An Efficient Page-level FTL to Optimize Address Translation in Flash Memory

You Zhou Fei Wu [*]
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China
{zhouyou, wufei}@hust.edu.cn

Ping Huang Xubin He
Department of Electrical and Computer Engineering, Virginia Commonwealth University, USA
{phuang, xhe2}@vcu.edu

Changsheng Xie Jian Zhou
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China
cs_xie@hust.edu.cn, jzhou.research@gmail.com

Abstract

Flash-based solid state disks (SSDs) have been very popular in consumer and enterprise storage markets due to their high performance, low energy, shock resistance, and compact sizes. However, the increasing SSD capacity imposes great pressure on performing efficient logical to physical address translation in a page-level flash translation layer (FTL). Existing schemes usually employ a built-in RAM cache for storing mapping information, called the *mapping cache*, to speed up the address translation. Since only a fraction of the mapping table can be cached due to limited cache space, a large number of extra operations to flash memory are required for cache management and garbage collection, degrading the performance and lifetime of an SSD. In this paper, we first apply analytical models to investigate the key factors that incur extra operations. Then, we propose an efficient page-level FTL, named *TPFTL*, which employs two-level LRU lists to organize cached mapping entries to minimize the extra operations. Inspired by the models, we further design a workload-adaptive loading policy combined with an efficient replacement policy to increase the cache hit ratio and reduce the writebacks of replaced dirty entries. Finally, we evaluate TPFTL using extensive trace-driven simulations. Our evaluation results show that compared to the state-of-the-art FTLs, TPFTL reduces random writes caused by address translation by an average of 62% and improves the response time by up to 24%.

[*] Corresponding author: wufei@hust.edu.cn

EuroSys'15, April 21–24, 2015, Bordeaux, France.
Copyright © 2015 ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741949

1. Introduction

NAND flash-based solid state drives (SSDs) [2] have been increasingly deployed in portable devices, personal computers as well as enterprise storage systems as persistent storage devices [12]. Built with semiconductor chips without any mechanically moving components, SSDs offer superior performance, low power consumption and shock resistance in small sizes. Moreover, the SSD market keeps growing due to the drop of its per-bit cost [29].

However, flash memory exhibits unique features, such as asymmetric read/write operations, the erase-before-write feature, and limited endurance. To hide these features and emulate a standard block device, a *flash translation layer* (FTL) [10] is employed to manage flash memory. Since data in flash memory cannot be updated in place due to the erase-before-write feature, the FTL simply writes incoming data to a free page, which is the unit of read and write operations [31], and invalidates the previously-mapped page. As a result, address mapping is required to translate a host logical address to a physical flash address, more narrowly, to translate a *logical page number* (LPN) to a *physical page number* (PPN), which is called *address translation*. Each flash block consists of a number of pages and is the unit of an erase operation. When the amount of reserved free blocks falls below a threshold, a garbage collection (GC) process is performed to reclaim invalidated pages by relocating valid pages and erasing victim blocks. Note that each block can only sustain a limited number of erasures, and erasure is about ten times slower than write, which is much slower than read [48]. Due to slow writes and GC operations, random writes degrade the performance and shorten the lifetime of an SSD, and thus should be reduced in flash memory [33].

To support fast address translation, the FTL leverages built-in RAM to cache the LPN-to-PPN mapping table in the mapping cache. With the increasing capacity of SSDs, the mapping table grows too large to be cached, especially when the FTL adopts a page-level mapping scheme, which has been demonstrated to show better performance [14]. For example, assuming the flash page size is 4KB and each LPN-to-PPN mapping entry takes 8B, a 1TB SSD would require 2GB RAM to accommodate the mapping table.

Although modern SSDs are having increasing built-in RAM to accommodate the expanding capacity [41], we believe the assumption of a relatively small mapping cache is reasonable and necessary, and it is still beneficial to reduce the RAM requirement for at least four reasons. First, only a small proportion of RAM space is dedicated to the mapping cache while the remaining is used for buffering user data [49] and storing other FTL metadata, such as priority queues for garbage collection and wear leveling [7], multiple request queues managed by a scheduler [52], even a fingerprint store for deduplication [5]. Moreover, reducing the RAM requirement for the mapping cache can increase the efficiency of other FTL functions. Second, large RAM significantly increases the cost on acquisition and energy consumption, as well as vulnerability to a power failure or system crash. Third, some SSDs employ the internal SRAM of flash controllers to store the mapping table [42]. Since SRAM is faster but much smaller than DRAM [3], the cache size would be less than a few megabytes. Finally, non-volatile Memory (NVM) technologies (e.g. phase-change memory [40]), which are persistent and fast, can be alternatives to the built-in DRAM or SRAM [46], but they are currently small in scale and capacity due to high cost. Note that in some special cases, such as the Fusion-IO PCIe SSD [9] and the SSD design in [4], the FTL utilizes the large host RAM to store mapping tables.

Several demand-based page-level FTLs have been proposed to reduce the RAM requirement. In those FTLs, a full mapping table is packed into pages (called *translation pages*) in the order of LPNs in flash memory, and only a small set of active mapping entries are held in the cache [14, 19, 39]. However, their caching mechanisms are based on the LRU (Least Recently Used) algorithm, which may be inefficient when directly adopted in mapping cache management. A translation page, ranging from 2KB to 8KB, is the access unit of the mapping table in flash memory, while the access unit of the mapping cache is an entry, which typically takes 8B. These two different access units compel cache management to cause substantial overhead. During the address translation phase, a cache miss incurs an extra page read to load the requested entry into the cache, which can happen more frequently if a high hit ratio cannot be guaranteed. When the cache is full and dirty entries are replaced, they need to be written back to flash memory to maintain the consistency of the mapping table. Thus, a large number of extra translation page writes are introduced, together with associated GC operations, in write-dominant workloads. Therefore, extra operations caused by inefficient caching mechanisms may potentially lead to severe degradations in both the performance and lifetime of an SSD.

In this paper, we aim to optimize the address translation in flash memory from the mapping cache management perspective. We first apply analytical models to investigate the key factors that incur extra operations in flash memory and then design a new demand-based page-level FTL with a translation page-level caching mechanism, called *TPFTL*, to reduce the extra operations caused by address translation with a small mapping cache. Specifically, our contributions include the following:

- We develop two models to analyze the overhead of address translation in a demand-based page-level FTL and show how address translation influences the performance and lifetime of an SSD.
- We propose a novel FTL, TPFTL. Considering the different access units between flash memory and the mapping cache, TPFTL clusters the cached mapping entries that belong to the same translation page, using the limited cache space efficiently.
- We propose a workload-adaptive loading policy to exploit the spatial locality to improve the cache hit ratio and an efficient replacement policy to reduce the replacement cost.
- We evaluate TPFTL with various enterprise workloads. Results show that TPFTL reduces page reads and writes caused by address translation by an average of 26.6% and 62%, respectively, and improves system response time by up to 24%, compared to the state-of-the-art FTLs.

The rest of the paper is organized as follows. Section 2 provides an overview of the background and related work. Section 3 presents our analytical models, observations and motivation. Section 4 describes the proposed TPFTL scheme. We evaluate TPFTL in Section 5 and conclude the paper in Section 6.

2. Background and Related Work

2.1 Solid State Drives

In this paper, flash memory refers to NAND flash memory specifically. Flash memory has made significant strides into the consumer and enterprise storage markets in the form of SSDs. An SSD mainly consists of three components: a software layer FTL, an internal RAM, and flash memory.

The FTL splits I/O requests into page accesses and performs LPN-to-PPN address translation. According to the granularity of address mapping, FTL schemes can be classified into three categories: page-level FTLs, block-level FTLs, and hybrid FTLs [6]. A page-level FTL maximizes the performance and space usage by maintaining a fine-grained mapping, in which a logical page can be mapped to any physical page. However, the page-level FTL consumes a prohibitively large RAM space for the whole mapping table. A block-level FTL is a coarse-grained mapping scheme and requires much less RAM. As pages can only go to fixed locations within blocks, the performance of a block-level FTL is very poor as a result of maintaining such a rigid mapping regularity. To make a compromise, both log-buffer based hybrid FTLs [23, 24, 43] and workload-adaptive hybrid FTLs [37, 45] are proposed, where both the block-level mapping and page-level mapping are employed to manage flash memory. Hybrid FTLs have better performance than block-level FTLs and require less RAM space than page-level FTLs, but they suffer from performance degradation in random write intensive workloads due to costly extra operations to couple the two mapping schemes. Since the page-level FTL provides the best performance, our work focuses on how to maintain its high performance under the condition of a small mapping cache.

The internal RAM serves as both a data buffer and mapping cache, and flash memory stores both user data and the mapping table. Since RAM is much faster than flash memory and supports overwrites, an internal RAM can improve the performance and lifetime of an SSD. As a data buffer, the RAM not only accelerates data access speed, but also improves the write sequentiality and reduces writes in flash memory, which has been well studied [21, 38, 46, 49]. As a mapping cache, the RAM accelerates address translation, but its management introduces extra operations to flash memory in a demand-based page-level FTL.

2.2 Accelerating Address Translation

Gupta et al. [14] proposed the first demand-based page-level FTL, called *DFTL*, to avoid the inefficiency of hybrid FTLs and to reduce the RAM requirement for the page-level mapping table. Taking temporal locality into consideration, DFTL adopts a segmented LRU replacement algorithm to keep recently used mapping entries in the cache. Hence, DFTL achieves good performance under workloads with strong temporal locality.

To further exploit spatial locality, CDFTL [39] and S-FTL [19] are proposed to improve the hit ratio and thus the performance. A translation page contains mapping entries whose LPNs are consecutive so that caching a translation page can effectively serve the sequential accesses. CDFTL designs a two-level LRU caching algorithm for the demand-based page-level FTL. The first-level cache, called *CMT*, stores a small number of active mapping entries, while the second-level cache, called *CTP*, selectively caches a few active translation pages and serves as the kick-out buffer for CMT. Replacements of dirty entries only occur in CTP and dirty entries in CMT won't be replaced unless they are also included in CTP. Hence, cold dirty entries reside in CMT. S-FTL employs an entire translation page instead of a mapping entry as the caching object and organizes recently used pages in an LRU list. To reduce space consumption, S-FTL shrinks the size of each cached translation page according to the sequentiality of PPNs of the entries in the page. In addition, a small cache area is reserved as dirty buffer to postpone the replacement of sparsely dispersed dirty entries. To ensure a small and stable consumption of cache space, ZFTL [34] divides the whole flash into several partitions, called *Zones*, and only caches the mapping information of a recently accessed Zone. However, Zone switches are cumbersome and incur significant overhead. In addition, ZFTL employs a two-tier caching mechanism, where the second-tier cache stores an active translation page and the first-tier cache reserves a small area to conduct batch evictions.

HAT [16] integrates a solid-state chip, such as PCM [40], to store the entire mapping table so that it can access the data and mapping table in parallel. However, it introduces additional hardware cost and new challenges of employing an emerging nonvolatile memory technology. APS and TreeFTL [44] present uniform caching algorithms for both the mapping cache and data buffer, which adjust the two partitions adaptively according to the workloads to maintain stable and good performance. Another work, Nameless Writes [53], is proposed to remove the FTL mapping layer and exposes physical flash addresses to a new designed file system to improve the performance.

Among these designs, our TPFTL is most related to CDFTL and S-FTL, which improve the efficiency of mapping cache management in a demand-based page-level FTL. The major differences are: (1) CDFTL and S-FTL are efficient only in sequential workloads, while TPFTL is friendly to various workloads due to its different page-level caching mechanism, which performs workload-adaptive prefetching; (2) Besides increasing the cache hit ratio, TPFTL goes further to reduce the writebacks of replaced dirty entries.

2.3 Extending an SSD's Lifetime

Since the write amplification corresponds to the amount of additional writes beyond user writes, it is a critical factor that affects the lifetime of SSDs and has received much attention. A probabilistic model [15] and an analytical model [51] are proposed to analyze the write amplification caused by relocations of valid pages during GC operations. After analyzing the write amplification from file system, OFTL [29] is proposed to move storage management from the file system to the FTL to reduce file system writes, and ReconFS [30] is proposed to reduce namespace metadata writebacks. Write amplification from various protection schemes, such as ECC, is modeled and analyzed in [35]. Write amplification of a flash-based cache device is analyzed and reduced by optimizations at both the cache and flash management in [50]. More generally, the overall write amplification is an effective indicator to show the efficiency of techniques proposed to reduce writes in flash memory [11, 18, 28]. In this paper, we focus on the write amplification caused by address translation.

In addition, various FTL designs are proposed to extend an SSD's lifetime. CAFTL [5] integrates de-duplication techniques into the FTL to eliminate duplicate writes. ΔFTL [47] exploits content locality between new data and its old version in flash memory to eliminate redundant writes. A thorough study on exploiting data compressibility to improve the lifetime is presented in [26]. Efficient garbage collection policies are studied to reduce the relocations of valid pages when reclaiming invalid pages [22]. Wear leveling techniques [20, 36] ensures that the full erase cycles of every flash block are exploited so as to avoid a device failure due to some blocks being worn out earlier.

3. Models, Observations and Motivation

Before moving on to our design, we first analyze the impact of address translation on both the performance and lifetime of an SSD which employs a demand-based page-level FTL. Then we present our observations and motivation.

3.1 Analytical Models

To clearly understand the problem, we have developed two models based on a demand-based page-level FTL to perform an in-depth analysis on the overhead of address translation: a performance model and a write amplification model. Table 1 gives a list of symbols used in our analysis.

1) ***Performance Model***

The time to serve a page request mainly includes three parts: address translation, user page access and garbage collection if needed. A user page access is to either read a data page in time T_{fr} or write a data page in time T_{fw} in flash memory.

Before accessing a data page in flash memory, an LPN-to-PPN address translation is performed. With rate H_r, the required mapping entry is present in the mapping cache, called *a cache hit*. Since RAM is about three orders of magnitude faster than flash memory, the time of accessing RAM can be negligible. Thus address translation can be done immediately with a cache hit. With rate $1 - H_r$, the required mapping entry has to be obtained by reading its translation page in flash memory in time T_{fr} and then be loaded into the cache, called *a cache miss*. Moreover, if the cache is full, an entry, called a *victim*, needs to be evicted to make room for the required entry. With probability P_{rd}, the victim is dirty and has to be written back to flash memory, resulting in a partial overwrite of a translation page in time $T_{fr} + T_{fw}$. In conclusion, the average time of an LPN-to-PPN translation can be derived as Equation 1.

$$T_{at} = (1 - H_r) * [T_{fr} + P_{rd} * (T_{fr} + T_{fw})] \qquad (1)$$

Note that this equation does not include some special cases, such as S-FTL [19], where the victim is an entire translation page and its writeback would be in time T_{fw}.

Table 1: A list of symbols.

Symbol	**Description**
A	Write amplification in flash memory
H_{gcr}	Hit ratio of modified mapping entries of migrated pages during GC in mapping cache
H_r	Hit ratio of LPN-to-PPN address translation in mapping cache
N_{dt}	Number of translation page updates caused by migrating data pages in GC operations
N_{gcd}	Number of GC operations for data blocks
N_{gct}	Number of GC operations for translation blocks
N_{md}	Number of data page migrations in GC operations
N_{mt}	Number of migrations of translation pages in GC operations
N_p	Number of pages in a flash block
N_{pa}	Number of user page accesses in the workload
N_{tw}	Number of translation page writes during the address translation phase
P_{rd}	Probability of replacing a dirty entry in mapping cache
R_w	Ratio of page writes among user page accesses
T_{at}	Average time of an LPN-to-PPN address translation
T_{fe}	Time for a block erase in flash memory
T_{fr}	Time for a page read in flash memory
T_{fw}	Time for a page write in flash memory
T_{gcd}	Average time of collecting data blocks per user page access
T_{gct}	Average time of collecting translation blocks per user page access
$\overline{V_d}$	Mean of valid pages in collected data blocks
$\overline{V_t}$	Mean of valid pages in collected translation blocks

When running out of free blocks, a GC process is performed. A GC operation consists of three steps: (1) choosing a victim block, either a data block or a translation block; (2) migrating valid pages remaining in the block to free pages and updating their mapping entries if the victim block is a data block; (3) erasing the block in time T_{fe} and moving it to the free block list. When the victim block is a data block, with rate H_{gcr}, the mapping entry of a migrated page is present in the cache, called *a GC hit*. With rate $1 - H_{gcr}$, the mapping entry has to be updated in flash memory in time $T_{fr} + T_{fw}$, called *a GC miss*.

Assume that N_{gcd} is the number of GC operations performed to collect data blocks, and $\overline{V_d}$ is the mean of valid pages in collected data blocks. The number of data page writes caused by migrating valid data pages is

$$N_{md} = N_{gcd} * \overline{V_d}\,. \qquad (2)$$

The number of translation page writes caused by updating the mapping entries of migrated data pages is

$$N_{dt} = N_{gcd} * \overline{V_d} * (1 - H_{gcr})\,. \tag{3}$$

Collecting data blocks includes migrating valid data pages in time $N_{md} * (T_{fr} + T_{fw})$, erasing data blocks in time $N_{gcd} * T_{fe}$, and updating translation pages in flash memory in time $N_{dt} * (T_{fr} + T_{fw})$. Hence, assuming the number of user page accesses in the workload is N_{pa}, the average time of collecting data blocks per user page access can be derived as Equation 4 from Equations 2 and 3.

$$T_{gcd} = \frac{N_{gcd} * [\overline{V_d} * (2 - H_{gcr}) * (T_{fr} + T_{fw}) + T_{fe}]}{N_{pa}} \tag{4}$$

Assume that N_{gct} is the number of GC operations to collect translation blocks, and $\overline{V_t}$ is the mean of valid pages in collected translation blocks. The number of translation page writes caused by migrating valid translation pages is

$$N_{mt} = N_{gct} * \overline{V_t}\,. \tag{5}$$

Collecting translation blocks includes migrating valid translation pages in time $N_{mt} * (T_{fr} + T_{fw})$ and erasing translation blocks in time $N_{gct} * T_{fe}$. Thus the average time of collecting translation blocks per user page access can be derived as Equation 6 from Equation 5.

$$T_{gct} = \frac{N_{gct} * [\overline{V_t} * (T_{fr} + T_{fw}) + T_{fe}]}{N_{pa}} \tag{6}$$

The average gain of free pages from collecting a data block with $\overline{V_d}$ valid pages is $N_p - \overline{V_d}$ and each page write consumes a free page. The number of user page writes is $N_{pa} * R_w$, assuming the SSD is in full use, N_{gcd} can be described as Equation 7.

$$N_{gcd} = \frac{N_{pa} * R_w}{N_p - \overline{V_d}} \tag{7}$$

According to the derivation of Equation 1 , the number of translation page writes during the address translation phase is given by Equation 8.

$$N_{tw} = (1 - H_r) * P_{rd} * N_{pa}\,. \tag{8}$$

Since the number of translation page writes, excluding the writes caused by collecting translation blocks, is $N_{tw} + N_{dt}$, N_{gct} can be described as Equation 9, similar to N_{gcd}.

$$N_{gct} = \frac{N_{tw} + N_{dt}}{N_p - \overline{V_t}} \tag{9}$$

In conclusion, the average time of collecting data blocks and translation blocks per user page access can be derived as Equation 10 (from Equations 4 and 7) and 11 (from Equations 3, 6, 8, and 9), respectively.

$$T_{gcd} = \frac{R_w * [\overline{V_d} * (2 - H_{gcr}) * (T_{fr} + T_{fw}) + T_{fe}]}{N_p - \overline{V_d}} \tag{10}$$

$$\begin{aligned} T_{gct} =& [(1 - H_r) * P_{rd} + \frac{R_w * \overline{V_d} * (1 - H_{gcr})}{N_p - \overline{V_d}}] * \\ & \frac{\overline{V_t} * (T_{fr} + T_{fw}) + T_{fe}}{N_p - \overline{V_t}} \end{aligned} \tag{11}$$

From Equations 1, 10 and 11, we see that address translation and GC introduce extra access cost in flash memory, which is influenced by H_r, P_{rd}, R_w, $\overline{V_d}$, $\overline{V_t}$ and H_{gcr}. Since we focus on the cost caused by address translation, R_w, which relies on the specific workload, $\overline{V_d}$, $\overline{V_t}$ and H_{gcr}, which are decided by the over-provisioning configuration and the choice of a GC policy, are beyond the scope of this paper. Then, two conclusions can be drawn. First, address translation leads to direct and indirect cost, degrading the performance of an SSD. The direct cost is incurred by cache misses and replacements of dirty entries in the mapping cache during the address translation phase, and the indirect cost is incurred by GC misses and collecting translation blocks during GC operations. Second, both the direct and indirect cost are subject to two factors: H_r and P_{rd}. *The address translation cost can be reduced by either increasing the cache hit ratio and/or reducing the probability of replacing a dirty entry.*

2) ***Write Amplification Model***

As discussed in Section 2.3, write amplification is a good metric to quantify the negative effect of extra writes, which refer to writes that are beyond user page writes ($N_{pa} * R_w$). The writebacks of replaced dirty entries cause translation page writes to flash memory (N_{tw}), and GC operations incur both data page writes (N_{md}) and translation page writes ($N_{dt} + N_{mt}$) to flash memory. Assuming the workload is not read-only ($R_w > 0$), the write amplification is

$$A = \frac{N_{pa} * R_w + N_{tw} + N_{md} + N_{dt} + N_{mt}}{N_{pa} * R_w}\,. \tag{12}$$

Incorporating Equations 2, 3, 5, 7, 8 and 9, the write amplification can be derived as Equation 13.

$$\begin{aligned} A =& 1 + (1 - H_r) * P_{rd} * \frac{N_p}{(N_p - \overline{V_t}) * R_w} + \\ & [1 + (1 - H_{gcr}) * \frac{N_p}{N_p - \overline{V_t}}] * \frac{\overline{V_d}}{N_p - \overline{V_d}} \end{aligned} \tag{13}$$

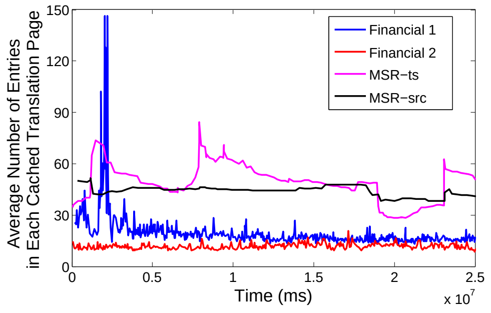
From Equations 12 and 13, we see address translation and GC cause extra writes to flash memory and increase the write amplification, which is influenced by H_r, P_{rd}, R_w, $\overline{V_d}$, $\overline{V_t}$ and H_{gcr}. Similar to the performance model, R_w, $\overline{V_d}$, $\overline{V_t}$ and H_{gcr} are beyond the scope of this paper, and two conclusions can be drawn. First, extra writes caused by address translation come from two aspects: (1) replacements of dirty entries in the mapping cache during the address translation phase; (2) GC misses and migrations of valid translation pages during GC operations. Second, the number of extra writes caused by address translation depends on two factors: H_r and P_{rd}. *The write amplification can be reduced by either increasing the cache hit ratio and/or lowering the probability of replacing a dirty entry in the cache.*

3.2 Observations

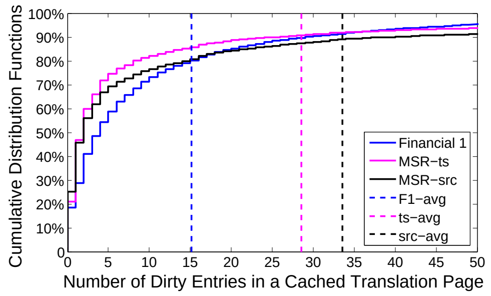
1) *Distribution of entries in the mapping cache*

In a demand-based page-level FTL, each translation page contains a fixed number of mapping entries. Therefore, there may be more than one entry that belong to the same translation page are cached. We have carried out experiments to study the distribution of cached entries in DFTL with four enterprise workloads. Since mapping entries in the mapping table are stored in the order of LPNs, their LPNs can be obtained by their offsets in the table. Thus, only the PPNs of mapping entries are stored in flash memory. In our experiments, a PPN takes 4B and the size of a flash page is 4KB so that each translation page contains 1024 mapping entries. The cache is set as large as the mapping table of a block-level FTL plus the GTD size, and more detailed experiment setup is presented in Section 5.1.

Figure 1(a) shows the average number of entries in each *cached translation page*, which refers to a translation page which has one or more entries in the cache, during the running phase. We can see an average of no more than 150 entries (no more than 90 entries most of the time) in each cached translation page are present in the mapping cache. *This result shows only a small fraction (less than 15%) of entries in a cached translation page are recently used.* Thus, it is not space-efficient to cache an entire translation page because most of the entries in the page will not be accessed in the near future. Figure 1(b) shows the cumulative distribution curves of the number of cached translation pages in respect to the number of dirty entries that a page contains under three write-dominant workloads. The vertical dashed lines represent the average numbers of cached dirty entries across cached translation pages. *We can see that 53%-71% of cached translation pages have more than one dirty entry cached, and the average numbers of dirty entries in each page are above 15.* Therefore, a cached translation page will be updated repeatedly, when the dirty entries in the page are successively evicted. This drawback is due to the inefficiency of the replacement policy of DFTL, which writes back only one dirty entry when evicting a dirty entry, regardless of the other more than 14 dirty entries, on average, that share the same translation page with the evicted one. Since most of the dirty entries in a page remain cached after evicting one, the probability of replacing a dirty entry is still high on the next eviction.

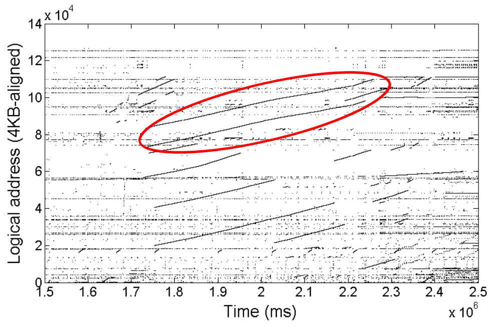


(a) Average number of entries in each cached translation page.

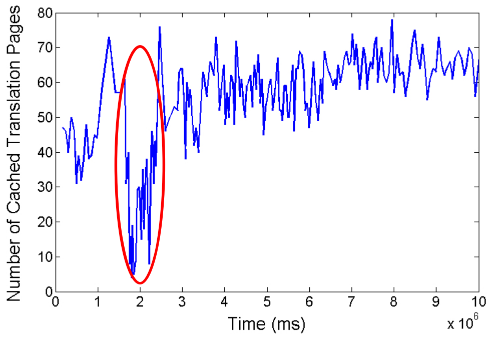


(b) CDFs of number of cached translation pages.

Figure 1: Distribution of entries in the mapping cache. Numbers were collected by sampling the mapping cache every 10,000 user page accesses during the entire running phase, which contains millions of user page accesses.



(a) Access Distribution of Financial1.



(b) Trends in the mapping cache.

Figure 2: Spatial locality analyses of Financial1 trace.

2) ***Spatial locality in workloads***

Spatial locality, which refers to the access pattern in which when a page is accessed, its logically neighbouring pages are likely to be accessed soon, is an important feature of enterprise workloads. We have studied the spatial locality in several representative enterprise workloads. The results reveal that most workloads exhibit some degree of spatial locality. As shown in Figure 2(a), each 4KB-aligned request corresponds to a dot. Although Financial1 is a random-dominant workload, it is evident that sequential accesses, denoted by the diagonal lines, are very common.

Sequential accesses have a different impact on the distribution of cached mapping entries from random accesses. Figure 2(b) shows how the number of cached translation pages in DFTL is changing over time in Financial1 workload. The ovals in Figures 2(a) and 2(b) correspond to the same time period. *We see that sequential accesses make the number of cached translation pages first decline sharply and then rise back.* The decline is because sequential accesses require consecutive mapping entries, which concentrate on a few translation pages. When they are loaded into the cache, sparsely dispersed entries in many cached translation pages are evicted to make room. As sequential accesses that have low temporal locality [24] come to an end, following random accesses load dispersed entries into the cache, replacing the consecutive entries. Thus, the number of cached translation pages increases. During the whole process, a large amount of loadings and replacements occur and the translation pages requested by sequential accesses are accessed repeatedly in flash memory. Similar phenomena are also observed in other workloads, providing a clue to leverage the sequential accesses in workloads to improve the cache hit ratio.

3.3 Motivation

Due to the employment of the MLC (Multi-Level Cell) technology, the price of flash-based SSDs has dropped dramatically and the capacity has increased significantly. MLC flash has a longer write time and a weaker endurance than SLC flash, making write operations more expensive in MLC flash [13, 17]. However, extra operations, including random writes, are introduced when demand-based page-level FTLs leverage a small mapping cache to accelerate the address translation. Our experiments (the configuration is shown in Section 5.1) show that the extra operations lead to an average of 58.4% performance loss and 42.3% block erasure increase in the four workloads, as shown in Table 2. Therefore, it is critical to reduce the extra operations caused by address translation for the sake of improving both the performance and lifetime of an SSD.

On the other hand, existing FTL schemes suffer from different problems. DFTL does not exploit the spatial locality, losing the chance to further improve the cache hit ratio, and its replacement policy cannot efficiently lower the probability of replacing a dirty entry. Thus, DFTL performs poorly in sequential workloads and write-dominant workloads. CDFTL and S-FTL keep translation pages cached to exploit the spatial locality, achieving high performance in sequential workloads. However, their hit ratio will decrease in random workloads, because it is not space-efficient to cache an entire page, as observed in Section 3.2. Furthermore, evicting an entire page increases the probability of replacing a dirty entry. As a result, CDFTL and S-FTL perform poorly in random workloads.

Table 2: Deviations of DFTL from the optimal FTL.

	Fin1	Fin2	ts	src
Performance	63.4%	52.6%	59.4%	58.2%
Erasure	45.9%	52.6%	30.4%	56.2%

These observations motivate us to design a novel demand-based page-level FTL that is able to perform fast address translation at the cost of minimal extra operations in flash memory with a small mapping cache. *The key insight has been proven to be improving the cache hit ratio as well as reducing the probability of replacing a dirty entry.*

4. Design of TPFTL

4.1 Overview

As shown in Figure 3, TPFTL adopts a demand-based page-level mapping, where flash memory is divided into two parts: data blocks, storing user data, and translation blocks, containing the mapping table which consists of LPN-PPN mapping entries. These entries are packed into translation pages in an ascending order of LPNs. Translation pages are also managed by the page-level mapping, where a *virtual translation page number* (VTPN) can be mapped to any *physical translation page number* (PTPN). The *global translation directory* (GTD), which is small and entirely resident in the mapping cache, maintains the physical locations (VTPN-PTPNs) of translation pages. The requested VTPN is the quotient of the requested LPN and the number of mapping entries each translation page contains. Given the VTPN, querying the GTD, we can locate the translation page (PTPN) in flash memory which stores the requested LPN-PPN mapping entry.

With limited cache space, TPFTL only holds a small set of popular mapping entries in the cache. Considering that the access unit of the mapping table in flash memory is a page, TPFTL clusters the cached entries of each cached translation page in the form of a *translation page node* (TP node). All the TP nodes in the cache are managed in an LRU list, called the *page-level LRU*. Each TP node has a VTPN and maintains an entry-level LRU list, which consists of its cached entries in the form of *entry nodes*. Each entry node records an LPN-PPN entry. Both the loading unit and replacing unit of TPFTL are a mapping

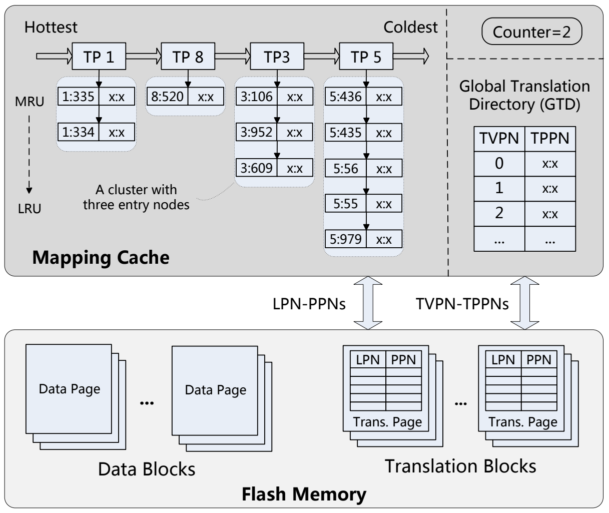


Figure 3: Architecture of TPFTL. Each page number is denoted by two-part: flash block number : page offset in the block (e.g. LPN 1359 is written as 1:335). As PPNs and PTPNs are not important here, they are represented by 'x:x'.

entry. In addition, a counter is maintained in the cache to record the number change of TP nodes, which is necessary to our loading policy. Since a few bits are enough, this space cost is negligible.

Employing the two-level LRU lists to organize cached mapping entries has three advantages. First, the distribution of cached entries over the entire mapping table is presented, providing a hint to design an efficient loading policy and replacement policy. Second, cache space utilization increases. Although extra TP nodes take up some cache space, they account for only a small percentage. More than that, each mapping entry can be stored in a compressed way. As the mapping entries in the translation pages are stored in order, the LPN of each mapping entry can be obtained from the VTPN and the offset of the entry inside the translation page. Hence, the offset is stored instead of the LPN for each entry. For example, a complete mapping entry consists of a 4 bytes LPN and a 4 bytes PPN. While the offset takes 10 bits of storage space, assuming each translation page holds 1024 entries, 6 bytes of storage space are enough for an entry in the mapping cache. Thus, more mapping entries can be cached, which has been verified by our experiments. Third, the two-level index can accelerate the searching speed compared to a one-level index.

4.2 Page-level LRU

As observed in Section 3.2, a TP node usually has more than one entry node with different hotness. In the page-level LRU list, the TP node that contains the hottest or most recently used (MRU) entry node may also contain cold entry nodes, which should be evicted ahead of the hot one. To achieve this, we define *page-level hotness* as the average hotness of all the entry nodes in a TP node to measure the hotness of the TP node. The position of each TP node in the page-level LRU list is decided by its page-level hotness.

When serving a request for an LPN-PPN entry, supposing no prefetching is performed (prefetching is discussed in Section 4.3), TPFTL first searches the page-level LRU list. If the TP node to which the requested entry belongs is cached, its entry nodes will be checked. If the entry node that shares the same LPN is found, a cache hit happens and the requested entry can be directly obtained in the cache. Otherwise, a cache miss occurs and the requested entry has to be fetched from flash memory in the following steps. First, the requested translation page is located and read out. Then, the requested entry is loaded into the MRU position of the entry-level LRU list of its TP node, which will be created first if not existing. During this process, if the cache is full, a replacement will occur. TPFTL first selects the coldest (LRU) TP node and then chooses its LRU entry node as a victim. If the last entry node of a TP node is evicted, the TP node will be removed from the cache. Note that the hotness of each entry node is obscured by the page-level hotness, which results in less efficiency in exploiting the temporal locality. The cache hit ratio is slightly improved due to increased cache space utilization, which has been verified by our experiments.

4.3 Loading Policy

As depicted in Section 3.2, sequential accesses are very common in workloads. If they are not recognized and utilized, significant access overhead will be incurred in flash memory. To alleviate this problem, TPFTL proposes two prefetching techniques to improve the cache hit ratio.

The first technique is *request-level prefetching*. When a request arrives at the FTL, it is split into one or more page accesses according to its start address and length. For a large request, several sequential page accesses to flash memory are generated. Traditional FTLs carry out address translation on each page access, which may result in multiple cache misses in one request. Instead, TPFTL performs address translation on the entire request so that one request causes one cache miss at most. To be specific, if the first page access split from a request is not in the mapping cache, TPFTL loads all the mapping entries required by the request instead of just the entry of the first page access. Therefore, the length of request-level prefetching is proportional to the number of page accesses contained in the original request.

In real workloads, sequential accesses are often interspersed with random accesses so that small requests may also be part of sequential accesses [25]. The request-level prefetching is efficient with large requests, but is powerless to recognize those small requests. Therefore, we propose the second technique, *selective prefetching*. We base selective prefetching on the important observation in Section 3.2 that the number of TP nodes decreases when sequential accesses are happening and increases when they are over. If the number of TP nodes continues to decrease by a threshold, TPFTL assumes sequential accesses are happening and performs selective prefetching when a cache miss occurs. If the number begins to continuously increase by the threshold, TPFTL assumes sequential accesses are over and stops selective prefetching. The number change is recorded by the counter, which increases by 1 when a new TP node is loaded into the cache and decreases by 1 when a cached TP node is evicted from the cache. When the absolute value of the counter reaches the threshold, the selective prefetching is deactivated if the counter is positive; or activated if the counter is negative, and then the counter is reset to 0. In our experiments, we empirically found that most sequential accesses in workloads can be well recognized when we set the threshold as 3. Further, TPFTL sets the length of selective prefetching as the number of cached predecessors, which are consecutive to the requested mapping entry in LPNs and share the same translation page with the requested one. This length is chosen because the longer a sequential access stream is, the more likely the subsequent data is going to be accessed in the near future [27]. As a whole, selective prefetching with dynamic length enables TPFTL to be adaptive to various workloads and achieve a high cache hit ratio most of the time.

An example of the selective prefetching is given in Figure 4. Initially, suppose the mapping cache is full, and the counter is equal to -3 so that the selective prefetching is activated and the counter is reset to 0. It takes seven steps to complete the address translation of LPN 1:336: (1) LPN 1:336 is not in the cache and its TP node ($VTPN = 1$) has been cached; (2) Two consecutive predecessors of LPN 1:336, LPN 1:334 and LPN 1:335, are found to be cached so that the prefetching length is 2; (3) Since the cache is full, the LRU TP node 3 as well as its entry, LPN 3:27, are evicted ($Counter = -1$), leaving two free slots. Then the prefetching length is reduced to 1 according to the principles described in Section 4.5. That is, only LPN 1:336 and LPN 1:337 will be loaded; (4) Read out the translation page ($VTPN = 1$) from flash memory, whose physical location is obtained by consulting the GTD; (5)-(6) Load the requested entry (LPN 1:336) and the prefetched entry (LPN 1:337), into TP node 1 in the cache; (7) The PPN of LPN 1:336 can be obtained in the cache.

4.4 Replacement Policy

Replacement policy plays a critical role in cache management because its inefficiency could lead to excessive writes. Although prefetching improves the hit ratio, it requires multiple victims, increasing the probability of replacing a dirty entry. To minimize the probability without sacrificing the hit ratio, TPFTL adopts two replacement techniques.

The first technique is *batch-update replacement*. To the best of our knowledge, batch update was first used in DFTL, but in a very limited way. In a GC operation,

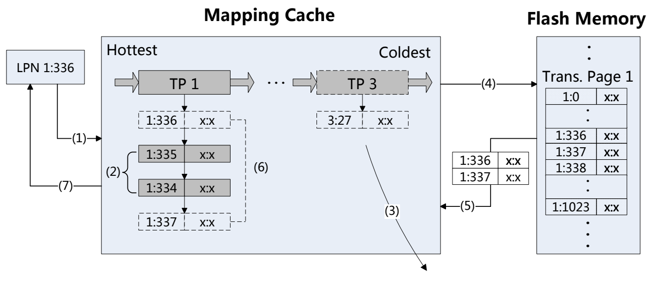


Figure 4: An example of selective prefetching. As PPNs are irrelevant here, they are represented by ‘x:x’.

multiple valid data pages in the victim block may have their mapping entries present in the same translation page, so DFTL combines these modifications into a single batch update. An analog is the batch eviction of ZFTL [34], which evicts all entries in a small reserved area that share the same translation page with the victim entry. However, thanks to the two-level lists, TPFTL can extend the batch-update technique throughout the whole mapping cache and on every writeback of a dirty entry. As observed in Section 3.2, there are more than one dirty entry in most TP nodes. When a dirty entry node becomes a victim, TPFTL writes back all the dirty entry nodes of its TP node, and only the victim is evicted while the others remain in the mapping cache with clean state. In addition, when a GC miss occurs and the mapping entry of a migrated data page needs to be updated in flash memory, TPFTL checks if the translation page that it belongs to is cached. If cached, all the cached dirty entries in the page are written back in a batch and become clean. Therefore, multiple dirty entries can be updated in each translation page update and the hit ratio will not decrease. As the number of cached dirty entries decreases rapidly, batch-update replacement can significantly lower the probability of replacing a dirty entry without increasing replacement overhead.

The second technique is *clean-first replacement*. Previous work [38] reveals that in the data buffer of an SSD, choosing a clean page as a victim rather than dirty pages can reduce the number of writes in flash memory. Based on this conclusion, TPFTL first selects the LRU TP node and then chooses its LRU clean entry node as a victim, when a victim is needed. If no clean entry node is found, the LRU dirty entry node will be the victim. As clean entries are more likely to be replaced, the probability of replacing a dirty entry is further reduced.

An example of the replacement policy is given in Figure 5. It includes four steps: (1) suppose the mapping cache is full, and three mapping entries, LPN 0:1000, LPN 0:1001 and LPN 0:1002, are going to be loaded into the cache; (2) the LRU TP node is VTPN 5, where evictions occur. Both the LRU clean entries (LPN 5:979 and LPN 5:435) and the LRU dirty entry (LPN 5:55) are chosen to be victims; (3)

all the dirty entries in VTPN 5 (LPN 5:55, LPN 5:56 and LPN 5:436) are updated to flash memory in a batch; (4) the requested entries are loaded into the cache.

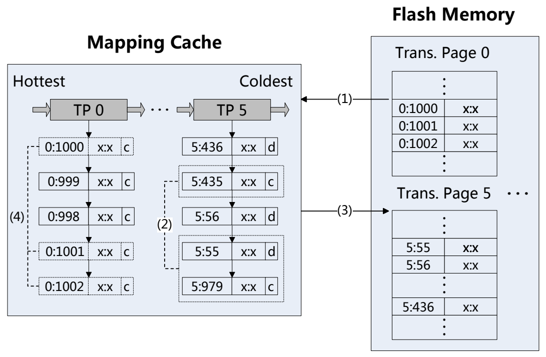


Figure 5: An example of the replacement policy. Each entry contains three fields: an LPN, a PPN and a dirty flag ('d' denotes that the entry is dirty while 'c' is clean). As PPNs are irrelevant here, they are represented by 'x:x'.

4.5 Integration of Prefetching with Replacement

Since the prefetching compels multiple mapping entries to be loaded or replaced, more than one translation page may be read or updated in the address translation of a page request. Thus, the serving time of a page request becomes unpredictable and the overhead may increase. To avoid these drawbacks, TPFTL limits the prefetching length by two rules. First, the prefetching should be limited in one translation page. If the prefetching length is larger than the number of entries that locates between the requested entry and the last entry in the translation page, it should be reduced. This prefetching with page boundary is similar to the limitation of CPU hardware prefetching [8], without which a page fault may occur. Second, the replacement should happen only in one cached translation page. If the prefetching length is larger than the number of entry nodes in the LRU TP node, the prefetching length should be reduced. These two rules ensure that no more than one translation page read or update occurs during each address translation, leading to a good balance between the loading and replacement of the mapping cache.

5. Evaluation

5.1 Experiment Setup

1) ***Trace-driven Simulator***

We use the trace-driven simulation to evaluate TPFTL and compare it with DFTL, S-FTL and the optimal FTL. Although CDFTL is relevant, it performs worse than S-FTL in our experiments and thus is not included in the following evaluation due to space limit. The optimal FTL, employing a page-level mapping with the entire mapping table cached, has minimal overhead that any FTL can possibly have. It is used to show what overhead address translation can incur. We set DFTL as the baseline to show how many performance and lifetime improvements TPFTL is able to achieve. The simulator is obtained by adding TPFTL, CDFTL and S-FTL modules into the Flashsim platform [14]. SSD parameters in our simulation are taken from [2] and listed in Table 3.

Traces have different sizes of logical address space. In order not to distort the original characteristics of workloads, we set the SSD as large as the logical address space of the trace. The mapping cache is set as large as the mapping table of a block-level FTL plus the GTD size, which is in proportion to the SSD capacity. Specifically, the capacities of the SSD and cache are 512MB and 8.5KB (8KB+512B), respectively, in Financial workloads, and 16GB and 272KB (256KB+16KB), respectively, in MSR workloads.

Table 3: SSD parameters in our simulation [2].

Flash Page Size	4KB
Flash Block Size	256KB
Page Read Latency	25us
Page Write Latency	200us
Block Erase Latency	1.5ms
Over-provision Space	15%

2) ***Workload Traces***

In the evaluation, we choose four enterprise traces to study the efficiency of different FTLs. Financial1 and Financial2 were collected from an OLTP application running at a large financial institution [1]. As random-dominant workloads, Financial1 is write intensive and Financial2 is read intensive. MSR-ts and MSR-src are write-dominant traces, collected on servers at Microsoft Research Cambridge (MSR) [32]. Different from Financial traces, their requests have larger sizes and stronger sequentiality. Table 4 presents the features of the workloads.

Table 4: Specification of workloads.

Parameters	Financial1	Financial2	MSR-ts	MSR-src
Write Ratio	77.9%	18%	82.4%	88.7%
Avg. Req. Size	3.5KB	2.4KB	9KB	7.2KB
Seq. Read	1.5%	0.8%	47.2%	22.6%
Seq. Write	1.8%	0.5%	6%	7.1%
Address Space	512MB	512MB	16GB	16GB

3) ***Evaluation Metrics***

As analyzed in Section 3.1, the *probability of replacing a dirty entry* and *cache hit ratio* are the two key factors that decide how much address translation impacts the performance and lifetime of an SSD. Hence, they are first evaluated to verify the efficiency of TPFTL. Then the *numbers of translation page reads and writes* are evaluated to show how TPFTL reduces the extra operations caused by address translation. For performance estimation, the *average system response time* is a good measure of the overall performance of the FTL, where the queuing delay, address translation and GC are all in consideration. For lifetime estimation, the *write amplification* and *block erase count* are evaluated to present more details.

5.2 Experiment Results

1) ***Probability of Replacing a Dirty Entry and Hit Ratios***

Figure 6(a) shows the probabilities of replacing a dirty entry, which refers to the ratio of the number of dirty entry replacements to the total number of mapping entry replacements during the entire running phase, of different FTLs under the four enterprise workloads. The probabilities of TPFTL are less than 4% in all workloads, most close to zero probabilities of the optimal FTL. Among these workloads, only Financial2 is read-dominant, so it has the lowest probabilities for all FTLs, among which TPFTL still achieves the lowest probability. As for the other write-dominant workloads, TPFTL reduces the probabilities by a range of 58% to 88.4%, compared to DFTL and S-FTL. This result demonstrates the efficiency of the replacement policy of TPFTL, which can reduce the number of cached dirty entries in a batch. Note that the replacement unit of S-FTL is an entire translation page instead of an entry, increasing the probability of replacing a dirty entry. The reason why S-FTL has lower probabilities than DFTL in Financial workloads is because S-FTL has a small dirty buffer to delay the replacements of sparsely dispersed dirty entries, which is efficient for random workloads. However, the dirty buffer works poorly for sequential workloads so that S-FTL has higher probabilities than DFTL in MSR workloads.

Figure 6(b) shows the cache hit ratios, including both reads and writes, of different FTLs under the four workloads. For Financial workloads with strong temporal locality and weak spatial locality, TPFTL improves the hit ratios by an average of 14.8% and 15.5%, respectively, compared to DFTL and S-FTL. For MSR workloads with strong temporal locality and spatial locality, TPFTL achieves higher hit ratios than DFTL by an average of 16.4% and comparable hit ratios as S-FTL, larger than 95%. We can conclude that TPFTL succeeds in maintaining a relatively high hit ratio in various workloads. This owes to the two-level LRU lists and the workload-adaptive loading policy of TPFTL. The former makes the cache more space-efficient and ensures that the temporal locality is exploited at the entry-level. The latter performs flexible prefetching to exploit the spatial locality. By contrast, DFTL only exploits the temporal locality, while S-FTL radically exploits the spatial locality by caching compressed translation pages and exploits the temporal locality in the page level. As a result, S-FTL matches DFTL in Financial workloads but outperforms DFTL in MSR workloads.

2) ***Numbers of Translation Page Reads and Writes***

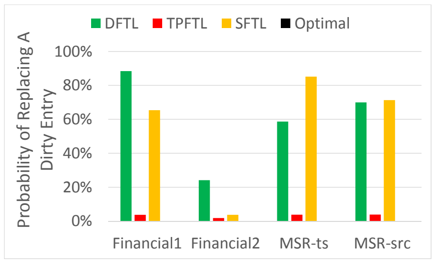
Figure 6(c) and 6(d) show the normalized numbers of translation page reads and writes, respectively, during both the address translation phase and GC operations of different FTLs under the four workloads. Each value is normalized to that of DFTL and a value 1 means they are equal. Note that a higher hit ratio leads to fewer translation page reads required by cache loadings, and a lower probability of replacing a dirty entry leads to fewer translation page reads and writes required by writing back dirty entries. It is no surprise to see that TPFTL reduces the numbers of translation page reads by an average of 44.2% and 34.9% for Financial workloads, and an average of 87.7% and 13.3% for MSR workloads, and reduces the numbers of translation page writes by an average of 50.5% and 31.4% for Financial workloads, and an average of 98.8% and 92.6% for MSR workloads, respectively, compared to DFTL and S-FTL. The reason why the reductions of translation page reads of TPFTL compared to S-FTL is less than those of translation page writes is because the replacement unit of S-FTL is a full page so that it eliminates the reads required for writing back dirty entries, which incurs partial page overwrites for TPFTL.

3) ***System Response Time***

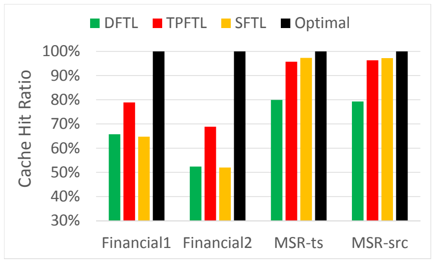
Figure 6(e) shows the normalized average system response times of different FTLs under the four workloads. Each value is normalized to that of DFTL. Although TPFTL has reduced both translation page reads and writes to flash memory, the reductions on response times are not as dramatic as those on the formers. This is because GC operations of data blocks account for a considerable proportion of the response time and TPFTL uses a similar GC mechanism to those FTL counterparts. Since TPFTL primarily targets random write intensive applications, it is not surprising to see TPFTL achieves the greatest advantages in Financial1 workload with 23.5% and 24.1% performance improvements, respectively, compared to DFTL and S-FTL. For Financial2 workload, the improvements are 20.9% and 11.7%, respectively. For MSR workloads, TPFTL matches S-FTL as well as the optimal FTL, and outperforms DFTL by an average of 57.6%. It looks strange that S-FTL has a little higher response time than DFTL in Financial1 workload in spite of a comparable hit ratio and a lower probability of replacing a dirty entry. This is related to the GC efficiency. S-FTL delays the evictions of sparsely dispersed dirty entries so that invalid pages are scattered among more blocks than DFTL, resulting in less GC efficiency.

4) ***Lifetime Analyses***

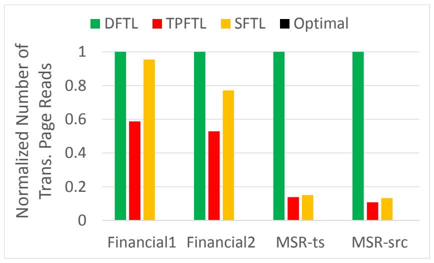
Figures 6(f) and 7(a) show the details of the lifetime evaluation of different FTLs under the four workloads. Reductions of write amplifications are not so dramatic



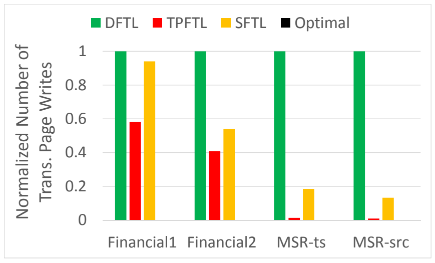
(a) Probability of replacing a dirty entry



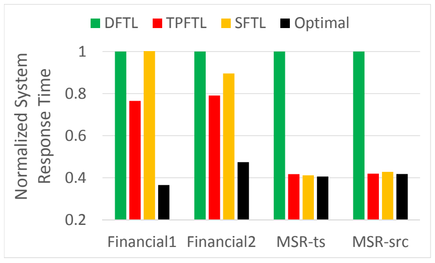
(b) Cache hit ratio



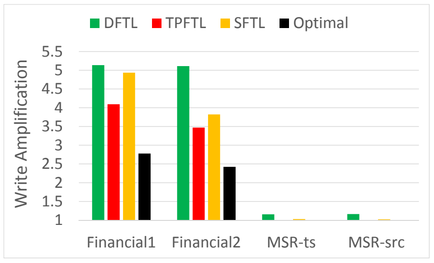
(c) Number of trans. page reads



(d) Number of trans. page writes

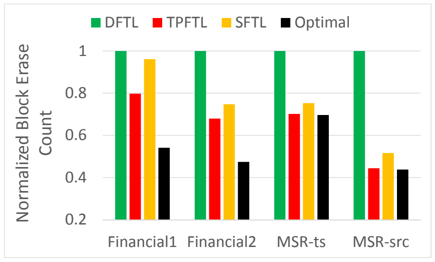


(e) System response time

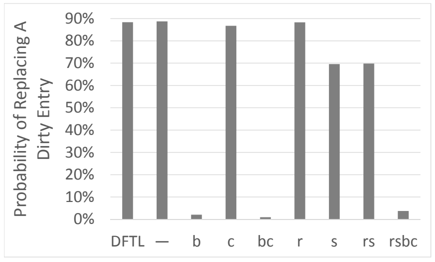


(f) Write amplification

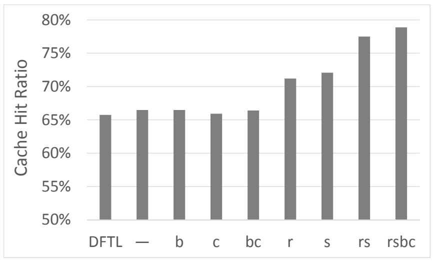
Figure 6: (a) shows the probabilities of replacing a dirty entry in the mapping cache; (b) shows the hit ratios of the mapping cache; (c) and (d) show the numbers of translation page reads and writes in flash memory, respectively; (e) shows the system response times; (f) shows the overall write amplifications.



(a) Block erase count

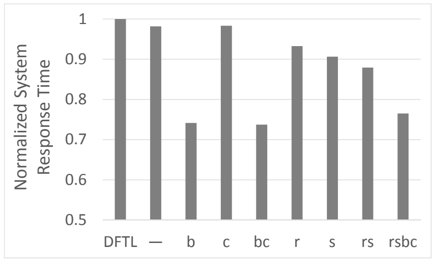


(b) Probability of replacing a dirty entry

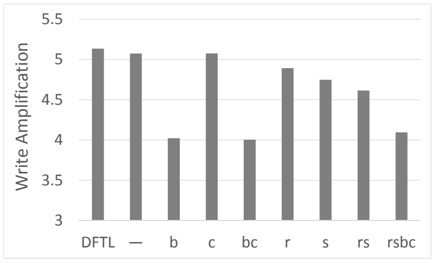


(c) Cache hit ratio

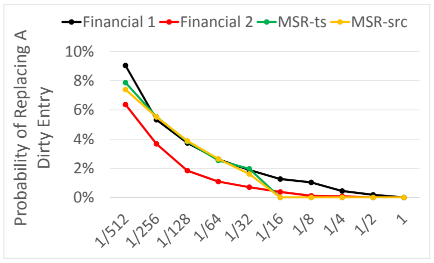
Figure 7: (a) shows the block erase count; (b) and (c) show the benefits of each TPFTL configuration on the probability of replacing a dirty entry and cache hit ratio in Financial1 workload. In (b) and (c), 'b', 'c', 'r', and 's' mean that the request-level prefetching, selective prefetching, batch-update replacement and clean-first replacement are turned on, respectively. Specially, '–' refers to the TPFTL variant without any prefetching or replacement optimization, and 'rsbc' refers to the complete TPFTL.



(a) System response time

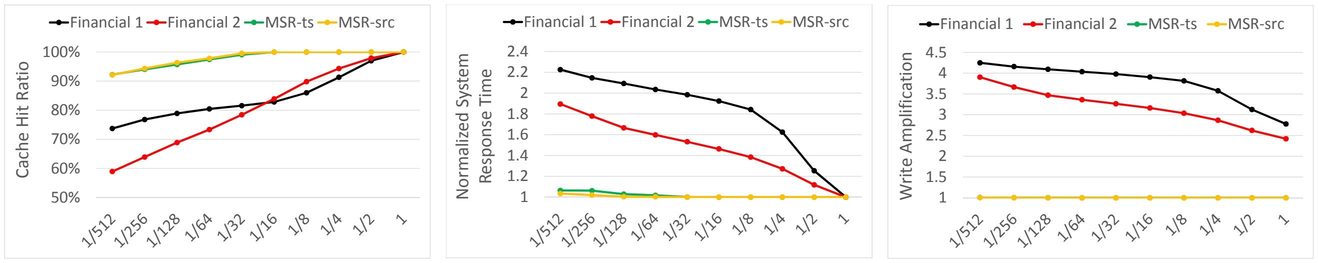


(b) Write amplification



(c) Impact of cache sizes on write amplification

Figure 8: (a) and (b) show the benefits of each TPFTL configuration on system response time and write amplification in Financial1 workload; (c) shows the impact of cache sizes on the probability of replacing a dirty entry of TPFTL. In (c), the X-axis represents the cache sizes, which are normalized to the size of the entire mapping table (each entry takes 8B).



(a) Probability of replacing a dirty entry (b) Cache hit ratio (c) System response time

Figure 9: Impact of cache sizes on the cache hit ratio, system repsonse time, and write amplification of TPFTL.

because data page writes account for a great amount of total page writes, as it is shown in Figure 6(f). For MSR workloads, only a few translation page updates are incurred during address translation because of high hit ratios, and almost no migrations of valid pages have occurred during GC operations because most writes are sequential. Thus, write amplifications in MSR workloads are close to 1, namely, very few extra writes are introduced. For Financial workloads, the write amplifications of FTLs range from 2.4 to 5.1 because of frequent translation page updates and massive page migrations. Nonetheless, TPFTL reduces the write amplifications by an average of 26.2% and 13.1% in Financial workloads, and an average of 13.8% and 2.3% in MSR workloads, respectively, compared to DFTL and S-FTL. Accordingly, Figure 7(a) shows that the block erase counts of TPFTL also decrease by an average of 34.5% and 11.8%, up to 55.6% and 17.1%, respectively, compared to DFTL and S-FTL. Therefore, TPFTL achieves an improvement on the lifetime of an SSD.

5) *Benefits of Each Technique in TPFTL*

To give further insight into the four techniques employed by TPFTL, we take Financial1 workload as an example to investigate the benefits of each technique. Specifically, we evaluate eight typical TPFTL configurations with the techniques turn on and off. Each TPFTL configuration is denoted by a monogram of enabled techniques, where 'r', 's', 'b' and 'c' mean that the request-level prefetching, selective prefetching, batch-update replacement and clean-first replacement are turned on, respectively. Specially, '–' refers to the TPFTL variant employing two-level LRU lists without any prefetching or replacement optimization, and 'rsbc' refers to the complete TPFTL. Figures 7(b), 7(c), 8(a) and 8(b) show the probabilities of replacing a dirty entry, cache hit ratios, system response times, and write amplifications of each TPFTL configuration in Financial1 workload, respectively.

In Figure 7(b), it is clear that compared to '–', the batch-update replacement ('b') significantly reduces the probability of replacing a dirty entry, while the clean-first replacement ('c') only achieves a little decrease due to rare clean entries in Financial1 workload. However, the clean-first replacement further reduces the probability by 54.3% in 'bc', compared to 'b'. This is because the batch-update replacement can write back more dirty entries on each eviction of a dirty entry when they are postponed to be replaced, indicating that *the clean-first replacement is an effective complement to the batch-update replacement*. Considering the prefetching techniques, 'rsbc' has a higher probability than 'bc', because the prefetching increases the probability as more entries are replaced each time.

In Figure 7(c), compared to '–', the request-level prefetching ('r'), the selective prefetching ('s'), and their alliance ('rs') increase the hit ratio by 4.7%, 5.6% and 11%, respectively. This reveals that *the request-level prefetching, which leverages the sequentiality in each large request, and selective prefetching, which leverages the sequentiality among requests, supplement each other well*. In addition, '–' achieves a little higher hit ratio than DFTL, verifying that the page-level LRU does not degrade the hit ratio. We can also conclude that the replacement techniques have little effect on the hit ratio.

Figure 8(a) shows that compared to '–', the replacement techniques ('bc') and the prefetching techniques ('rs') reduce the system response time by 24.9% and 10.4%, respectively. Moreover, they reduce the write amplification by 21.1% and 9.1%, respectively, according to Figure 8(b). These results manifest the efficiency of the replacement policy and loading policy of TPFTL. Note that 'bc' outperforms 'rsbc' in Financial1 workload, although 'rsbc' has a higher hit ratio. This indicates that the probability of replacing a dirty entry may play a more important role in deciding the performance and write amplification of an SSD than the hit ratio, most probably in workloads dominated by random writes.

6) *Impact of Cache Sizes*

We have shown the experiment results about the performance and write amplification of TPFTL with a mapping cache of the same size. Then, we would like to see the impact of cache sizes on TPFTL. Figures 8(c) and 9 show the probabilities of replacing a dirty entry, cache hit ratios, normalized system response times and write amplifications of TPFTL with varying cache sizes. Cache sizes are normalized to the size of the entire page-level mapping table. Specially, '1/128' refers to the cache size that we used in the preceding experiments, and '1' refers to the cache size that accommodates the entire mapping table.

As shown in Figure 8(c), the probabilities of replacing a dirty entry of TPFTL decrease with increasing cache capacity and reach 0% when the mapping table is entirely cached. This is because the average number of entry nodes in each TP node gets bigger with a larger cache, and each writeback makes more dirty entries to be clean, which helps to postpone the evictions of dirty entries in turn. As shown in Figure 9(a), the hit ratios of TPFTL increase with increasing cache capacity as more mapping entries can be held, and reach 100% when the mapping table is entirely cached. The relatively low hit ratios for Financial traces are due to their large working sets.

Figures 9(b) and 9(c) show that both the response times and the write amplifications of TPFTL decrease with increasing cache capacity. The response time is normalized to that of TPFTL with '1' cache size. A larger cache helps little in MSR workloads for two reasons. First, TPFTL almost eliminates the address translation overhead by achieving high hit ratios, higher than 92%, as well as low probabilities of replacing a dirty entry, less than 8%. Second, the strong sequentiality in MSR workloads leads to few migrations during GC operations. On the other hand, a larger cache can further improve the performance and lifetime so as to be rewarding in Financial workloads. This is because not only random accesses limit the improvement of the hit ratio, but also random writes introduce significant GC overhead. Although FTLs usually find themselves at an unfavourable position in cases of random write intensive workloads, TPFTL steps forward by reducing extra writes caused by address translation.

7) ***Space Utilization of the Mapping Cache***

Employing two-level LRU lists provides an opportunity to compress the storage space of each mapping entry in the cache. Thus, TPFTL can hold more entries than DFTL in the cache of the same capacity. Figure 10 shows the improvements of the cache space utilization of TPFTL with different cache sizes, compared to DFTL. We see that larger improvements, up to 33%, are obtained with a larger cache. This is because a larger cache enables more entries to be cached in the compressed form. The reason of the 33% limit is that the size of each cached entry is compressed from 8B to 6B. Since the sequentiality of requests gathers lots of entry nodes in a few TP nodes, the improvements in MSR workloads are greater than those in Financial workloads. A higher space utilization implies a higher hit ratio and a lower probability of replacing a dirty entry.

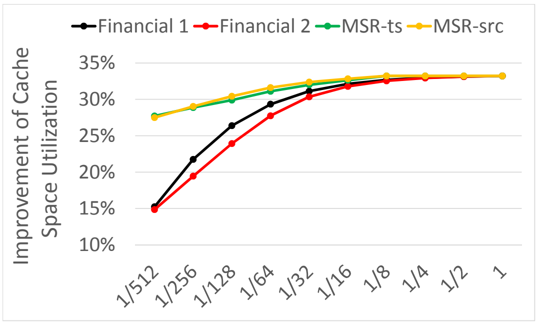


Figure 10: Improvement of cache space utilization.

6. Conclusion

This paper develops a performance model and a write amplification model for a demand-based page-level FTL to analyze the negative impact of extra operations caused by address translation on the performance and lifetime of an SSD, respectively. Two observations are noted based on the model analysis. First, the extra operations degrade both the performance and lifetime. Second, both a high hit ratio and a low probability of replacing a dirty entry of the mapping cache play a crucial role in reducing the system response time as well as the overall write amplification. Then, we propose a new demand-based page-level FTL, called TPFTL. Considering the different access units between flash memory and the mapping cache, TPFTL employs two-level LRU lists to organize the mapping entries in the cache. Further, a workload-adaptive loading policy is used to jointly exploit the temporal locality and spatial locality in workloads to improve the cache hit ratio, and an efficient replacement policy is utilized to minimize the probability of replacing a dirty entry. Extensive evaluations with enterprise workloads show that TPFTL can efficiently leverage a small cache space to perform fast address translation with low overhead and thus to improve the performance and lifetime of an SSD.

Acknowledgement

We are grateful to our shepherd Michael Swift and the anonymous reviewers for their insightful feedback. This work was sponsored in part by the National Basic Research Program of China (973 Program) under Grant No. 2011CB302303 and the National Natural Science Foundation of China No. 61300047, No. 61370063, and No. 61472152. The work conducted at VCU was partially sponsored by U.S. National Science Foundation under Grants CNS-1320349 and CNS-1218960. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] Traces from UMass Trace Repository. `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of USENIX Annual Technical Conference*, pages 57–70, 2008.

[3] R. Bryant, O. David Richard, and O. David Richard. *Computer systems: a programmer's perspective*, volume 2. Prentice Hall Upper Saddle River, 2003.

[4] E. Budilovsky, S. Toledo, and A. Zuck. Prototyping a high-performance low-cost solid-state disk. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, 2011.

[5] F. Chen, T. Luo, and X. Zhang. CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST)*, pages 77–90, 2011.

[6] T. Chung, D. Park, S. Park, D. Lee, S. Lee, and H. Song. System software for flash memory: a survey. In *Proceedings of Embedded and Ubiquitous Computing*, pages 394–404. Springer, 2006.

[7] B. Debnath, S. Krishnan, W. Xiao, D. J. Lilja, and D. Du. Sampling-based garbage collection metadata management scheme for flash-based storage. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2011.

[8] U. Drepper. What every programmer should know about memory. `http://lwn.net/Articles/252125/`.

[9] Fusion-IO. Fusion-IO drive datasheet. `www.fusionio.com/data-sheets/iodrive-data-sheet/`.

[10] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[11] G. Goodson and R. Iyer. Design tradeoffs in a flash translation layer. In *Proceedings of Workshop on the Use of Emerging Storage and Memory Technologies*, 2010.

[12] J. Gray and B. Fitzgerald. Flash disk opportunity for server applications. *Queue*, 6(4):18–23, 2008.

[13] L. M. Grupp, J. D. Davis, and S. Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–24, 2012.

[14] 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 International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 229–240, 2009.

[15] X. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR: The Israeli Experimental Systems Conference*, 2009.

[16] Y. Hu, H. Jiang, D. Feng, L. Tian, S. Zhang, J. Liu, W. Tong, Y. Qin, and L. Wang. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2010.

[17] P. Huang, G. Wu, X. He, and W. Xiao. An aggressive worn-out flash block management scheme to alleviate ssd performance degradation. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014.

[18] A. Jagmohan, M. Franceschini, and L. Lastras. Write amplification reduction in NAND flash through multi-write coding. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2010.

[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 *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2011.

[20] X. Jimenez, D. Novo, and P. Ienne. Wear unleveling: improving NAND flash lifetime by balancing page endurance. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 47–59, 2014.

[21] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[22] S. Ko, S. Jun, K. Kim, and Y. Ryu. Study on garbage collection schemes for flash-based linux swap system. In *Proceedings of Advanced Software Engineering and Its Applications (ASEA)*, pages 13–16, 2008.

[23] 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. *ACM Trans. Embed. Comput. Syst.*, 6(3), 2007.

[24] S. Lee, D. Shin, Y. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, 2008.

[25] C. Li, P. Shilane, F. Douglis, D. Sawyer, and H. Shim. Assert(!Defined(Sequential I/O)). In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[26] J. Li, K. Zhao, X. Zhang, J. Ma, M. Zhao, and T. Zhang. How much can data compressibility help to improve nand flash memory lifetime? In *Proceedings of 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 227–240, 2015.

[27] M. Li, E. Varki, S. Bhatia, and A. Merchant. TaP: table-based prefetching for storage caches. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[28] J. Liu, Y. Chai, X. Qin, and Y. Xiao. PLC-Cache: endurable SSD cache for deduplication-based primary storage. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2014.

[29] Y. Lu, J. Shu, and W. Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 257–270, 2013.

[30] Y. Lu, J. Shu, and W. Wang. ReconFS: a reconstructable file system on flash storage. In *Proceedings of the 12th USENIX*

Conference on File and Storage Technologies (FAST), pages 75–88, 2014.

[31] Micron. MLC 25nm Micron NAND Flash L74A 64Gb 128Gb 256Gb 512Gb Asynchronous/Synchronous NAND Flash Memory Data Sheet, 2009.

[32] Microsoft. MSR Cambridge Traces. `http://iotta.snia.org/traces/388`.

[33] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 139–154, 2012.

[34] W. Mingbang, Z. Youguang, and K. Wang. ZFTL: a zone-based flash translation layer with a two-tier selective caching mechanism. In *Proceedings of the 14th IEEE International Conference on Communication Technology (ICCT)*, 2011.

[35] S. Moon and A. N. Reddy. Write amplification due to ECC on flash memory or leave those bit errors alone. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2012.

[36] Y. Pan, G. Dong, and T. Zhang. Error rate-based wear-leveling for NAND flash memory at highly scaled technology nodes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(7):1350–1354, 2013.

[37] D. Park, B. Debnath, and D. Du. CFTL: a convertible flash translation layer adaptive to data access patterns. *ACM SIGMETRICS Performance Evaluation Review*, 38(1):365–366, 2010.

[38] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 234–241, 2006.

[39] Z. W. 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 *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 157–166, 2011.

[40] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. Lung, et al. Phase-change random access memory: a scalable technology. *IBM Journal of Research and Development*, 52 (4.5):465–479, 2008.

[41] Samsung. SSD 840 EVO 2.5" SATA III 1TB. `http://www.samsung.com/us/computer/memory-storage/MZ-7TE1T0BW`.

[42] K. Vättö. LSI Announces SandForce SF3700: SATA and PCIe in One Silicon. `http://www.anandtech.com/show/7520/`.

[43] C. Wang and W. Wong. ADAPT: efficient workload-sensitive flash management based on adaptation, prediction and aggregation. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2012.

[44] C. Wang and W. Wong. TreeFTL: efficient RAM management for high performance of NAND flash-based storage systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 374–379. EDA Consortium, 2013.

[45] Q. Wei, B. Gong, S. Pathak, B. Veeravalli, L. Zeng, and K. Okada. WAFTL: a workload adaptive flash translation layer with data partition. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2011.

[46] Q. Wei, C. Chen, and J. Yang. CBM: a cooperative buffer management for SSD. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2014.

[47] G. Wu and X. He. Delta-FTL: improving SSD lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*, pages 253–266, 2012.

[48] G. Wu and X. He. Reducing SSD read latency via NAND flash program and erase suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, pages 117–123, 2012.

[49] G. Wu, B. Eckart, and X. He. BPAC: an adaptive write buffer management scheme for flash-based solid state drives. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2010.

[50] J. Yang, N. Plasson, G. Gillis, N. Talagala, S. Sundararaman, and R. Wood. HEC: improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference*, 2013.

[51] Y. Yang and J. Zhu. Analytical modeling of garbage collection algorithms in hotness-aware flash-based solid state drives. In *Proceedings of Mass Storage Systems and Technologies (MSST)*, 2014.

[52] J. Zhang, M. Shihab, and M. Jung. Power, energy and thermal considerations in SSD-based I/O acceleration. In *Proceedings of the 6th USENIX conference on Hot Topics in Storage and File Systems*, 2014.

[53] 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 (FAST)*, pages 1–16, 2012.