well-known DFTL.**

I. Introduction

NAND flash-based Solid State Drives (SSDs) serve as an important role in diverse environments due to their high performance, power efficiency, and shock resistance. One of the most important concerns about SSD is its reliability, especially for environments with heavy write traffic, such as embedded and enterprise systems [1], [2]. New flash memory technologies such as MLC/TLC (Multi/Triple-Level Cell) can improve the density, but at the cost of sacrificing endurance and reliability compared to SLC (Single-Level Cell). As a result, typical applications, such as enterprise and industrial grade SSDs, still leverage the SLC flash due to its high performance and long-term reliability [3], [4].

SSDs employ a Flash Translation Layer (FTL) to translate logical addresses to physical addresses in flash memory, called address translation [5]. Due to simplicity and efficiency, Demand-based FTL (DFTL) [6] has been one of the most popular FTL designs in the past few years. DFTL selectively caches page-level mappings in SRAM (i.e., mapping cache), and stores the entire mapping table in flash pages (i.e., translation pages). Since the SRAM size is typically constrained for energy and cost concerns, plenty of extra operations to translation pages are caused for the address translation, which degrades the performance and lifetime of SSDs [7], [8].

In this paper, we provide an extensive experimental analysis for DFTL and obtain several useful observations. Specifically, by running trace-driven simulations under various real-world workloads, we find that 41.40-46.37% of the total access time is consumed in the extra operations to translation pages (referred to as *TP Overhead*). We further track where the extra time exactly goes in DFTL and classify them into three parts. (i) The time spent on loading required mapping information from translation pages to the mapping cache, which is caused by mapping cache misses (denoted as *TP Load* overhead). (ii) The time spent on updating translation pages, which is caused by dirty entries being evicted from the mapping cache, or data pages being migrated to new physical locations during garbage collection (denoted as *TP Update* overhead). (iii) The time spent on executing garbage collection operations for translation pages (denoted as *TP GC* overhead). According to our experimental results, TP GC and TP Update respectively consume about 48.47% and 43.08% of TP Overheads.

Based on the above observations, we propose a Log-aware Synergized scheme for page-level FTL design, called LSFTL. The key design of LSFTL is based on two novel techniques, which synergistically reduce the predominant extra overheads caused by TP GC and TP Update. First, by exploiting the partial programmability of SLC flash memory, a portion of a translation page is dedicated as a logging area so that it can hold multiple updates to the entries of that translation page. Through this *in-place logging* policy, the GC-induced overheads are significantly reduced. Second, to reduce the number of TP updates, a *log-aware flush back* policy is proposed to enhance the mapping cache management, which allows the system to evict multiple dirty cache lines that share the same translation page in a single transaction.

The major contributions of this work are as follows:

(1) We present an experimental study of TP Overhead of DFTL. We identify that performance bottlenecks are caused by TP GC and TP Update operations. Only a few mapping entries are modified for most translation page updates.

(2) We propose a new page-level FTL design called LSFTL that employs *in-place logging* and *log-aware flush back* to significantly reduce the overheads of TP Update and TP GC, thus improving SSD's performance and lifetime.

(3) We evaluate LSFTL under various real-world workloads. The experimental results show that compared to DFTL, LSFTL reduces the TP Overhead by 53.35-70.31%, leading to 28.42-58.35% reduction in average system response time and 34.37-39.99% reduction in total block erase count.

II. Background and Motivation

A. Flash Characteristics

NAND flash can be classified into Single/Multi/Triple-Level-Cell (SLC/MLC/TLC), storing one/two/three bits per cell. Due to higher read/write performance, longer retention time and write endurance, SLC becomes an excellent storage medium for high performance and reliability [4], [9].

Recent works have revealed that SLC flash supports per-page partial programming in a progressive manner [9]–[11]. Each cell in SLC flash is in either erased state or programmed state, representing the storage bit of '1' or '0', respectively. A cell can only be changed from '1' to '0' during the program procedure, and an erase operation will reset all bits in a block to '1'. When writing 0 to a cell, circuits of the memory chip apply programming bit-line voltage to it. When writing 1 to a cell, it will be applied prohibitive bit-line voltage to avoid being programmed. This raises the possibility that an SLC flash page can be programmed multiple times on different portions, i.e., partial programming.

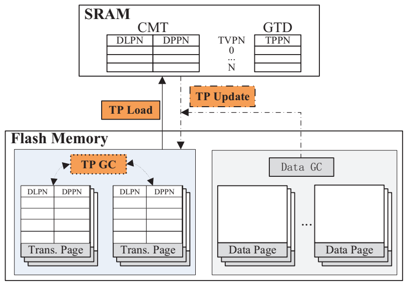


Figure 1: DFTL overview and the TP overhead.

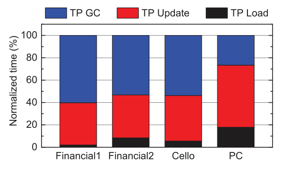
B. Motivation

As shown in Figure 1, DFTL [6] uses page-level mapping and caches a subset of the mappings in SRAM (i.e., mapping cache). The entire mapping table is persistently stored in separate flash pages (i.e., translation pages). In this subsection, we analyze the TP overheads of DFTL by conducting trace-driven simulations under various workloads. Besides DFTL, we also make analyses for a variant of DFTL (referred to as DFTL-E), which adopts an extended cache scheme (see Section III-C) with optimal cache line sizes. Although a larger built-in SRAM can reduce the extra overheads, a relatively small mapping cache is reasonable and necessary [8]. Therefore, the configured cache size is as small as what is required for the block-level FTL, like previous studies [6]–[8]. More detailed configurations are presented in Section IV-A.

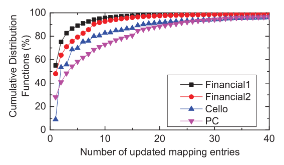
We first study what proportion of the total flash access time is introduced by the extra operations in DFTL(-E). As shown in Table I, TP overheads in DFTL account for 41.40-46.37%, on average 42.90%, of the total access time under

TABLE I: TP overheads in DFTL and DFTL-E

	Financial1	Financial2	Cello	PC
DFTL	41.40%	41.67%	46.37%	42.14%
DFTL-E	28.60%	34.98%	27.10%	11.13%



(a) Breakdown of the TP overhead.



(b) CDFs of number of updated mapping entries when updating a translation page in the flash memory.

Figure 2: The analysis of the TP overheads in DFTL under various workloads.

various workloads. DFTL-E adopts an enhanced cache scheme to achieve more hit rates, resulting in less extra overheads than DFTL. However, we can still observe notable proportions of the overheads from 11.13% to 34.98%, and 25.45% on average. *This observation implies that it is critical to reduce these extra operations to improve the efficiency of DFTL(-E).*

We classify the extra overheads into three parts as described in Section I. We only present the results of DFTL for the sake of simplicity. As shown in Figure 2(a), among the three types, the overheads of TP GC and TP Update account for large percentages in all cases. Specifically, 60.24%, 53.28%, 53.77%, and 26.58% of the overheads are caused by TP GC operations under the four workloads. And for TP Update, the percentages are 37.74-55.56%. *This observation implies that reducing the overheads caused by TP GC and TP Update plays an important role in reducing the TP overheads.*

Existing page-level FTLs [6]–[8] perform "out-of-place" write on each TP update operation, incurring costly GC operations for translation pages. We further study the distribution of the update size, i.e., the number of entries flushed back on each TP update operation. Figure 2(b) shows the cumulative distribution curves of the update size under the four workloads. We can see that the update sizes are quite small. Specifically, a large fraction (more than 90%) of update operations only modify a few entries (less than 25) in all cases. *This observation implies that it would not be cost effective to consume a new flash page because of a few dirty entries.*

Based on these observations, we propose new techniques to mitigate the "out-of-place" TP update by using "in-place logging", which is feasible thanks to the partial programmability of the SLC flash memory. Specifically, dirty entries can be logged into a reserved log area within the same translation page. The basic idea is similar to previous studies such as IPL [12] and PDL [13]. However, unlike LSFTL, these approaches store original data and log entries separately at different flash pages, leading to inherent read amplification [10] (i.e., fetching two or more pages to serve one read request). In addition, LSFTL is complementary to these approaches proposed for data page updates, and reduce overheads caused by translation page accesses. Note that, some commercial MLC/TLC-based SSDs store metadata (e.g., mapping table) in a small portion of flash operating in SLC mode [14]. We argue that our techniques are also applicable to such devices.

III. LSFTL Design

A. Overview

Rather than building from scratch, LSFTL is an enhancement to the demand-based page-level FTL like DFTL [6]. LSFTL stores mappings and data in separate flash blocks. Data blocks store regular data from host, and translation blocks store the entire mapping table. A subset of the mappings are selectively cached in the cached mapping table (CMT), which contains pairs of logical/physical data page numbers—(DLPN, DPPN). The global translation directory (GTD) tracks all the physical locations of the translation pages. Both CMT and GTD are maintained in the built-in SRAM for fast accesses.

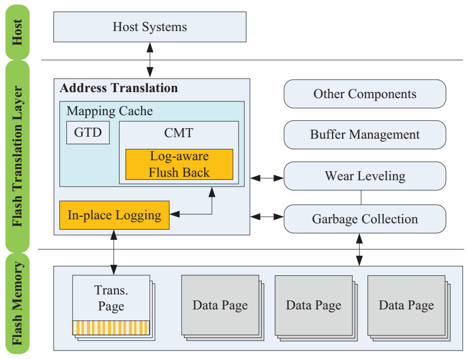


Figure 3: The overview of LSFTL.

The log-aware synergized scheme of LSFTL relies on two new modules, as shown in Figure 3. On each translation page update, the *in-place logging* module tries to avoid consuming new flash pages by only storing the modified entries in the same page. To this end, LSFTL reserves a small portion within each translation page as a log area to accumulate consecutive update operations. To further reduce the TP overhead, the *log-aware flush back* module is applied to the cache management. It flushes more dirty entries each time while striving to achieve high utilization of the log space, thus effectively reducing the number of updates operations to translation pages.

B. In-place Logging

In previous FTLs such as DFTL, even a single dirty entry eviction requires allocating a new flash page. The frequent update operations cause expensive GC operations on translation pages, which account for a large proportion of the TP

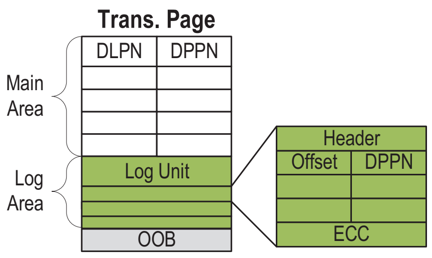


Figure 4: Illustration of the translation page layout in LSFTL.

overhead. In order to mitigate these overheads, we propose an in-place logging strategy by leveraging the partial programming feature of SLC flash memory. Specifically, a small portion in each translation page is reserved as a log area for accumulating new updates. Since most update sizes are quite small as shown in Section II-B, the translation page can be '*overwritten*' multiple times before it is erased, which can significantly reduce the number of TP GC operations.

Figure 4 shows the layout of each translation page in LSFTL. The translation page is partitioned into the main area and the log area. In the main area, mapping entries are sequentially packed according to the DLPNs. The log area consists of multiple log units. Each log unit is comprised of a header and multiple (or one) log entries, all of which are protected with BCH codes [15] for high reliability. The header indicates the number of log entries and uses a special value (e.g., all'0's) as the header marker. Each log entry represents the updated mapping information using a (offset, DPPN) tuple. The offset points to the modified entry within the main area of the same translation page, and the DPPN indicates the new physical address of the data page in the flash memory.

Two counters are maintained to indicate the current status of the log area—one for the tail of the log and one for the number of stored log units. There are two choices of where and how to store the counters. First, we can write them to the OOB (Out-of-Band) area along with each logging operation with virtually no overhead. However, the downside is a lot of OOB reads are incurred for checking the log status. Since GTD stores the addresses of all translation pages, LSFTL adopts an alternative choice which attaches the counters with each entry in GTD for fast accesses. As a result, GTD size is increased, resulting in less cache space for CMT. Fortunately, the impacts on hit ratios are trivial (see Section IV-B2).

The basic process for each translation page update is described below. The two counters are first checked by consulting the GTD. If there is enough log space and the current number of log units is less than a given threshold (*LU threshold*), LSFTL encodes those modified mapping entries into one log unit, appends it to the log tail, and then updates the two counters. Otherwise, LSFTL fetches the translation page, decodes and replays the stored log units, updates the mapping entries, and re-writes the latest mapping information to a new translation page with an empty log area. Note that the LU threshold implies the maximum number of programming operations within one flash page, which is used to avoid noticeable extra physical damage to flash memory cells [10]. Compared to DFTL, the in-place logging strategy brings extra computational costs, such as BCH encoding/decoding, log entries replaying, etc. However, these processing latencies are much less than the page access latencies thanks to the powerful computing capability of modern SSDs [16], [17].

C. Log-aware flush-back

Since each translation page in LSFTL has a relatively small-sized log space and allows storing a limited number of log units, a critical factor in reducing the TP overhead is to improve the utilization of the log area. To this end, we propose another technique, called log-aware flush back, which can be used as an add-on module for the mapping cache schemes. For the convenience of representation, we describe our new approach using an extended cache scheme based on the LRU (Least Recently Used) algorithm.

In the extended cache scheme, the CMT is organized in an LRU list on the cache line basis, which contains one or multiple mapping entries. The cache line is the unit for cache allocation and cache eviction, which is critical for the efficiency of the cache scheme. On the one hand, a too small cache line cannot fully exploit the spatial locality of workloads. On the other hand, a too big cache line can cause cache pollution because many entries may not be accessed after they are loaded into CMT. The optimal cache line size depends on the workload, usually in the range from one to the number of entries stored in one translation page. Therefore, it is not uncommon for the cache lines belonging to the same translation page to be scattered across the LRU list of CMT.

Whenever a victim cache line which has dirty mapping entries is evicted, the corresponding translation page must be updated. If the cache lines which share the same translation page are flushed separately, it may produce multiple small-sized log units. As a result, log space may be underutilized when the LU threshold of the translation page is reached. On the contrary, if we always flush all the dirty entries which belong to the same translation page, it may reduce the benefit of write cancellation (i.e., the ability to absorb writes) in CMT and produce more log entries for the log units. Consequently, log space of the translation page may be exhausted before the LU threshold is reached. Based on these analyses, our log-aware flush back mechanism uses a simple heuristic to achieve a balance between the two strategies by considering the status of the log area. Specifically, we set a quota on the size of the log unit for each flush back operation as follows.

$$Quota = \begin{cases} LCrem/LTrem; & \text{if } LTrem > 0 \\ 0; & \text{if } LTrem = 0 \end{cases} \quad (1)$$

$LCrem$ is the remaining capacity of the log area, and $LTrem$ is the remaining times allowed for appending new log units. According to the calculated $Quota$, we further formalize the proposed log-aware flush-back mechanism. Let $C_0, C_1, \ldots, C_{m-1}$ (C_0 is the victim cache line) represent the cache lines which have dirty entries and share the same translation page, and D_i be the number of dirty entries in each C_i, $0 \leq i < m$. The set of cache lines which will be flushed back through in-place logging can be denoted as follows.

$$\begin{aligned} S = \{C_i |\, & \overbrace{Func(\sum_{i=0}^{m-1} D_i) \leq Quota \text{ OR}}^{\text{Condition 1}} \\ & \underbrace{(i = 0 \text{ and } Func(D_i) \leq LCrem)}_{\text{Condition 2}}\} \end{aligned} \quad (2)$$

The $Func()$ represents a simple function which transforms the number of dirty entries to the size of the encoded log unit. For each cache eviction, the Condition 1 in Equation 2 is used to batch as many dirty entries as possible for flush-back, while avoiding exhausting the log space too soon. Thus, the TP update operations can be effectively reduced. Occasionally, the log unit size derived from the victim cache line exceeds the $Quota$ but there is enough remaining log space. In this case, we ignore the $Quota$ and use in-place logging to postpone the allocation for a new flash page, as shown in Condition 2. If $S = \emptyset$, it flushes back all dirty entries that share the same translation page using out-of-place updates.

D. Process Flow in LSFTL

1) Address translation: For each page access, the request can be serviced directly if it hits in the CMT. Otherwise, cache eviction may be triggered on a cache miss. If some entries within the victim cache line have been modified since the time they were loaded into CMT, LSFTL updates the translation page with the log-aware flush back policy. If an out-of-place update is triggered, the GTD needs to be updated to reflect the change. To load the mapping entries into CMT, LSFTL first obtains the physical address of the corresponding translation page (TPPN) by querying the GTD according to the virtual translation page number (TVPN), which is the quotient of DLPN divided by the number of mapping entries stored in each translation page‘s main area. Then, LSFTL fetches the translation page and decodes the log units to generate the required mapping information for servicing the request.

2) Read/Write and GC: For a read request, it is simply handled by sending flash page read operations according to the DPPNs found in CMT. For write requests, they will be written to free flash data pages, after which the old data pages are invalidated and the CMT is updated. When the number of free blocks in flash memory is below a certain threshold, GC is invoked to reclaim the invalid pages and erase the flash blocks. The victim flash block is chosen by using a cost-benefit strategy like DFTL. If it is a translation block, mapping entries of each valid translation page are first updated by applying the log units, then copied to new flash pages, and finally the GTD is updated. Otherwise, LSFTL copies the valid data pages into new flash pages, and updates the associated mapping entries. For the entries residing in CMT, they are updated in the SRAM without extra flash operations. Otherwise, they are written to flash using the proposed in-place logging strategy. Finally, the victim block is erased and added into the free flash block pool.

IV. Evaluation

A. Experimental Setup

We measure the effectiveness of the proposed techniques with the Flashsim simulator [6]. Implemented FTLs include DFTL(-E), LSFTL(-E), and an optimal FTL which has no translation page accesses (Optimal). LSFTL employs the same cache scheme as DFTL. DFTL-E/LSFTL-E adopt the extended cache scheme with optimal cache line sizes. They are evaluated to show that our techniques are enhancements to the framework of the demand-based page-level FTLs [6]–[8].

The simulated SSD [18] is configured with 4KB page size and 256KB block size. The page read time, page write time, and block erase time are set as 25μs, 200μs, and 1.5ms, respectively. SSD capacity is set according to the logical address space of the workloads. The default mapping cache size is set as what is required for the block-level FTL as previous studies [6]–[8]. Specifically, a 4GB SSD with 64KB cache is simulated under the Financial workloads, and an 8GB SSD with 128KB cache under other workloads. For LSFTL and LSFTL-E, the proportion of log area in each translation page is set to 25%, and the LU threshold is set to 3 (according to [11]) unless otherwise stated.

We choose several real-world traces from both enterprise-scale and consumer environments. The Financial traces are collected from OLTP applications running at large financial institutions provided by the Storage Performance Council (SPC) [19]. The Cello99 trace is collected from a time-sharing server which runs the HP-UX operating system at HP Laboratories [20]. Besides these server workloads, we also collected block-level traces by monitoring I/O accesses to a small system disk partition from a desktop PC. Table II shows the main characteristics of these workloads. For DFTL-E and LSFTL-E, we first obtain the optimal cache line sizes under four workloads, which contain 32, 16, 256, and 256 mapping entries, respectively.

TABLE II: The Workload Characteristics

Workloads	Avg. Req. Size (KB)	Req. Num	Read (%)	Address Space
Financial1 [19]	6.87	5,334,945	19.41	4G
Financial2 [19]	5.88	3,698,863	81.05	4G
Cello [20]	10.54	259,892	30.80	8G
PC	19.63	305,057	57.27	8G

B. Experimental Results

1) TP Overhead: Figure 5 plots the TP overheads of different FTLs under various workloads. Compared to DFTL, LSFTL significantly reduces the overheads by 53.35-70.31%. The reasons are twofold. First, LSFTL reduces the TP GC overhead by 84.80-89.09% compared to DFTL due to consuming less translation pages using in-place logging strategy. Second, TP Update overhead is reduced by 46.28-55.70% in LSFTL due to the log-aware flush-back mechanism. Compared to DFTL-E, LSFTL-E also reduces the TP overheads by 40.91-58.78%, which indicates our approaches can be used together with improved cache schemes to reduce the TP overhead.

2) Cache Hit ratios: Figure 6(a) shows the mapping cache hit ratios of the FTLs. LSFTL(-E) consumes more SRAM space due to the increase of GTD compared with DFTL(-E), however, the degradations of their hit ratios are negligible under all workloads. Specifically, the differences of the cache hit ratios between DFTL and LSFTL are 0.29%, 1.11%, 1.35%, and 0.78% under four workloads, respectively. And for DFTL-E and LSFTL-E, the differential hit ratios are less than 0.70%.

3) System Response Time: Figure 6(b) shows the normalized average system response times for different FTLs. We can

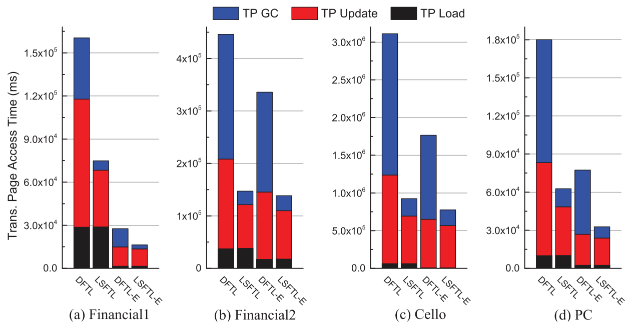


Figure 5: The TP overheads of different FTL schemes under various workloads.

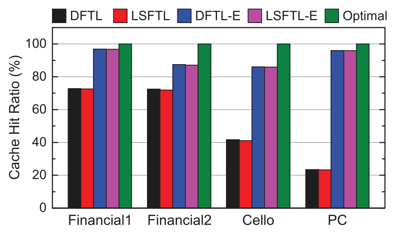
see from Figure 6(b) that LSFTL(-E) consistently outperforms DFTL(-E) under all workloads. Compared to DFTL, LSFTL reduces the average system response time by 38.89%, 31.92%, 28.42%, and 58.35% under the four workloads, respectively. Compared to DFTL-E, LSFTL-E also achieves significant advantages in the workloads with 25.09%, 28.10%, 25.31%, and 11.18% performance improvements, respectively. The results validate that reducing the TP overheads plays an important role in improving the FTL performance.

4) Block Erase Counts: Figure 6(c) shows the total block erase counts of different FTL schemes. The optimal FTL scheme has the least block erase counts since no TP overheads are induced. Compared to DFTL(-E), LSFTL(-E) reduces the block erase counts by 34.37-39.99% (8.14-32.76%). The advantages of LSFTL-E over DFTL-E are not as dramatic as those of LSFTL over DFTL. This is because by using more efficient mapping cache schemes, GC operations of data blocks account for a larger proportion of the total block erase counts. Nonetheless, by using the proposed techniques, our new FTL schemes consistently reduce the block erase counts compared to the FTL counterparts, thus improving the lifetime of SSDs.

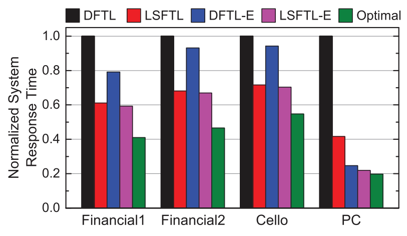
5) Impact of LU Threshold: Figure 7 shows the results of LSFTL with LU threshold ranging from 1 to 7 (according to [10]). When the LU threshold is increased to 7, the TP overhead is reduced by 13.57-32.38% under various workloads. This is because more TP updates can be accumulated with larger LU thresholds. The average system response time and the total block erase count are reduced by 10.21-15.50% and 9.76-15.31%, respectively. We omit the results of LSFTL-E where similar trends are found.

V. Conclusion

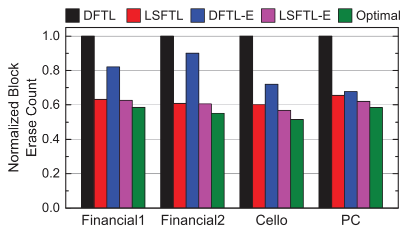
In this paper, we propose LSFTL to reduce the overheads involved with translation page accesses. LSFTL uses an *in-place logging* strategy by exploiting the partial programmability of SLC flash, through which LSFTL can ‘overwrite’ the translation page multiple times in one erase cycle, thus significantly reducing the TP GC overhead. In addition, a *log-aware flush-back* policy is used to improve the log utilization and reduce the TP Update overhead. We conduct extensive trace-driven simulations to demonstrate the effectiveness of LSFTL. Results show that LSFTL significantly improves both the performance and lifetime of SSDs. We also make compar-



(a) Cache hit ratio

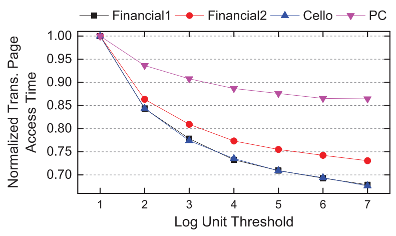


(b) System response time

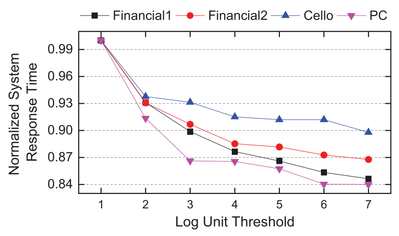


(c) Block erase count

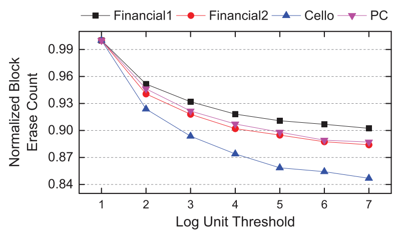
Figure 6: Cache hit ratio, average system response time, and block erase count with different FTL schemes and workloads.



(a) TP overhead



(b) System response time



(c) Block erase count

Figure 7: Impact of Log Unit Threshold on the TP overhead, system response time, and block erase count of LSFTL.

isons by using advanced mapping cache policy for the FTLs, and our techniques still achieve notable improvements.

Acknowledgment

This work was supported by the National Key Research and Development Program of China under Grant 2016YFB1000202; the National High Technology Research and Development Program (863 Program) No.2015AA016701, No.2015AA015301; NSFC No.61303046, No.61472153, No.61402189, No.61502191; State Key Laboratory of Computer Architecture, No.CARCH201505; Wuhan Applied Basic Research Project (No.2015010101010004); This work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China.

References

[1] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing flash memory: Anomalies, observations, and applications," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2009, pp. 24–33.
[2] J. H. Kim, S. H. Kim, and J. S. Kim, "Subpage programming for extending the lifetime of nand flash memory," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 555–560.
[3] J. Kim, E. Lee, J. Choi, D. Lee, and S. H. Noh, "Chip-level raid with flexible stripe size and parity placement for enhanced ssd reliability," *IEEE Transactions on Computers*, vol. 65, no. 4, pp. 1116–1130, April 2016.
[4] "Slc vs. mlc: An analysis of flash memory," http://www.supertalent.com/datasheets/SLC_vs_MLCwhitepaper.pdf, 2012.
[5] D. Ma, J. Feng, and G. Li, "A survey of address translation technologies for flash memories," *ACM Comput. Surv.*, vol. 46, no. 3, pp. 36:1–36:39, Jan. 2014.
[6] 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*. ACM, 2009, pp. 229–240.
[7] 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 *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2011, pp. 1–12.
[8] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou, "An efficient page-level FTL to optimize address translation in flash memory," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys)*. ACM, 2015, pp. 12:1–12:16.
[9] "Nand flash memories application note," https://www.elnec.com/sw/an_elnec_nand_flash.pdf, jan 2014.
[10] X. Zhang, J. Li, H. Wang, K. Zhao, and T. Zhang, "Reducing solid-state storage device write stress through opportunistic in-place delta compression," in *14th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 111–124.
[11] "1 gbit, 2 gbit, 4 gbit, 3 v slc nand flash for embedded," http://www.cypress.com/file/207521/download, apr 2016.
[12] S.-W. Lee and B. Moon, "Design of flash-based dbms: An in-page logging approach," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2007, pp. 55–66.
[13] Y.-R. Kim, K.-Y. Whang, and I.-Y. Song, "Page-differential logging: An efficient and dbms-independent approach for storing data into flash memory," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. New York, NY, USA: ACM, 2010, pp. 363–374.
[14] D. Sharma, "System design for mainstream TLC SSD," in *Proc. of Flash Memory Summit*, aug 2014.
[15] S. Lin and D. J. Costello, *Error Control Coding, Second Edition*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2004.
[16] Y. Kang, Y. s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, May 2013, pp. 1–12.
[17] Y.-S. Lee, L. C. Quero, Y. Lee, J.-S. Kim, and S. Maeng, "Accelerating external sorting via on-the-fly data merge in active SSDs," in *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. Philadelphia, PA: USENIX Association, Jun. 2014.
[18] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX 2008 Annual Technical Conference (ATC)*. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70.
[19] *OLTP Trace from UMass Trace Repository*, http://traces.cs.umass.edu/index.php/Storage/Storage, 2010.
[20] *HP Labs: Tools and Traces*, http://tesla.hpl.hp.com/public_software/.