

## FASTer FTL for Enterprise-Class Flash Memory SSDs

Sang-Phil Lim

*School of Info & Comm Engr  
Sungkyunkwan University  
Suwon, 440-746, Korea  
Email: lsfeel0204@skku.edu*

Sang-Won Lee

*School of Info & Comm Engr  
Sungkyunkwan University  
Suwon, 440-746, Korea  
Email: wonlee@ece.skku.ac.kr*

Bongki Moon

*Dept. of Computer Science  
University of Arizona  
Tucson, AZ 85721, U.S.A.  
Email: bkmoon@cs.arizona.edu*

**Abstract**—For the past decade, numerous methods have been proposed for the design of a flash translation layer (FTL), which is the core engine of flash memory drives that critically determines the performance of the drives. In this paper, we revisit one of the popular FTL schemes called FAST, and augment it with new optimization techniques aiming particularly at online transaction processing (OLTP) workloads. As flash memory solid state drives (SSDs) are increasingly adopted for large-scale enterprise-class storage systems, it is important to develop an FTL that can deal with OLTP workloads in a scalable manner, which are characterized by a large number of small, random and skewed IO operations. With the proposed optimization methods such as giving a second chance to valid pages and isolating cold ones, the enhanced FTL, called *FASTer*, outperforms FAST considerably. In our experiments, *FASTer* reduced average elapsed time by more than 30 percent, and minimized the fluctuation of response time drastically. Overall, the performance of *FASTer* was comparable to a page mapping FTL, which tends to consume much more DRAM space to store a large mapping table. This will make the *FASTer* FTL one of the most cost effective and scalable FTL schemes for OLTP workloads.

**Keywords**-Flash Translation Layer, Full Associativity, OLTP, Hot-cold Separation

### I. INTRODUCTION

As flash memory solid state drives (SSDs) with a large capacity become available in the market, storage systems for large scale enterprises are emerging as one of the most promising application domains for flash memory SSDs. In particular, flash memory SSDs are considered a serious replacement of magnetic disks drives in the on-line transaction processing (OLTP) market [11], [16]. To make flash memory SSDs more appealing to the OLTP systems, it is desirable to optimize the performance of SSDs for the access patterns common in the OLTP workloads.

In order for the SSDs to successfully support OLTP applications, it is critical to optimize the performance of SSDs, specifically, the performance of the firmware called

a flash translation layer (FTL), for random write operations. This is because small random reads and writes are dominant in the OLTP workloads and the write operation often becomes a bottleneck in the overall performance of SSDs. The recent trend in the industry is that enterprise class SSDs are equipped with more resources such as a large DRAM buffer and an over-provisioned capacity for high performance and throughput [16]. In this paper, we aim at developing a high performance FTL for enterprise class SSDs.

We propose a new FTL called *FASTer* optimized for OLTP applications such that it is scalable with the increasing capacity of contemporary SSDs without consuming too much resources. *FASTer*, like its predecessor FAST [17], is in principle based on block address mapping rather than page address mapping, so that the size of a mapping table does not grow too large and does not consume the DRAM buffer too much.

The FAST FTL, developed previously for embedded systems, has been known for its superior performance for random write operations. One of the key ideas of FAST is to rely on the full associativity between data blocks and log blocks in order to avoid log block thrashing problem and improve log block utilization. It can handle frequently overwritten pages well so they will not degrade the overall performance. Despite the strengths of FAST, however, it has a few limitations as well. When the amount of over-provisioned capacity is too small (less than 3% of data capacity), FAST does not take full advantage of temporal locality of page write requests, relies excessively on costly merge operations, and the response time of a page write may fluctuate much [9], [15].

These observations motivate our work on the *FASTer* FTL scheme targeted for scalable OLTP systems. To design the *FASTer* FTL, we have analyzed an OLTP workload to understand the relationship between the write patterns common in OLTP applications and core principles of an FTL based on block address mapping. We define such concepts as *write interval* and *log window*, and use them as the primary means to model the temporal locality of write requests with respect to the logging capacity of page updates by the *FASTer* FTL.

The *FASTer* FTL adopts a new hot-cold separation strategy

\* This work was supported in part by MKE, Korea under ITRC NIPA-2009-(C1090-0902-0046), the Korea Research Foundation Grant (KRF-2008-0641), and Seoul Metropolitan Government 'Seoul R&BD Program (PA090903)'. This work was also sponsored in part by the U.S. National Science Foundation Grant IIS-0848503. The authors assume all responsibility for the contents of the paper.

called a *second chance* that exploits the skewedness in the OLTP write requests. When a log block is chosen as a victim for reclamation, by carrying the valid pages from the log block over to a new log block, these pages are effectively given a second chance to be invalidated before being merged to their corresponding data blocks. This strategy prevents many valid pages from being merged from a log block to multiple data blocks, and improves the overall performance of page writes as much as 30 percent compared with the FAST FTL. In effect, adopting this second chance policy is equivalent to doubling the size of a log window without physically increasing the amount of over-provisioned capacity. We demonstrate that *FASTer* can identify hot and cold pages only by using an appropriate log space, and be more scalable for large capacity flash memory SSDs.

In addition, *FASTer* uses an *isolation area* to collect cold pages whose write intervals are longer than the ‘doubled-up’ log window. These pages are very cold ones that remain in the log area even after a second chance is given to them. If they were merged to their corresponding data pages, as the FAST FTL would do, it might cause the response time of a page write to fluctuate widely, because a merge operation might involve as many data blocks as the cold data pages to be merged from a log block. Instead, *FASTer* collects them in the isolation area and merges them progressively whenever a new write request arrives, so that the merge overhead is hidden and the variance in the response time of a page write is minimized.

The key contributions of this work are summarized as follows.

- We use write interval and log window to understand the temporal locality of write requests with respect to the logging capacity of *FASTer*. This approach provides a general framework that can be used to analyze the behaviors of other FTLs based on log blocks.
- *FASTer* adopts a hot-cold separation strategy called a second chance to exploit the skewedness in the OLTP workloads and avoid overly frequent merge operations. In effect, adopting the second chance policy is equivalent to doubling the size of a log window without increasing the over-provisioned capacity in an SSD physically.
- *FASTer* uses an isolation area to collect cold pages and merge them progressively to minimize the variance in the response time of a page write.

The rest of this paper is organized as follows. Section II describes background and related works. Section III analyzes the skewedness in OLTP workloads, and discusses its implications on FTLs, and introduces new concepts such as write interval and log window. Section IV presents the key optimization strategies for *FASTer*, namely, the second chance policy and the isolation area. Section V presents the performance results, and Section VI concludes the paper.

## II. BACKGROUND

### A. Flash Memory

There are two types of flash memory: NOR and NAND [19]. NOR flash memory is mainly used for program code storage while NAND flash memory is for large data storage. For this reason, we deal with only NAND flash memory in this paper. NAND flash memory, according to the block size, has two types: small-block NAND with 32KB block and large-block NAND with 128KB. Because the small-block NAND flash is not popular any more, we assume large-block NAND flash memory in this paper. The basic unit of read and write in large-block NAND flash memory is a page of 2KB, and one block consists of 64 pages. With flash memory, no data page can be updated in place without erasing a block of flash memory containing the page<sup>1</sup>. This characteristics of flash memory is called “erase-before-write” limitation. Depending on the cell type, NAND flash memory is further divided into single level chip (SLC) and multi level chip (MLC). Another limitation of flash memory is the number of erases allowed in each block, and the maximum number of erases per block is 100K in SLC, and this is further limited to 10K in MLC. Throughout this paper, we assume the performance characteristics of state-of-the-art large-block SLC NAND flash chip in Table I.

Read	Write	Erase
25 $\mu$ s (2KB)	200 $\mu$ s (2KB)	1,500 $\mu$ s (128KB)

Table I  
LATENCY OF LARGE-BLOCK SLC NAND FLASH ([19])

### B. Flash Translation Layer

In order to alleviate the “erase-before-write” problem in flash memory, most flash memory storage devices are equipped with a software or firmware layer called Flash Translation Layer (FTL) [10]. An FTL makes a flash memory storage device look like a hard disk drive to the upper layers. One key role of an FTL is to redirect each logical page write from the host to a clean flash memory page which has been erased, and to remap the logical page address from an old physical page to a new physical page. In addition to this address mapping, an FTL is also responsible for data consistency and uniform wear-leveling.

For the past decade, numerous FTLs have been proposed (for example, [4], [5], [8], [9], [10], [12], [13], [14], [15], [17]). By the granularity of address mapping, they can be largely classified into three types: page mapping FTLs, block mapping FTLs, and hybrid mapping FTLs. In a block mapping FTL, address mapping is done coarsely at the level of

<sup>1</sup>Though the unit of IO varies, we assume the 2KB page in this paper because the unit of IO in the database we get the trace is 2KB page.

blocks. Thus, the size of a mapping table tends to be small, but the address mapping is inherently inflexible because the page offset within a block must be identical for both logical and physical addresses. This mapping constraint often incurs significant overhead for page writes, because even a single page write requested for a certain data block may require another partially written data block to be reclaimed [6].

In a page mapping FTL, on the other hand, address mapping is finer-grained at the level of pages, which are much smaller than blocks. The process of address mapping is more flexible, because a logical page can be mapped to any physical page without the aforementioned page offset constraint [6], [8], [13]. When an update is requested for a logical page, the FTL stores the data in a clean page, and updates the mapping information for the page. When it runs out of clean pages, the FTL initiates a block reclamation in order to create a clean block. The block reclamation usually involves erasing an old block and relocating valid pages. Although the overhead of a page-level address mapping is lower than that of a block-level address mapping, a page mapping FTL requires a much larger memory space to store a mapping table. For example, a page mapping FTL consumes up to 64MB memory to store a mapping table for an SSD of 32GB capacity with 2KB pages. An on-demand caching can be applied to reduce the memory use for mapping, but it will in turn incur additional flash memory read and write operations for mapping addresses and updating the mapping information itself [9].

Hybrid mapping FTLs have been proposed to overcome the limitations of both page and block mapping FTLs, namely, the large memory requirement of page mapping and the inflexibility of block mapping [5], [17]. In a hybrid mapping FTL, flash memory blocks are logically partitioned to a data area and a replacement (or log) area. Blocks in a data area are managed by block-level mapping, while blocks in a log area are done by page-level mapping. In a hybrid mapping FTL called FMAX, for example, each data block is associated with zero or one extra block called a replacement block [5]. When a page in a data block is updated, the new version of the page is written into any clean page in a replacement block associated with the data block, if there is one already. Otherwise, a new replacement block is allocated from a pool of clean blocks and assigned to the data block.

If FMAX runs out of clean blocks, then it has to reclaim a clean block by merging a data block and its replacement block. Unless the number of replacement blocks available in an SSD is more than or equal to the number of data blocks (that is, 100% over-provisioning), each page write may trigger a block reclamation, which thrashes a replacement block between different data blocks and lowers replacement block utilization. Obviously, this problem will exacerbate for OLTP workloads in which page write requests are randomly scattered widely over a large data space.

To overcome the low block utilization and block thrashing

problems of FMAX, another hybrid mapping FTL called FAST relaxes the tight block-level associativity between data and log blocks by introducing full associativity between the two groups of blocks [17]. Since no single log block is tied up with any particular data block due to the full associativity, any page update can be written to *any* clean page in *any* one of the log blocks. This improves the block utilization significantly and avoids the block trashing problem as well. However, a new concern is raised by the full associativity itself of the FAST FTL. When a log block is chosen as a victim for reclamation, the log block may contain valid pages belonging to many different data blocks. Those valid pages should then be merged to their corresponding data blocks. This type of merge operation is called a full merge, and is usually very expensive as it involves merging as many blocks as the valid pages [9], [15].

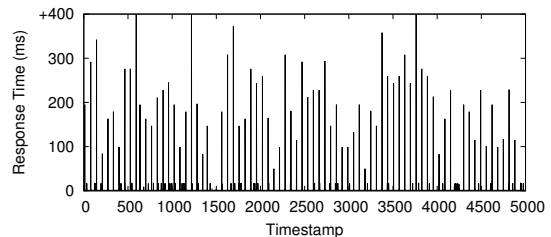


Figure 1. Write response times with FAST (log space: 3%)

If the amount of over-provisioned flash memory is not sufficient, log blocks may be reclaimed frequently and the full merge operations are also required frequently. Figure 1 shows the response times of page write requests, when the number of log blocks was as little as 3% of the total number of data blocks. The write response times fluctuated considerably, because the latency of a write request depended heavily on whether a full merge was triggered by the write request.

### III. FAST AND OLTP WRITE PATTERNS

In this section, we further analyze the write patterns in OLTP workloads with respect to the full associativity of the FAST FTL, and show how the skewed distribution of write requests affect the performance of FAST. We then present such concepts as write interval and log window, and show that FAST can identify hot and cold pages effectively when the amount of over-provisioned capacity (or log blocks) is sufficient.

#### A. Write Patterns in OLTP Workload

Typical on-line transaction processing (OLTP) systems such as retail, banking, insurance, and telecommunications, deal with a large number of concurrent read and write requests in order to process a continuous stream of small transactions [3]. Consequently, the access patterns in the OLTP workload are commonly characterized by small IO

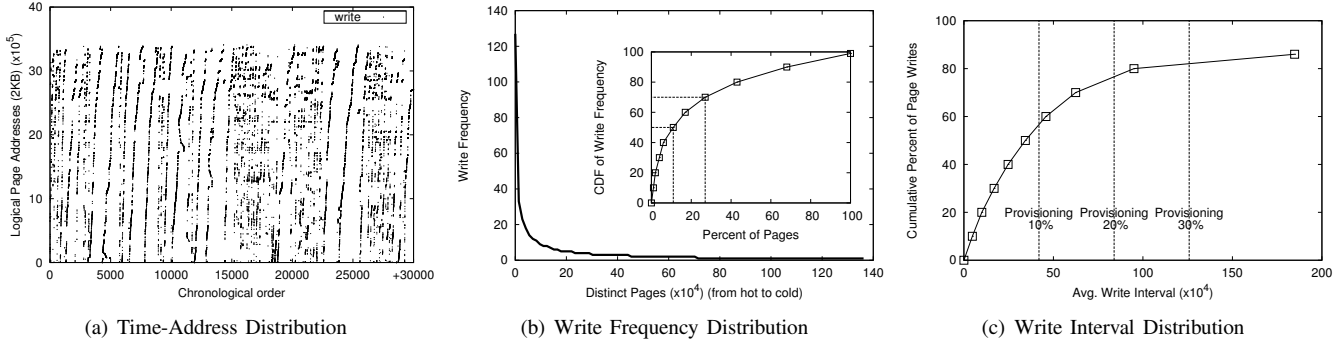


Figure 2. Write skewedness and temporal locality in TPC-C benchmark

operations (accessing mostly one or two data pages) randomly scattered over a large database space but with non-uniformly distributed access frequencies.

Since it is more critical to optimize the write performance of a flash memory device than the read performance, we focus on understanding how the patterns of write requests from OLTP workloads impact the performance of a flash memory device and its FTL. We obtained physical level IO traces by running a TPC-C benchmark, which is a standard benchmark for OLTP applications. The TPC-C benchmark is a mixture of five types of read-only transactions and update intensive transactions that simulates the activities commonly found in real world OLTP applications [18]. We used a commercial tool to create TPC-C database and workload, and ran the workload on a commercial database server with its page size set to 2KB. The size of a TPC-C database was 8GB, and the workload was based on the default ratio of mixed transactions with 30 concurrent virtual users who were accessing the database for four hours. During that time, we traced all the IO requests submitted to a disk drive that was bound as a raw device. The Linux block trace utility [1] was used to extract write requests only.

Figure 2(a) shows the pattern of writes observed in the TPC-C benchmark. The  $x$  and  $y$  axes in the figure represent the timestamps of write requests and the logical page addresses directed by the requests. We can make two important observations about the write pattern of OLTP in Figure 2. First, as shown in Figure 2(a), the write pattern in OLTP is characterized by a large number of small random write operations. The write requests from the database server were randomly scattered over the entire database of 8GB (or 1.4 million pages of 2KB), and most writes were as small as 2KB. The percentage of writes larger than 32KB was less than 0.01%. This type of write pattern provides a clear contrast with those found in embedded and personal computing applications, where sequential writes are dominant, and strong temporal and spatial locality are observed.

Second, the write requests in TPC-C benchmark are skewed. As shown in Figure 2(b), there are pages that are written to more frequently than others. In the figure, the

logical addresses of 1.4 million pages are sorted in the descending order of their write frequencies. While each page of the 1.4 million pages is written at least once during the benchmark run, these pages are only about 35 percent of the entire database space of 8GB. Again in the figure, we can see that 70% of the writes were directed to 30% of the pages, and almost 50% of the writes were to the 10% of the pages. This means that there was a deeper skew in the head of the distribution. Another to note is that the 30% of the writes in the tail of the distribution were scattered rather uniformly over 70% of pages.

In order to better understand the write pattern with respect to the access frequency of individual pages, we use the concept of *write interval*. This is useful in understanding the performance of FTLs, especially for FAST and *FASTer* (that will be presented in the next section). The write interval of a page  $p$  is defined to be the number of write requests made to the other pages between two consecutive write requests made to the page  $p$ . Figure 2(c) shows the cumulative distribution of average write intervals of pages that were calculated every individual write request.

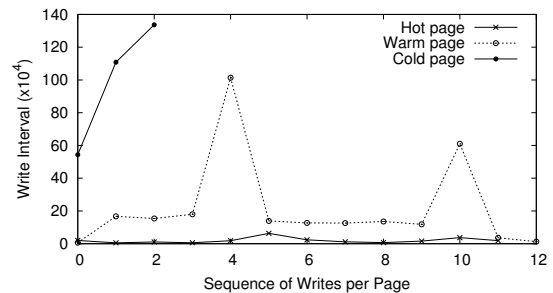


Figure 3. Write intervals: hot, warm and cold pages

Write intervals provide a quantitative way of distinguishing hot pages with high temporal locality (or cold pages with low temporal locality) from the rest. Roughly speaking, a page with short average write interval is considered hot, while a page with long average write interval is considered cold. In this paper, we classify data pages to three different

types based on their average write intervals: *hot*, *cold* and *warm*.

Figure 3 shows examples of hot, cold and warm pages taken from the TPC-C trace. In the figure, the  $x$  axis represents the ordinal sequences of writes requested for each of the three page types, and the  $y$  axis represents the intervals between two consecutive writes. In the figure, the write intervals of a hot page was less than 100K requests, and the write interval of a cold page was greater than 500K requests. There were only three writes for the cold page in the TPC-C trace. The average write interval of a warm page shown in the figure was about 200K requests, but, in general, the range of write intervals for warm pages is stretched rather broadly.

As will be discussed in Section IV, additional attention is paid to warm data pages by the *FASTer* FTL, because their average write intervals are not short enough to be invalidated by FAST with a given log area, but are still short enough so that frequent merges for them can be avoided with a little larger log area.

### B. Impact of Write Intervals on FTLs

In the previous subsection, we explained the skewed write patterns in TPC-C benchmark and introduced the concept of write interval. In this subsection, we will explain why the FAST FTL scheme is very efficient in exploiting the write skewedness in OLTP applications using the concept of write interval. The FAST scheme, in combination with skewed writes, can distinguish hot and cold pages without any special mechanism for measuring the hotness of pages.

First, recall that the main role of the log space in FAST is to absorb the writes. We define the *log window size* as the number of page writes that can be absorbed by blocks in the log space. For example, suppose there are  $M$  log blocks in the log space and each block contains  $b$  pages. Then, the blocks in the log area can absorb  $M \times b$  page write requests.

It is important to understand the relationship between the log window size and the write interval. After a page is written at the end of the log space in FAST, it can remain in the log space until the log block containing the page becomes the victim. Because the victim block in FAST is chosen in round-robin fashion [17], a written page will remain in the log space until as many new writes as the log window size arrive. When a page is written in the log space, if its write interval is less than the log window size, then the old copy of the page stored in the log space will be invalidated before its log block becomes a reclamation victim. When the log block containing the invalidated copy of the page becomes a victim, no merge operation will be required for the page as it has already been invalidated [17].

Therefore, if the average write interval of a (hot) page is less than the log window size, FAST will invalidate the old copy before its log block becomes victim, and thus avoiding the merge operation. In contrast, if a (cold) page has an

average write interval larger than the log window size, FAST has to merge the page. In this respect, FAST has the ability to distinguish hot pages from cold ones, although the hotness itself is a relative concept depending on the log window size.

Note that the log window size is determined by the amount of over-provisioned storage capacity made available to an FTL as a log space. If the log space provided for FAST is too small (e.g., 3% of the entire storage capacity [9]), then only a small number of pages with very short write intervals would benefit from the FAST scheme (by avoiding merges), and the remaining hot pages will experience the merge operations when the log block containing the recent copy of the page become a victim. This is the main reason why FAST is unfairly perceived as lacking the ability to exploit locality and suffering from excessive full merge operations [9], [15]. Again, it was merely because of the fact that the 3% of the log space was too small for FAST to exploit its potential ability to skip numerous merge operations for hot pages.

### C. Impact of Log Window Size

The next question to ask is obviously “what is the right size of a log window?” If the log window size increases by making the log space large, there will be more pages whose write intervals come with the new log window, and a block reclamation will cause a merge operation for a smaller number of valid pages. In the case of write requests with a deep skew as shown in Figure 2, adding only a small amount of log space can skip numerous merge operations for many hot pages. In other words, with a larger log space, FAST could expand its ability to distinguish hot pages and avoid many merge operations, especially for writes with a skewed distribution, and thus its performance gain would outweigh the cost of an additional log space.

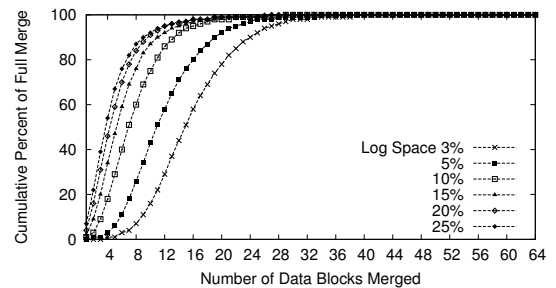


Figure 4. Cumulative distribution of a full merge cost

In order to confirm this conjecture, we evaluated the performance of FAST using our OLTP trace with a varying log space from 3% to 25%. The result is presented in Figure 4. In the figure, the  $x$  axis represents the average number of data blocks involved in a full merge operation triggered by a log block reclamation. Note that this number is no more than the number of valid data pages stored in a log block chosen as a reclamation victim. The  $y$  axis represents the

cumulative percentage of full merge operations that required a given number of data blocks to be merged.

In case of a 3% log space, for almost 90% of victim log blocks, each had more than 20 valid pages and needed to perform a merge operation for 15 to 20 data blocks on average. This result is consistent with the observations made by others [9]. As the size of a log space increased to 15% or beyond, however, the number of data blocks involved in a full merge decreased drastically. For example, in the case of a 15% log space, a full merge for more than 90% of all the victim log blocks involved less than 10 data blocks. The general trend is that, as the size of a log space increases, the log window size increases as well, and the write intervals of more data pages come within the enlarged log window making those pages more probable to be overwritten or invalidated and rendering merges for them unnecessary.

Another important aspect of the SSD performance is the stability in response time [9], [16]. The response time of a page write can vary considerably depending on whether merge operations are triggered. Stable response time is very important virtue of storage devices [11], [16], and thus it is important to minimize the deviation in response times. The standard deviation in response time was high for FAST with a 3% log space. However, the variation in response times was reduced considerably as the log space of FAST was increased.

Meanwhile, it is noteworthy in Figure 4 that, as the log space size increased to 25%, the reduction in the number of data blocks to be merged flattened. This can also be explained by the write skewedness shown in Figure 2. With a 15% log space, the write intervals of most hot pages were already shorter than the log window size. Consequently, adding more log blocks to the log space benefited only a small additional hot pages. In fact, as shown in the inner graph of Figure 2(b), the write intervals of all the other cold pages were so large that merge operations could not be avoided for them.

In this section, we have shown that FAST has the ability to deal with hot and cold pages in OLTP workloads, and, by providing a sufficiently large log space (10% to 20% of the storage capacity), FAST can successfully address its two main drawbacks, namely, the high overhead of full merge operations and the high variation in response time. Considering the current trend in industry towards providing sufficient over-provisioning for high performance SSDs, the FAST FTL appears well suited for enterprise workloads such as OLTP.

#### IV. FASTER FTL FOR OLTP APPLICATIONS

As is discussed in Section III, the write requests in the OLTP workload is clearly skewed, and different pages have different write intervals. Consequently, the performance of FAST will be affected by the size of a log space, because the number of merge operations that can be avoided depends on

the size of a log space. As is shown in Figure 4, FAST can avoid merge operations for a considerable number of data blocks by increasing the number of *physical* blocks allocated in a log space. However, this will increase the manufacturing cost of SSDs, as more flash memory chips will be required to increase the raw capacity of an SSD due to the increased size of a log space.

One of the goals of this work is to develop a new FTL that outperforms FAST significantly without consuming more resources (e.g., DRAM and log space) than FAST does. In this section, we present a new FTL called *FASTer*, an improved version of FAST, which can make a better use of temporal locality in the OLTP workload without having to increase the size of a physical log space. We also present a method that reduces the variance in the response time of page write operations.

##### A. Second Chance Policy

When a page write request arrives, FAST appends the page to the end of a log space. If the previous version of the page already exists in the log space, the old copy is marked as invalid. When the log block containing the invalidated copy of the page becomes a victim of reclamation, the invalidated copy will not be subject to a merge operation, because only the valid copy needs to be merged. In general, a merge operation is likely to be avoided for a data page whose average write interval is shorter than the log window. For the same reason, a data page with an average write interval longer than the log window will be likely to be merged when its log block is chosen as a reclamation victim, because the data page is unlikely to be overwritten before the log block is reclaimed.

Due to its high cost, the full merge operation of FAST is the most critical limiting factor in minimizing the latency and variation of write operations. As is mentioned above, the challenge here is to find a cost-effective way that allows us to avoid full merge operations for *warm* data pages (as the one shown in Figure 3) without consuming additional resources such as DRAM and over-provisioned flash memory.

The proposed *FASTer* FTL adopts a simple but elegant solution that extends the size of a log window virtually without increasing the number of log blocks in the log area physically. When a log block is chosen as a reclamation victim, *FASTer* carries the valid pages from the victim log block to a newly allocated log block instead of merging them with corresponding data blocks immediately. We call this a *second chance policy*, because the valid data pages stored in a victim log block are given another chance to stay in the log area before being merged to data blocks.

Figure 5 illustrates how the second chance policy of *FASTer* works. When a log block is reclaimed, each page in the log block is examined. An invalid page is simply ignored, because its logical data page has already been updated. A valid page is relocated to a log block newly allocated from

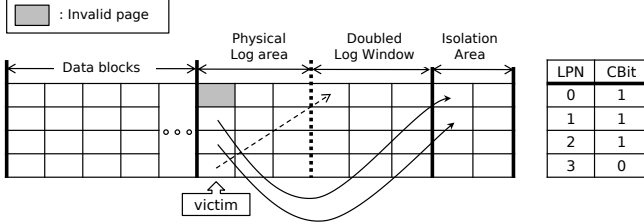


Figure 5. Second chance policy and extended log window

a pool of clean blocks, if the valid page has not been given a second chance yet. If there is a valid page that has been given a second chance already, the page is moved to a special location called an *isolation area* for *progressive merge*. The isolation area is part of a log area. More details about the isolation area and progressive merge will be described in the next section.

The logical-to-physical address mapping information will be updated for all valid pages carried over to a new log block or an isolation area. Once all the pages in a victim log block are processed, the victim block is erased and returned to the pool of clean blocks.

One important point to note in Figure 5 is that the second chance policy of *FASTer* has the effect of *doubling* the size of a log window. For warm pages whose average write intervals are longer than the physical log window but shorter than the virtually enlarged log window, *FASTer* is more likely to carry them over to another log block instead of performing costly merge operations. By letting the warm pages stay in the log area for an extended period of time, they will have more chances to be invalidated and will be less likely to be merged. This will result in considerably lower average response time of write operations and lower variation in response times as well.

### B. Isolation Area

With a virtually enlarged log window, more warm pages will avoid being merged to their data blocks prematurely. However, other cold pages whose write intervals are longer than the enlarged log window will continue to be carried over until they are eventually invalidated, increasing the average latency of a write operation and lowering the log block utilization.

*FASTer* addresses this problem by separating such cold pages in an isolation area. The cold pages stored in the isolation area are merged to the corresponding data blocks progressively, a few cold pages at a time, whenever a new write request arrives. They are merged this way in order to minimize the variation of response times for a write operation. Note that the isolation area is not a physically separated space but just a part of the log area.

## V. PERFORMANCE EVALUATION

### A. Evaluation Setup

In order to evaluate and compare the performance of four FTLs, FMAX [5], FAST, *FASTer*, and page mapping [13], we developed a simulator for each scheme, using the performance characteristics summarized in Table I. We ran each simulation using a TPC-C trace described in Section III. The size of a data space was 8GB. We assume that each FTL has a sufficient DRAM buffer space to store its mapping information, and we do not include the overhead for maintaining the mapping information persistently for power-off recovery. This setting is clearly favorable to page mapping that requires more DRAM for a large mapping table. In order to mimic the aging effect of flash memory devices [7], we measured the elapsed time from each simulation after overwriting all the flash blocks with the same trace once.

During each simulation run, the number of write, page copy-back and erase operations are counted, and the total elapsed time is calculated by multiplying the count of each operation by its latency and summing up the partial results.

### B. Performance Analysis

Figure 6 shows the total elapsed time of FMAX, FAST, *FASTer* and a page mapping FTL for the OLTP trace. The elapsed time of the FTLs were measured with a varying size of a log space from 3% to 25% of the data capacity.

As shown in Figure 6(a), FMAX was quite inferior to the others over the entire range of over-provisioning. Due to the random writes scattered over a large data space, almost 75% of data blocks received at least one page write. Because of the tight block-level associativity, FMAX caused even hot blocks to be subject to log block thrashing [17].

As the log space increased, FAST improved its performance proportionally as shown in Figure 6(a). This performance improvement was consistent with the improvement in merge costs shown in Figure 4. This confirms again that the full associativity between data blocks and log blocks and the finer grained page level mapping for log blocks match well with the random write patterns in the OLTP workload. On the other hand, Figure 6(b) demonstrates that the excessive full merges in FAST result in many page copy-backs, leading to the performance inferior to the *FASTer* or a page mapping FTL.

With the second chance policy and its extended log window, *FASTer* can avoid the merge operations for many warm pages and thus can outperform FAST by 30% to 35%. The reduced page copy-back operations in Figure 6(b) confirms the effect of second chance policy. In particular, the performance of *FASTer* with a 10% log space is similar to that of FAST with a 20% log space. The extended log window size, in combination with the deep write skews, makes *FASTer* comparable to FAST with a doubled log space. Despite the previous study reporting that page mapping schemes would generally outperform block mapping

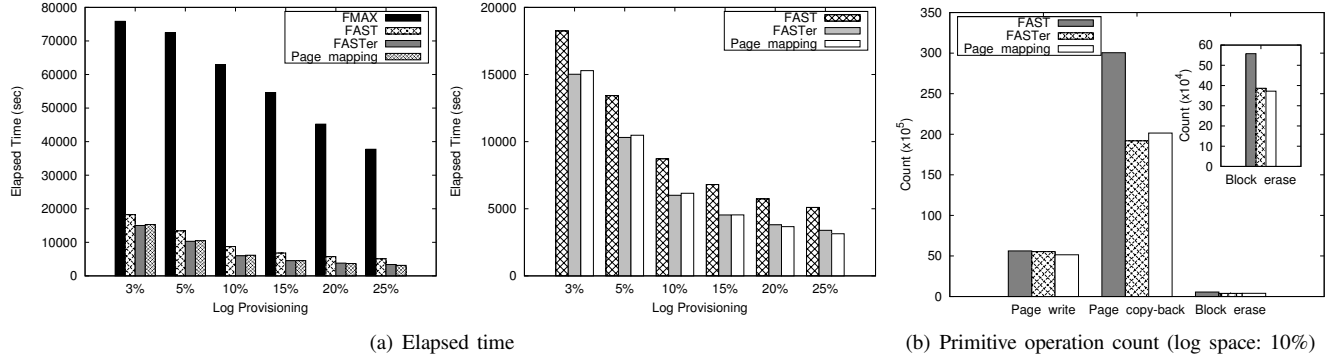


Figure 6. Performance comparison

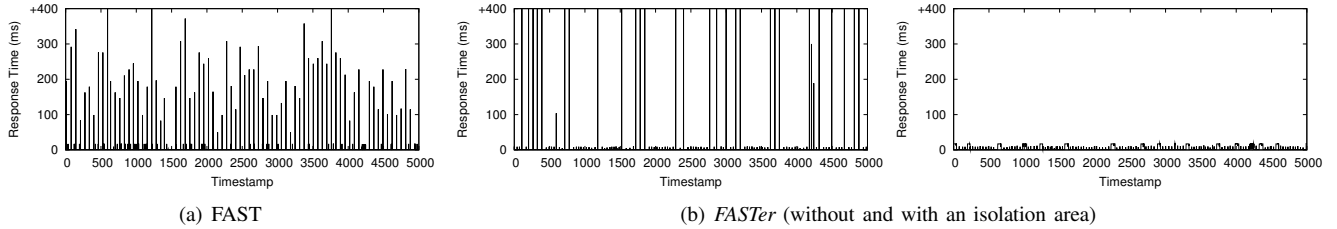


Figure 7. Response time variations with FAST and *FASTer* (log space : 3%)

Log Space (%)		3	5	10	15	20	25
Average Response Time (ms)	FAST	3.59	2.64	1.71	1.33	1.12	1.00
	<i>FASTer</i> (without isolation area)	3.11	2.11	1.24	0.92	0.76	0.66
	<i>FASTer</i> (with isolation area)	3.04	2.08	1.20	0.90	0.75	0.66
	Page mapping	3.00	2.05	1.20	0.89	0.72	0.61
Standard Deviation of Response Time (ms)	FAST	27.6	19.9	12.2	9.01	7.20	6.19
	<i>FASTer</i> (without isolation area)	38.9	30.8	21.6	17.3	14.6	12.7
	<i>FASTer</i> (with isolation area)	5.99	5.00	3.66	3.02	2.64	2.40
	Page mapping	5.73	4.74	3.44	2.77	2.32	2.01

Table II  
RESPONSE TIME COMPARISON

FTLs especially for random writes [9], the performance of *FASTer* was comparable to that of a page mapping FTL.

Figure 7 shows the response time trends in FAST and *FASTer* with a 3% log space. (For the convenience of comparison, Figure 1 is repeated here as Figure 7(a) for FAST.) The left figure in Figure 7(b) represents the variation in response time when only the second chance policy is applied, while the right figure in Figure 7(b) does when an isolation area as well as the second chance policy is applied. As shown in the figure, the use of an isolation area reduced the response time fluctuation drastically.

For more comprehensive performance evaluations, Table II presents the average response time and the standard deviation of response times of the FTLs by varying the log space from 3% to 25% of the data capacity. As we increased the log space size, *FASTer* outperformed FAST by more than 30% in average response time. *FASTer* also reduced the standard deviation (that is, the fluctuation of

response time) by a factor of three or four. In addition, the performance of *FASTer* was comparable to a page mapping FTL, which tends to consume much more DRAM to store a large mapping table. In summary, the *FASTer* FTL was cost effective and scalable for the OLTP workloads.

### C. Synthetic Workloads

In order to evaluate *FASTer* under more configurable workload conditions, we ran the FTL simulators with several synthetic workloads generated by the IOZone tool [2]. We modified the `random_perf_test` function in the IOZone tool so that it can generate only write requests with a skewed distribution. The skewedness can be controlled by two parameters  $x$  and  $y$  (denoted by  $x/y$ ) such that  $x\%$  of writes are directed to  $y\%$  of pages. Using this modified IOZone tool on the `ext2` file system of Linux kernel 2.6.31, we collected four IO traces with a varying skewedness from 90/10 to 60/40 by decrementing the  $x$  value by 10 (and

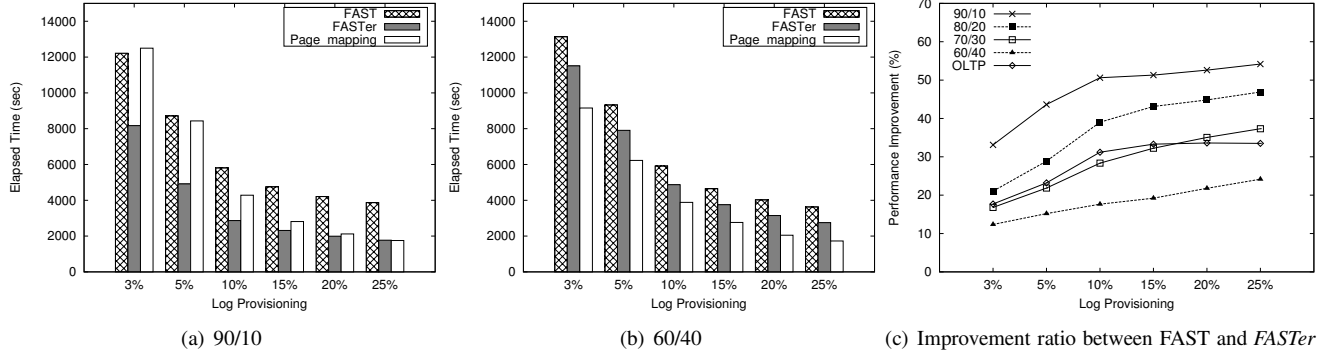


Figure 8. Performance comparison of non-OLTP workloads (synthetic workloads generated using a modified IOzone tool [2])

increasing  $y$  value by 10 as well).<sup>2</sup> Like the OLTP workload, the write traces were generated in a data space of 8GB, and a total of about 4 millions write requests were made to 1.4 million distinct pages.

Figure 8 shows the total elapsed time of FAST, *FASTer* and a page mapping FTL for the synthetic traces. The elapsed times of FTLs were measured with a varying log space size from 3% to 25% of the data capacity. In Figure 8(a) and Figure 8(b), the relative performance gap between FAST and *FASTer* grew larger as the skewedness increased in the write patterns. In particular, we observed two interesting results from Figure 8(a). First, the performance of FAST was almost comparable to that of the page mapping scheme. For this comparable performance between the two FTLs, there are two contributing factors. One is that, as the write requests are more skewed, more pages would be invalidated and thus more merges could be avoided by FAST. The other factor is that, as the write requests are more skewed to a smaller set of hot pages, there are more valid pages per blocks being reclaimed by the page mapping FTL, and this will incur more copy overhead during block reclamation and will also decrease the log block utilization in the page mapping scheme.

Second and more importantly, *FASTer* outperformed even the page mapping FTL significantly. This unexpected result can be explained as follows. The page mapping scheme, without the second chance policy, had to pay the cost of merges for warm pages, while *FASTer* had more chances to invalidate those warm pages using the enlarged log window avoiding merge operations for those pages.

Figure 8(c) shows the performance improvement ratio of *FASTer* over FAST with a varying size of a log space and different skewedness configurations. For a comparison purpose, we add the performance improvement ratio for the OLTP workload to the figure. The improvement ratio for OLTP was quite close to that for the trace with 70/30 skewedness. As the skewedness increased from 60/40 to

80/20, the improvement ratio increased steadily, and increased more notably when the skewedness increased from 80/20 to 90/10, especially with log provisioning between 3% and 10%. This steep performance improvement can be attributed to the rapid increase in the number of warm pages that can avoid the merge operations by *FASTer*.

## VI. CONCLUSION

In this paper, we propose *FASTer* as an enhancement of the FAST FTL. The improved performance and the low-variance response time of *FASTer* make it a cost-effective and scalable choice for the next generation large scale SSDs, especially for OLTP applications.

This paper makes several contributions. First, it shows that OLTP workloads have a skewed write distribution, and the skewed distribution can be exploited by FTLs such as FAST and *FASTer* utilizing over-provisioned flash storage. Second, more insightful analysis is provided for FTL performance using such concepts as write intervals for pages and a log space as a window of invalidation opportunity. Third, based on these concepts, a new FTL called *FASTer* is proposed that adopts a second chance policy to extend the log window virtually without increasing the number of physical log blocks. For warm pages with relatively long but not too short write intervals, the second chance policy allows many of them to avoid merge operations just by extending the log window. In our experiments, *FASTer* outperformed FAST by more than 30 percent in elapsed time for the TPC-C traces. Finally, in order to minimize the fluctuation in response time, we introduce an isolation area, develop the progressive merge strategy, and demonstrate its effectiveness in the experiment.

## ACKNOWLEDGMENT

The authors thank Mr. Woon-Hak Kang for helping to modify the IOZone tool for the skewed write patterns.

## REFERENCES

- [1] Blktrace. <http://linux.die.net/man/8/blktrace>.

<sup>2</sup>The `ext2` file system was used instead of `ext3` in order to capture the logical addresses of I/O requests unaffected by the journaling function.

- [2] IOzone Filesystem Benchmark. <http://www.iozone.org>.
- [3] Transaction Processing Performance Council. TPC-C Benchmark (revision 5.8.0). <http://www.tpc.org/tpcc/>, 2006.
- [4] A. Ban. Flash File System. Unites States Patent, No 5.404,485, 1993.
- [5] A. Ban. Flash File System Optimized for Page-mode Flash Technologies. Unites States Patent, No 5,937,425, 1999.
- [6] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A Design for High-Performance Flash Disks. *ACM SIGOPS Operating Systems Review*, 41(2):88–93, Apr. 2007.
- [7] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 181–192, 2009.
- [8] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang. Using Data Clustering to Improve Cleaning Performance for Flash Memory. *Software Practices and Experiences*, 29(3):267–290, 1999.
- [9] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 229–240, 2009.
- [10] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. Application Note AP-684, Dec. 1998.
- [11] Intel Corporation. OLTP Performance Comparison: Solid-state Drives vs. Hard Disk Drives. Test report, Jan. 2009.
- [12] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, 2006.
- [13] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-memory based File System. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 13–13, 1995.
- [14] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A Space-efficient Flash Translation Layer for CompactFlash Systems. *Consumer Electronics, IEEE Transactions on*, 48(2):366–375, May 2002.
- [15] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-aware Sector Translation for NAND Flash Memory-based Storage Systems. *SIGOPS Operating Systems Review*, 42(6):36–42, 2008.
- [16] S.-W. Lee, B. Moon, and C. Park. Advances in Flash Memory SSD Technology for Enterprise Database Applications. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 863–870, 2009.
- [17] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A Log Buffer-based Flash Translation Layer using Fully-associative Sector Translation. *ACM Trans. Embed. Comput. Syst.*, 6(3):18, 2007.
- [18] S. T. Leutenegger and D. Dias. A Modeling Study of the TPC-C Benchmark. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 22–31, 1993.
- [19] Samsung Electronics. 2Gx8 Bit NAND Flash Memory (K9WAG08U1A). Data sheet, 2006.