LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems

Sungjin Lee*, Dongkun Shin†, Young-Jin Kim* and Jihong Kim*

*School of Computer Science and Engineering, Seoul National University, Seoul, Korea
{chamdoo, youngjk, jihong}@davinci.snu.ac.kr

†School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea
dongkun@skku.edu

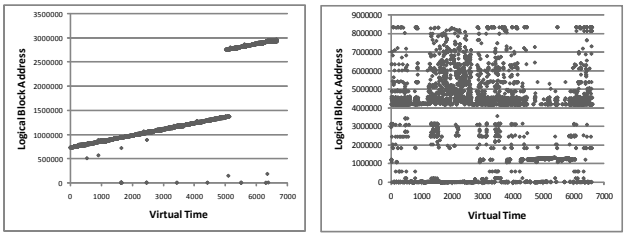
Abstract

As flash memory technologies quickly improve, flash-memory-based storage devices are becoming a viable alternative as a secondary storage solution for general-purpose computing systems such as personal computers and enterprise server systems. Most existing flash translation layer (FTL) schemes are, however, ill-suited for such systems because they were optimized for storage write patterns of embedded systems only. In this paper, we propose a new flash management technique called LAST which is optimized for access characteristics of general-purpose computing systems. By exploiting the locality of storage access patterns, LAST reduces the garbage collection overhead significantly, thus increasing the I/O performance of flash-based storage devices. Our experimental results show that the proposed technique reduces the garbage collection overhead by 54% over the existing flash memory management techniques.

1 Introduction

Flash memory has been widely used as a storage device for mobile embedded systems (such as MP3 players and PDAs) because of its low-power consumption, nonvolatility, high random access performance and high mobility. With continuing improvements in both the capacity and the price of flash memory, flash memory is increasingly popular in general-purpose computing markets. For example, leading notebook vendors recently started replacing hard disk drives with NAND flash memory-based solid state disks (SSD). Furthermore, as the energy efficiency of the enterprise systems becomes more critical [1], the enterprise systems are also expected to adopt more SSDs [2]. However, several limitations of flash memory make it difficult to replace hard disk drives with SSDs in a straightforward fashion.

Generally, NAND flash memory consists of multiple blocks, and each block is composed of multiple pages. Each page is a unit of read and write operation, and each block is a unit of erase operation. Unlike a traditional hard disk drive, flash memory does not support overwrite operations because of its write-once nature. When the data at a specific page is modified, the new data value is written to another empty page and the page with the old data should be invalidated. This special feature of flash memory requires two storage management schemes. First, we need to provide an address mapping scheme, which maps the logical address from the file system to the physical address in flash memory by maintaining an address mapping table. Second, we need a garbage collection scheme to reclaim the invalidated pages. The garbage collection scheme should select a block which has many invalid pages and erase the block to be reused after migrating the valid pages in the block to a clean block. In order to support these two management tasks, a flash translation layer (FTL) is commonly used between the file system and flash memory devices.



(a) An MP3 player (b) General-purpose applications

Figure 1: Logical block access patterns

The FTL is often implemented in resource-constrained environments and the overall I/O performance of the FTL implementation largely depends on its garbage collection efficiency. So, most existing FTL schemes were focusing on reducing the garbage collection overhead using a small address mapping table.

Although the existing FTLs perform efficiently for their target consumer electronics devices, the efficiency of their garbage collection scheme can be deteriorated when they were used in general-purpose computing systems. For example, in our experiments, we observed that the garbage collection overhead may account for 40%-60% of the total I/O time. The main source of this inefficiency comes from different write access patterns. As shown in Figure 1, most write requests in an MP3 player are sequential with only a small number of random writes. On the other hand, in general-purpose applications, the write request pattern is more complex with the following three characteristics. The

first difference is that there is a high temporal locality because there are many sectors which are updated frequently. The second one is that the sequential locality is also high (although it is not so high as in mobile consumer applications) because there are many sequential writes. The last one is that there are many random writes which are interposed between sequential writes.

In order to build a high performance FTL for general-purpose computing systems, these three characteristics should be taken into account. In particular, both the temporal locality and sequential locality should be efficiently exploited. To this end, we need to separate the sequential access from the random access. Therefore, the existing FTLs optimized for the access patterns of consumer devices should be redesigned to exploit write access patterns of general-purpose computing systems.

Based on this observation, we propose a new FTL scheme using a locality-aware sector translation (LAST). To exploit both temporal locality and sequential locality[1], we reorganize the flash memory space into two regions depending on the type of locality and adopt more intelligent garbage collection policies for each region. Experimental results based on a trace-driven simulator show that LAST reduces the garbage collection overhead by up to 54% over the existing FTLs for general purpose applications.

The rest of this paper is organized as follows. In Section 2, we survey previous approaches in managing flash memory. Section 3 describes the details of the proposed LAST scheme. Experimental results are presented at Section 4. Section 5 concludes with a summary and directions for future works.

2 Background and Related Work

2.1 Flash Translation Layer

Generally, FTL schemes can be classified into three groups depending on the granularity of address mapping: page-level, block-level, and hybrid-level FTL schemes. In the page-level FTL scheme [3, 4, 5], a logical *page* number (LPN) from the file system can be mapped to a physical page number in flash memory. This mapping approach shows a great garbage collection efficiency, but it is impractical due to its huge mapping table size. In the block-level FTL scheme [6], only the logical *block* number of a logical address is mapped to a physical block number and the page offset in a block is not changed. By using more coarse-grained mapping units, it can reduce a mapping table size significantly. However, when the data of a page should be modified into a new data, all the data in the corresponding block as well as the new data should be written into another empty block. This constraint results in high garbage collection overhead. In order to overcome these disadvantages, the hybrid-level FTL scheme was proposed. Log buffer-based FTLs [7, 8, 9] are representative hybrid-level schemes. They show a high garbage collection efficiency and require a small-sized mapping table.

[1] In this paper, the sequential locality refers to the property that if an access has been made to a particular location p, then it is likely that an access will be made to the location $(p + 1)$ in the near future.

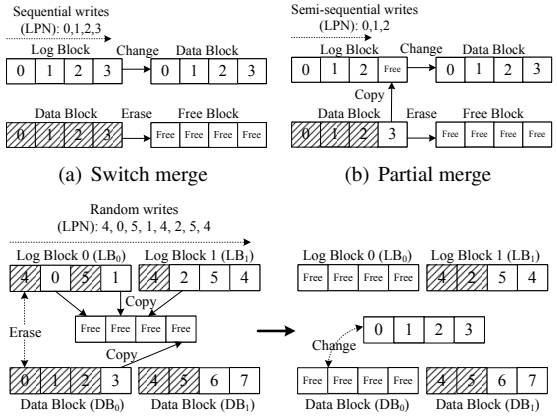


Figure 2: Three types of merge operations

2.2 Log buffer-based FTL schemes

In the log buffer-based FTL scheme, it distinguishes flash memory blocks into *data blocks* and *log blocks*. Data blocks represent the ordinary storage space, and are managed by the block-level mapping. Log blocks are the invisible storage space to be used for storing log data, and handled by the page-level mapping. A set of log blocks is called a *log buffer*. Because only the small fixed number of log blocks is used, the memory overhead for page-level mapping is low. When a write request modifying the data in a data block arrives, the log buffer-based FTL writes the new data (log data) temporarily in the log buffer invalidating the corresponding data in the data block. So, it can reduce the extra operations required to maintaining the data block's block-level mapping information. However, all the log blocks are exhausted, some of log data in log blocks should be flushed into the data blocks to make free space in the log buffer. The valid data in the data block and the log data of the log block should be merged and rewritten into an empty data block. We call this work as a *merge operation*.

Figure 2 illustrates three types of merge operations: *switch merge*, *partial merge* and *full merge*. In this figure, we assume that each block is composed of four pages. The number within the small boxes denotes a LPN of each page and the shaded box represents a page with old data (invalid page). The switch merge is the most cheap merge operation. As shown in Figure 2(a), the FTL simply erases the data block with only invalid pages and changes the log block into a data block. Therefore, it requires only one erase operation. The switch merge is performed only when all

the pages in the data block are sequentially updated starting from the first logical page to the last logical page. The partial merge is similar to the switch merge. But, it requires additional copy operations, as depicted in Figure 2(b). After all the valid pages are copied, we simply apply the switch merge. The partial merge typically occurs when the sequential write does not fill up one block (semi-sequential). The full merge operation is most expensive. Figure 2(c) shows the snapshot of the full merge. There are two log blocks, LB_0 and LB_1, and two data blocks, DB_0 and DB_1. We assume that LB_0 is selected as a victim log block. The FTL first allocates one free block and copies all the valid pages both from LB_0 and from DB_0 to the free block. Especially, the data block DB_0 is called an *associated data block* of LB_0 because it has the corresponding invalid page for a valid page in the victim block LB_0. The number of the associated data blocks can be increased up to the number of pages per a single block. After copying all the valid pages, the free block becomes a data block, and DB_0 and LB_0 are erased. Therefore, the full merge requires several copy operations and erase operations. The full merge is typically required when the pages are updated in random order.

2.3 Related Work

Kim *et al.* have proposed a log buffer-based FTL scheme which uses a block associative sector translation (BAST) [7]. In the BAST scheme, one data block is associated with only one log block, i.e., a log block can have log data only for a data block. If there is a write request, its data is written into the corresponding log block sequentially. The merge operation is triggered when there is no associated log block for a write request and there is no free log block. BAST provides an efficient garbage collection for consumer devices whose access patterns are mainly sequential. This is because most merge operations can be performed by the cheap switch merge. However, as the write pattern becomes more random, the space utilization of the log buffer gets worse because even a single page update of a data block requires a whole log block. So, when a large number of small-sized random writes are issued from the file system, most of log blocks are selected as victim blocks with only a small portion of the block being utilized. This phenomenon is called a *log block thrashing problem* [8]. Since all the under-utilized log blocks should be merged by the full merge, the merge cost is significantly increased.

In order to overcome this shortcoming of the BAST scheme, a fully associative sector translation (FAST) [8] has been proposed. In FAST, one log block can be shared by all the data blocks, and update requests are sequentially written in a log block irrespective of their corresponding blocks. The garbage collection is performed only when there is no free space in the log buffer. This approach efficiently removes the block thrashing problem, and then increases the garbage collection efficiency for the random

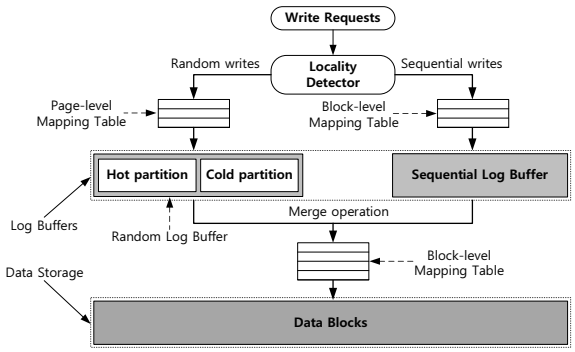


Figure 3: The overall architecture of the LAST scheme

workload. FAST also maintains a single log block, called sequential log block, to manipulate the sequential writes. However, FAST does not consider the multiple sequential write streams by multiple tasks and does not efficiently handle the mixture of random write request and sequential write request. In addition, it does not exploit the temporal locality of the random writes.

Recently, a SUPERBLOCK scheme [9] demonstrated that the temporal locality can be exploited by allowing the page-level mapping in a superblock which is a set of consecutive blocks. Then, the cold data and the hot data are separated automatically into different blocks within a superblock, thus the garbage collection efficiency is improved by reducing the number of full merge operations. However, their approach does not efficiently distinguish the cold pages from the hot pages. Moreover, the critical shortcoming of the SUPERBLOCK scheme is that the page-level mapping information within a superblock should be maintained.

3 Locality-Aware Sector Translation Scheme

3.1 Overall Architecture

As noted in Section 1, the typical workload in the general purpose computing is a mixture of random writes and sequential writes. Therefore, it is important to extract the sequential writes from the mixture of different write requests so that more switch merges and partial merges can be applied. Additionally, by isolating the random writes, we can efficiently handle the temporal locality of the random writes. To do that, LAST partitions the log buffer into two parts; *random log buffer* and *sequential log buffer* as shown in Figure 3. Upon the arrival of a write request, the *locality detector* identifies the type of locality of the write request and sends it into the sequential log buffer if it has a sequential locality. Otherwise, LAST sends it into the random log buffer.

The sequential log buffer consists of several sequential log blocks, and one sequential log block is associated with

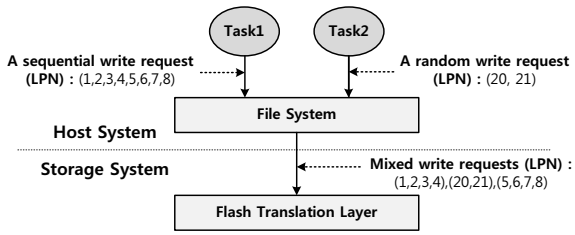


Figure 4: Behaviors of write requests

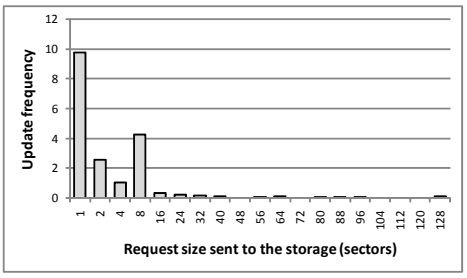
only one data block (block associative mapping) like BAST [7]. This is because the block associative mapping is appropriate for the sequential access patterns. The random log buffer is composed of several random log blocks, and each random log block can be associated with multiple data blocks (fully associative mapping) like FAST [8]. The fully associative mapping is advantageous to manage the random access patterns, especially when they have a high temporal locality. Additionally, LAST divides the random log buffer into two partitions, hot and cold, and redirects the arriving requests to either one depending on their temporal localities. By clustering the data with high temporal locality within the hot partition, we can reduce the merge cost of the full merge, which mainly occurs in the random write patterns.

3.2 Detecting the Locality Type

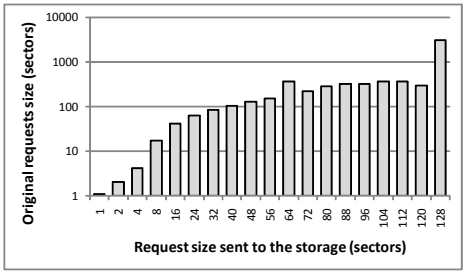
To exploit the different locality type of each request, we need a policy which determines the locality type of a write request. Figure 4 shows the typical behaviors of write requests which are generated by multiple tasks and are sent to the FTL through the file system in a general-purpose computing system. Since multiple I/O requests are usually interleaved at the FTL due to a multi-tasking, we need to know how the locality of each request may change.

From the observation on the characteristics of the I/O requests gathered from the general-purpose computing system, we found that the locality type of each request is deeply related to its size. Figure 5(a) illustrates the relationship between the write update frequency and the size of the request sent to the FTL. The update frequency of the request of size s (sectors) represents the average number of update operations over all requests of size s. As shown in Figure 5(a), small writes have high temporal localities while large writes have lower temporal localities.

Another observation is that small writes have little sequential localities. Figure 5(b) shows the relationship between the size of each write request and the size of its original request. The original request is a request issued by a task to the file system as shown in Figure 4. The file system may divide a long sequential write request into small write requests and send them to the FTL, interleaving a write request from another task. So, the request size shown to the FTL will be smaller than the original size of the write request generated by tasks. However, as shown in Figure 5(b), we notice that small writes almost come from small original writes, and a large write request is likely to be a part of a long sequential access. Since I/O clustering strengthens the sequential property of each request in a request queue within a device driver [10], if a write which arrives at the FTL is small, we can assume that it is likely to have a random access pattern. Therefore, we can presume that a small write in general-purpose computing systems usually has a high temporal and low sequential locality; on the other hand, a large write has a relatively high sequential locality.



(a) A distribution of the update frequency over the request size

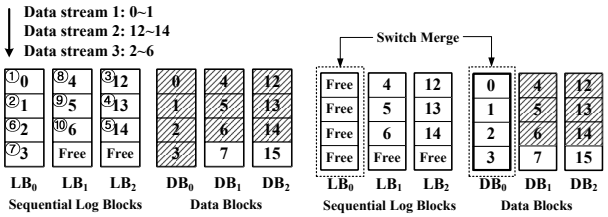


(b) A distribution of the original request size over the request size

Figure 5: Characteristics of write requests

Based on these observations, we made a simple locality detecting policy based on the request size without maintaining additional information for identifying the locality type of the write request. This policy determines the locality type by comparing the size of each request with a threshold value. If the size of the request is larger than the threshold value, it is redirected to the sequential log buffer; otherwise, it is written in the random log buffer. This threshold value should be carefully determined. If the threshold is too small, a large amount of small data will be frequently written in the sequential log buffer, incurring the block thrashing problem. If the threshold is too large, sequential writes will be forwarded to the random log buffer. Therefore, LAST may lose many chances for the switch merge while it increases the number of the full merges since the data written in the random log buffer should be evicted by the full merge.

In our experiment, we found that the overall merge cost is minimized when the threshold value is 4 KB (i.e., 8 sectors), especially for write request patterns on the Microsoft



(a) Write process (b) Switch merge

Figure 6: Operations in the sequential log buffer

Windows XP operating system if we assume that a single block size is 128KB.

3.3 Exploiting Sequential Locality

Figure 6(a) shows the write process in the sequential log buffer when three sequential data streams are issued from the file system. In this example, we assume that all the log blocks are initially empty. The write sequence of each page is denoted within a small circle. When all the sequential log blocks are exhausted, LAST first searches the log block where all the pages are valid data and do the switch merge. If there is no such a log block, LAST selects the victim log block using LRU replacement policy and then applies the partial merge. Figure 6(b) shows the switch merge operation. Since all the pages in the LB_0 are occupied by newly updated data, LB_0 is selected as the victim.

Usually, in general-purpose computing environments, several sequential write streams are simultaneously issued from the file system. By maintaining several sequential log blocks, we can accommodate multiple sequential write streams in the sequential log buffer. So, we can reduce the number of the partial merges due to the contention between multiple sequential streams. For instance, in Figure 6, if there is only one sequential log block like FAST [8], each sequential data stream will expel the other data stream from the one sequential log block. So, its performance degrades in the general-purpose computing systems.

3.4 Exploiting Temporal Locality

In the random log buffer, the full merge inevitably occurs. To reduce the full merge cost, LAST utilized the temporal locality. Before describing how to exploit the temporal locality, it is necessary to know which factor determines the full merge cost.

3.4.1 Modeling the Full Merge Cost

As shown in Eq. (1), the full merge cost consists of the page migration cost and the block erase cost. Before the full merge, we first select the victim log block, and identify the set of associated data blocks of the victim block. In this paper, we define the number of associated data blocks as an *associativity degree*, and denote it as N_a. Each page in an associated data block should be migrated into empty data block from the associated block if it is valid in the data block or from the log block if it is invalid in the data block. Then, the number of page migrations for each associated data block is N_p when there is no free page in the data block, where N_p is the number of pages per a single block. After copying all the valid pages, we erase associated data blocks and the victim log block. Therefore, we can express the full merge cost as Eq. (2) where C_e means the cost for erasing a single block and C_c is the cost for copying a single page.

$$full\ merge\ cost = migration\ cost + erase\ cost \quad (1)$$

$$= N_a \times \{(N_p \times C_c) + C_e\} + C_e \quad (2)$$

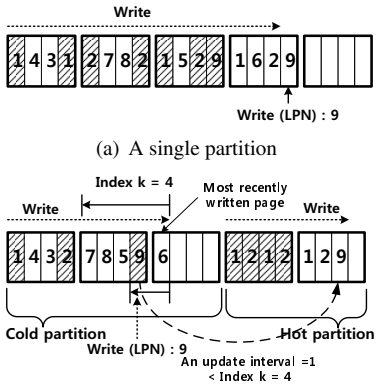
From Eq. (2), we can know that the overall full merge cost deeply depends on the associativity degree N_a. Consequently, how to reduce the overall associativity degree is a critical point in reducing the full merge cost.

In this paper, we use the temporal locality of random writes as a key consideration in reducing the associativity degree. The first approach is to cluster pages with high temporal locality (hot pages) into the same log block. The hot pages are frequently updated than other pages (cold pages), thus make a large number of invalid pages within the random log buffer. Then, we can increase the number of log blocks which has no associated data blocks, and get a free block without the page migration. The second approach is to wait until the associativity degree of a log block is decreased if it has the possibility. Since the hot pages are likely to be invalidated by the frequent writes, it is more beneficial to select the victim among the cold blocks whose associativity degree will rarely be changed. In the LAST scheme, these two ideas are implemented by the random log buffer partitioning policy and random log buffer replacement policy, respectively.

3.4.2 Random Log Buffer Partitioning Policy

The random log buffer partitioning policy is proposed to efficiently remove invalid pages from the random log buffer. We found that a large amount of invalid pages (above 50%) occupy the random log buffer space, and most of them are originated from hot pages whose data is updated frequently. However, if the invalid pages are distributed into several log blocks, a random log block will have both invalid pages and valid pages. Then, a full merge should be performed to reclaim such a log block.

To make the invalid pages to be generated mainly in a small portion of log blocks, we divide the random log buffer into two partitions, one for hot pages (hot partition) and the other for cold ones (cold partition), and redirects arriving requests into different partitions depending on their temporal localities. Since hot pages are frequently rewritten, all the hot pages within the same log block are likely to be invalidated in the near future. Since a dead block which has only invalid pages is not associated with any data blocks (N_a =



(a) A single partition

(b) Two partitions (Hot/Cold)

Figure 7: Write processes in two types of the random log buffers

0), only one erase operation is required during the garbage collection. Since a lot of dead blocks are generated from the hot partition, by aggressively evicting dead blocks, we can reduce the full merge cost. Additionally, this also delays the merge operation on the cold partition. This delayed merge is usually beneficial to reduce the full merge cost. This is because lots of pages that belong to the cold partition can be invalidated during the extended time, thus decreasing the overall associativity degree of the cold partition. Figure 7 shows the write processes in the random log buffer with two partitions, and compares it with a single partition.

When partitioning the log buffer, we need to determine which pages should be regarded as hot. As shown in Figure 7(b), LAST distinguishes the hotness of each page based on its update interval. The update interval can be measured by a page distance between the most recently written page and the page with old data of the requested page. To evaluate whether an update interval is frequent or infrequent, we define an index k as a criterion. If the update interval of the requested page is smaller than k, we regard it as a hot page; otherwise, it is regarded as a cold page. Once a page is regarded as hot, it remains in the hot partition until it is evicted. For example, in Figure 7(a), the data of the requested page is sequentially written to the random log buffer regardless of its hotness. On the other hand, in Figure 7(b), pages 1, 2 and 9 are treated as hot pages because their update intervals are smaller than k, and thus clustered into the hot partition.

The index k should be periodically adjusted depending on the workload pattern to effectively identify the hot pages. Moreover, we also need to redefine the size of the hot partition since as the k value changes, the amount of data to be written in the hot partition also changes. To control them, we refer to a *space utilization* of each partition (U_{hot} and U_{cold}) and *the number of dead blocks* in the hot partition (N_d). The space utilization can be formally defined as the ratio of the valid pages in each partition.

Assuming that the hot pages are properly identified, the size of the hot partition is changed in the following two cases. The first case is when N_d is being increased. Because it means that too many log blocks are assigned to the hot partition, we reduce the size of the hot partition. If N_d is reduced while U_{hot} is increased, it means a large number of hot pages remain as a valid status due to the lack of the space in the hot partition. Therefore, we increase the hot partition size. Assuming that the size of hot partition is large enough to accommodate the hot data, the k value can be changed in the following cases. If N_d is being reduced and U_{hot} remain relatively low, it means that a number of cold pages are distributed in the hot partition. Therefore, we reduce the k value. If U_{cold} is being reduced, it means that a number of hot pages are written in the cold partition. So, we increase the k value.

3.4.3 Random Log Buffer Replacement Policy

The random log buffer replacement policy is proposed to provide a more intelligent victim block selection. The victim selection policy is composed of two steps. In the first step, we determine a victim partition, where the victim log block is to be selected. If there is a dead block in the hot partition, we select the hot partition as the victim partition; otherwise, we choose the cold partition. The rationale behind this approach is to delay the eviction of the hot pages in the hot partition as long as possible. However, when there are pages which were hot pages but lose their temporal localities and changed into cold pages, we should evict them. To do this, if there is a log block whose updated time is smaller than a certain threshold time, we select the hot partition as the victim partition.

The second step is to select the victim log block from the victim partition. If the victim partition is the hot partition, we select a dead block as the victim log block. If there is no dead block, we choose a least recently updated log block. For the cold partition, we choose the log block with the *lowest merge cost* as the victim. By recycling this block, we do the full merge at the lowest cost, and expect log blocks with higher merge cost to remain in the random log buffer until their full merge costs become small enough. To do this, we need to maintain a merge cost table. Each entry of the merge cost table keeps the associativity degree of each log block, and only requires $\log_2 N_p$ bits.

After merging the victim, we need to reassign the reclaimed log block to either partition. If the victim is a dead block from the hot partition, it means that there are enough dead blocks, thus give a free block to the cold partition. If not, we give a free block to the hot partition.

Before finishing this section, we need to mention wear-leveling issues related to the LAST scheme. Typically, each block of NAND flash memory has a finite number of erase-write cycles, and a block becomes unreliable after the limit. In the LAST scheme, since hot pages are likely to be kept in

Table 1: Key parameters of the target large block NAND flash memory

NAND flash memory organization	Block size	128 KB
	Page size	2 KB
	Number of pages per block	64
Access time for each operation	Read operation (1 page)	25 usec
	Write operation (1 page)	200 usec
	Erase operation (1 block)	2000 usec

the hot partition, blocks that belong to the hot partition are intensively erased. On the other hand, blocks with cold data are rarely updated, thus erase counts of these blocks will be much smaller than those of other blocks. Due to this, we need to adopt a hot-cold swapping algorithm, which tries to balance erase cycles by periodically swapping the blocks containing hot data with blocks having cold data.

4 Experimental Results

To evaluate the performance of the LAST scheme, we have developed a trace-driven FTL simulator. We compared LAST over three existing FTL schemes: BAST [7], FAST [8] and SUPERBLOCK [9]. The flash memory model used in simulation is Samsung K9WBG08U1M large block NAND flash memory. Important parameters are listed in Table 1. The workloads used for our experiments were extracted from Microsoft Windows XP-based a notebook PC and a desktop PC, running several applications, such as documents editors, music players, web browsers and games. In addition, we also captured storage access patterns while running the TPC-C benchmarks to reflect the workload of an enterprise server system. We compared the number of copy operations and the number of erase operations during the merge operations. The garbage collection overhead is calculated by multiplying the number of each operation by the corresponding time value listed in Table 1.

Figure 8 shows the normalized garbage collection overhead for each FTL scheme when the log buffer size is 512 MB (=4096 log blocks). If we assume that the total capacity of flash memory is 32 GB, the ratio of the log buffer is less than 1.56% of total flash memory space. The sequential log buffer size is set as 32 MB (=256 log blocks), and the remaining log blocks are assigned to the random log buffer. Among all the evaluated schemes, LAST shows the best garbage collection efficiency. LAST reduces the garbage collection overhead by 46-67% compared with the SUPERBLOCK scheme.

BAST shows the worst garbage collection efficiency compared to other FTLs. This is because BAST cannot efficiently handle random write patterns. As mentioned in Section 2, the block associativity mapping results in the block thrashing problem for the workloads containing a large amount of random writes. Therefore, the overall cost of the full merge is significantly increased in BAST. Although the LAST scheme uses the block associativity mapping for managing the sequential log buffer, the most of the random writes can be filtered by the locality detector. So, the block thrashing problem does not occur in LAST.

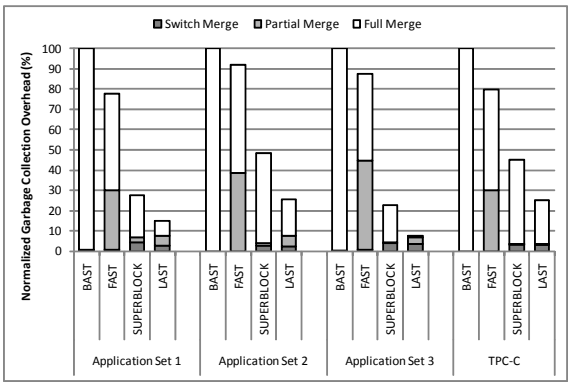


Figure 8: Normalized garbage collection overhead

FAST exhibits a better garbage collection performance than BAST by efficiently removing the block thrashing problem. However, it cannot outperform LAST because it does not exploit the locality of traces. First of all, it does not exploit the temporal locality of the random writes. Therefore, its full merge cost is the largest among all the schemes, except for the BAST scheme. FAST also shows high partial merge cost. This is because the sequential locality of the sequential write patterns is likely to be broken by the random write requests. As a result, it reduces the chance of the switch merge operation, and thus increases the overall merge cost. On the other hand, in the LAST scheme, since the locality detector isolates the sequential writes from the random writes, we can take advantage of the switch merge for reducing the overall garbage collection overhead.

The SUPERBLOCK scheme exhibits more improved garbage collection performance compared with FAST. Especially, its partial merge cost is much smaller than that of the LAST scheme. In the LAST scheme, semi-sequential writes are redirected into the sequential log buffer. Therefore, when they are evicted from the sequential log buffer, the partial merge is required for merging them. This is why the LAST scheme yields more increased partial merge cost than the SUPERBLOCK scheme. However, by removing the semi-sequential writes from the random log buffer, LAST can reduce the full merge cost significantly. Consequently, due to the reduced full merge cost, LAST can outperform SUPERBLOCK. In addition, more efficient hot/cold separation policies used on the LAST scheme contribute to reducing the full merge cost.

The LAST scheme also shows the best garbage collection efficiency in TPC-C benchmark. The workload of TPC-C was collected while executing the *undo* operations on an Oracle relational database system. The access patterns of TPC-C are categorized into two types: random writes with

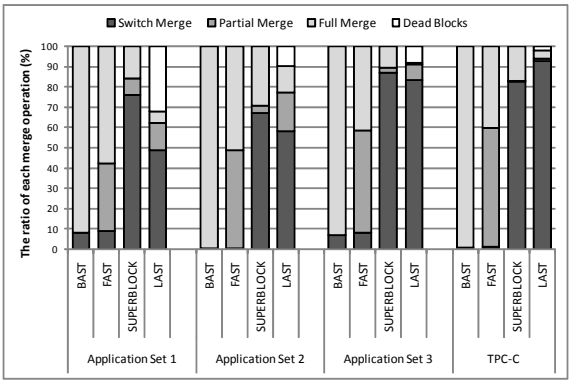


Figure 9: The ratio of each merge operation

high temporal locality and sequential writes. Because these two types of write requests are simultaneously issued from the file system, they arrive at the FTL layer in mixed patterns of random and sequential accesses. LAST efficiently extracts the sequential writes from the random writes and isolates them into the different log buffer. Due to high temporal locality of the random writes, a large number of dead blocks are generated from the random log buffer. On the other hand, in the sequential log buffer, most of the merge operations can be performed by the switch merge.

Figure 9 shows the ratio of each merge operation. For the LAST scheme, we also denote the number of dead blocks occurred during the merge operations. In the BAST scheme, most of the merge operations are performed by the full merge operation because of the block thrashing. In the FAST scheme, the full merge and partial merge account for about 50% of the total merge operations, respectively. In the SUPERBLOCK and LAST schemes, most of the merge operations are performed by the switch and partial merge. However, the number of the full merges in LAST is much smaller than that of SUPERBLOCK since a large number of dead blocks are generated in the random log buffer.

Finally, we compare the mapping table size of each FTL scheme. When the total capacity of flash memory is 32 GB and the log buffer size is 512 MB, the mapping table size of LAST (1.96 MB) is slight smaller than those of other schemes (2.0 MB). This is because LAST handles the sequential log buffer using the block-level mapping instead of the page-level mapping. Additional memory space required for the merge cost table is quite small (3 KB).

5 Conclusion

We have proposed a new FTL scheme called LAST which is designed to support a more efficient garbage collection in general-purpose computing systems with flash memory as a secondary storage. By exploiting both temporal locality and sequential locality, LAST reduces the garbage collection overhead by 54% over the existing FTLs.

The proposed LAST scheme can be further improved in several directions. First, in this paper, we fixed the size of the sequential log buffer as 32 MB. However, we observed that the overall garbage collection overhead can be further reduced by adjusting the size of the sequential log buffer dynamically. Second, the proposed locality detector cannot efficiently identify sequential writes when the small-sized write has a sequential locality. Therefore, we are developing a more intelligent locality detection algorithm.

6 Acknowledgements

This work was supported in part by the Brain Korea 21 Project in 2007. This work was also supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (MOST) (No. R0A-2007-000-20116-0). The ICT at Seoul National University provides research facilities for this study.

References

[1] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. "JouleSort: a balanced energy-efficiency benchmark," in Proc. ACM SIGMOD International Conference on Management of Data, 2007.

[2] E. Spanjer. "Enterprise SSD - the next killer app," http://www.flashmemorysummit.com/English/Collaterals/Presentations/2007/20070807_Issues_Spanjer.pdf, 2007.

[3] M. Wu and W. Zwaenepoel. "eNVy: a non-volatile, main memory storage system," in Proc. Architectural Support for Programming Languages and Operating Systems, pp. 86-97, 1994.

[4] H. Kim and S. Lee. "A new flash memory management for flash storage system," in Proc. Computer Software and Applications Conference, pp. 284-289, 1999.

[5] M. L. Chiang, P. C. H. Lee, and R. C. Chang. "Cleaning policies in mobile computers using flash memory," Journal of Systems and Software, vol. 48, no. 3, pp. 213-231, 1999.

[6] A. Ban. "Flash file system," United States Patent, no. 5,404,485, April, 1995.

[7] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. "A space-efficient flash translation layer for compact flash systems," IEEE Transactions on Consumer Electronics, vol. 48, no. 2, pp. 366-375, 2002.

[8] S. W. Lee, D. J. Park, T. S. Chung, W. K. Choi, D. H. Lee, S. W. Park, and H. J. Song. "A log buffer based flash translation layer using fully associative sector translation," ACM Transactions on Embedded Computing Systems, vol. 6, no. 3, 2007.

[9] J. U. Kang, H. Jo, J. S. Kim, and J. Lee. "A superblock-based flash translation layer for NAND flash memory," in Proc. International Conference on Embedded Software, pp. 161-170, 2006.

[10] D. P. Bovet and M. Cesati. "Understanding the linux kernel," O'Reilly, 3rd edition, 2005.