Exploiting Sequential and Temporal Localities to Improve Performance of NAND Flash-Based SSDs

SUNGJIN LEE, Massachusetts Institute of Technology
DONGKUN SHIN, Sungkyunkwan University
YOUNGJIN KIM, Ajou University
JIHONG KIM, Seoul National University

NAND flash-based Solid-State Drives (SSDs) are becoming a viable alternative as a secondary storage solution for many computing systems. Since the physical characteristics of NAND flash memory are different from conventional Hard-Disk Drives (HDDs), flash-based SSDs usually employ an intermediate software layer, called a Flash Translation Layer (FTL). The FTL runs several firmware algorithms for logical-to-physical mapping, I/O interleaving, garbage collection, wear-leveling, and so on. These FTL algorithms not only have a great effect on storage performance and lifetime, but also determine hardware cost and data integrity. In general, a hybrid FTL scheme has been widely used in mobile devices because it exhibits high performance and high data integrity at a low hardware cost. Recently, a demand-based FTL based on page-level mapping has been rapidly adopted in high-performance SSDs. The demand-based FTL more effectively exploits the device-level parallelism than the hybrid FTL and requires a small amount of memory by keeping only popular mapping entries in DRAM. Because of this caching mechanism, however, the demand-based FTL is not robust enough for power failures and requires extra reads to fetch missing mapping entries from NAND flash. In this article, we propose a new flash translation layer called LAST++. The proposed LAST++ scheme is based on the hybrid FTL, thus it has the inherent benefits of the hybrid FTL, including low resource requirements, strong robustness for power failures, and high read performance. By effectively exploiting the locality of I/O references, LAST++ increases device-level parallelism and reduces garbage collection overheads. This leads to a great improvement of I/O performance and makes it possible to overcome the limitations of the hybrid FTL. Our experimental results show that LAST++ outperforms the demand-based FTL by 27% for writes and 7% for reads, on average, while offering higher robustness against sudden power failures. LAST++ also improves write performance by 39%, on average, over the existing hybrid FTL.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; B.3.2 [**Design Styles**]: Mass Storage

General Terms: NAND Flash Memory, Solid-State Drives, Storage Systems

Additional Key Words and Phrases: Flash translation layer, address mapping, garbage collection

15

This work was supported by the National Research Foundation of Korea (NRF) grant (NRF-2013R1A6A3A03063762). The work of Jihong Kim was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2013R1A2A2A01068260). The ICT at Seoul National University and IDEC provided research facilities for this study.
Authors' addresses: S. Lee, the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA; email: chamdoo@csail.mit.edu; S. Lee's current address is Department of Computer Science and Information Engineering, Inha University, Incheon, Republic of Korea; email: sungjin.lee@inha.ac.kr; D. Shin, College of Information & Communication Engineering, Sungkyunkwan University, Suwon-si, Gyeonggi-do, Republic of Korea; email: dongkun@skku.edu; Y.-J. Kim, Department of Electrical and Computer Engineering, Ajou University, Republic of Korea; email: youngkim@ajou.ac.kr; J. Kim, Seoul National University, Republic of Korea; email: jihong@davinci.snu.ac.kr.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2016 ACM 1553-3077/2016/05-ART15 $15.00
DOI: http://dx.doi.org/10.1145/2905054

ACM Reference Format:
Sungjin Lee, Dongkun Shin, Youngjin Kim, and Jihong Kim. 2016. Exploiting sequential and temporal localities to improve performance of NAND flash-based SSDs. ACM Trans. Storage 12, 3, Article 15 (May 2016), 39 pages.
DOI: http://dx.doi.org/10.1145/2905054

1. INTRODUCTION

NAND flash memory has been widely used as storage media for mobile embedded systems, such as MP3 players and mobile phones, because of its low-power consumption, nonvolatility, high performance, and high mobility [Lawton 2006]. With continuing improvements in both the capacity and the price of NAND flash memory, NAND flash-based Solid-State Drives (SSDs) are increasingly popular in general-purpose computing markets. For example, many laptop and desktop PC vendors have replaced Hard Disk Drives (HDDs) with NAND flash-based SSDs. Enterprise systems are employing more flash-based SSDs to improve storage performance and energy efficiency.

The physical structures and characteristics of NAND flash memory are different from those of traditional HDDs. NAND flash memory consists of multiple blocks, and each block is composed of multiple pages. A page is a unit of read and write (program) operations, and a block is a unit of erase operations. NAND flash memory does not support overwrite operations because of its write-once nature. To update data previously written to a specific page, a block with that page has to be erased first. The number of Program/Erasure (P/E) cycles allowed for each block is usually limited to several thousand cycles. To hide such physical characteristics and to provide a block device interface, an intermediate software layer, called a Flash Translation Layer (FTL), is used between a file system and NAND flash memory.

The FTL is responsible for several functions that have a great effect on hardware resources, performance, lifetime, and data integrity. The FTL maps logical addresses from a file system to physical addresses in NAND flash. This mapping function of the FTL helps us to avoid the write-once nature of NAND flash. However, it often requires lots of DRAM because it has to maintain a logical-to-physical mapping table. Moreover, since logical-to-physical mapping decides the place to which incoming pages are written, it has a huge influence on the exploitation of device-level parallelism. Garbage collection is one of the important functions. Logical-to-physical mapping inevitably creates invalid pages in NAND flash. The garbage collection of the FTL reclaims wasted space occupied by invalid pages and supplies new free space for future writes. The FTL selects a block with invalid pages and erases the block after copying valid pages to a free block. All I/O activities associated with garbage collection are extra overheads, so they must be minimized for better I/O performance. Because of the limited P/E cycles of blocks, the FTL must support wear-leveling that prolongs the overall lifetime of NAND flash by evenly distributing the number of P/E cycles across flash blocks. In addition to hardware resources, performance, and lifetime, the FTL has a high effect on the robustness of a storage device against sudden power failures and system crashes. The FTL not only manages important mapping information, but also performs several management operations. These functions of the FTL are completely hidden behind the block I/O interface, making it difficult for the OS to ensure data integrity in cases of sudden power failures [Zheng et al. 2013; Moon et al. 2010]. The improper design of the FTL thus results in permanent data loss and/or requires a significant amount of time for system recovery.

A hybrid FTL and a demand-based FTL have been widely used for many flash storage systems. As its name implies, the hybrid FTL uses a hybrid mapping approach that combines page- and block-level mapping. The main advantage of the hybrid FTL is that it requires a small amount of DRAM space for logical-to-physical mapping, exhibiting fairly good performance. It also offers high data integrity for sudden power failures or system crashes. This hybrid mapping, however, is less efficient than

fine-grain mapping (e.g., page-level mapping) for exploiting highly parallelized storage architectures and also often incurs high garbage collection overheads. Unlike the hybrid FTL, the demand-based FTL is based on pure page-level mapping. By keeping only popular mapping entries in a small DRAM cache, it reduces memory requirements for logical-to-physical mapping. Based on fine-grain mapping, the demand-based FTL can maximally exploit device-level parallelism and considerably improve garbage collection efficiency, resulting in higher performance than the hybrid FTL. Because of its caching mechanism, however, the demand-based FTL is vulnerable to power failures. Moreover, whenever a mapping entry is not available in the DRAM cache, DFTL has to read the entry from NAND flash before servicing a read request, which results in degradation of read performance.

In this article, we propose a new FTL scheme, called *LAST++*, which addresses the shortcomings of two representative FTL designs (i.e., the hybrid and demand-based FTLs). LAST++ is based on the hybrid FTL; therefore, it can enjoy the inherent benefits of the hybrid FTL, such as low resource requirements, strong robustness for power failures, and high read performance. At the same time, LAST++ is designed to overcome the problems of hybrid FTLs, achieving better I/O parallelism and low garbage collection overheads. The key contributions of LAST++ are as follows:

—Efficient exploitation of localities of I/O references is the main novelty of LAST++. LAST++ considers two kinds of localities, temporal and sequential, which are typically observed in a storage device. I/O requests with different localities are isolated into different types of flash blocks: sequential and random log blocks. This separation of incoming write requests not only increases device-level parallelism, but also improves overall garbage collection efficiency. Data destined to random log blocks are also differently managed depending on their temporal locality, which helps us to further reduce garbage collection costs.
—In order to effectively handle data that have neither sequential nor temporal locality (which is commonly called *cold data*), LAST++ supports a background garbage collection technique which hides garbage collection overheads for cold data from end-users. In particular, LAST++ selects a victim block that contains only cold data to prevent lifetime degradation from premature garbage collection.
—LAST employs a simple yet efficient recovery scheme that keeps logical-to-physical mapping information in reserved pages of flash blocks. This recovery scheme is not only easily combined with hybrid FTL architectures, but also supports quick recovery time even when an SSD capacity is huge.
—We developed a trace-driven FTL simulator and carried out a series of evaluations using several workloads to evaluate LAST++. We compared LAST++ with the demand-based FTL scheme [Gupta et al. 2009] and several hybrid FTL schemes [Kim et al. 2002; Lee et al. 2007; Kang et al. 2006]. Our experimental results showed that LAST++ outperformed the demand-based FTL: It improved write performance by 27% and read performance by 7%, respectively, while providing higher robustness against sudden power failures. LAST++ also improved write response times and storage lifetimes by 39% and 40%, on average, over other hybrid FTLs.

The rest of this article is organized as follows. In Section 2, we give a brief description of the FTL. We explain well-known FTL schemes in Section 3. Section 4 explains the details of the proposed LAST++ scheme. Experimental results are presented in Section 5. Finally, Section 6 concludes with a summary and directions for future work.

2. BACKGROUND

In this section, we first introduce the basics of the FTL, including the hybrid and demand-based FTLs, especially focusing on their pros and cons in terms of resource requirements, performance, I/O parallelism, and data integrity.

2.1. Flash Translation Layer (FTL)

Generally, FTL schemes can be classified into two groups depending on the granularity of address mapping: page-level and block-level FTL schemes. In the page-level FTL scheme [Kim and Lee 1999; Chiang and Chang 1999], logical pages from the file system can be mapped to any physical pages in NAND flash. The page-level FTL exhibits excellent garbage collection efficiency and maximally exploits the inherent parallelism of high-performance SSDs equipped with multiple buses. Because of its huge mapping table size, however, the page-level FTL is impractical for real-world products. In the block-level FTL scheme [Ban 1995], a logical block is mapped to a physical block, and a page offset within a block is always fixed. By using coarse-grain mapping, the block-level FTL reduces the size of a mapping table significantly. Keeping the offset of a page in a block, however, incurs lots of page copies whenever overwrites occur. To update the data of a page in a block, for example, valid pages in that block as well as new data have to be written to another free block. The original block must be erased for future use. This not only increases the number of extra page copies, but also shortens the lifetime of NAND flash. To overcome these disadvantages, hybrid and demand-based FTLs are proposed.

2.2. Hybrid FTL

The hybrid FTL is a well-known alternative to the block- and page-level FTLs [Lee et al. 2008; Kim et al. 2002; Lee et al. 2007; Kang et al. 2006]. Even though many hybrid FTLs have been proposed, their overall architectures are similar. The hybrid FTL divides NAND flash blocks into *data blocks* and *log blocks*. Data blocks represent an ordinary storage space and are managed by block-level mapping. Log blocks are an invisible storage space for logging newly updated data. Unlike data blocks, log blocks are managed by page-level mapping. In the hybrid FTL, only a small number of blocks are used as log blocks. Therefore, the size of a page-level mapping table for managing log blocks is small. The hybrid FTL appends newly updated data to pages in log blocks, invalidating pages in data blocks that contain the old version of data. This helps us to avoid lots of page copies to maintain the block-level mapping information of data blocks. Once free space in log blocks is exhausted, however, the hybrid FTL has to create free log blocks by flushing valid data in log blocks to data blocks. This operation is called a *merge operation* because valid pages in log and data blocks are merged into new data blocks.

Figure 1 illustrates three types of the merge operations: *switch merge*, *partial merge*, and *full merge* operations. We assume that a block is composed of four pages. A white box represents a page with up-to-date data, whereas a shaded box is a page with obsolete data. The former is called a valid page and the latter an invalid page. A number inside a box denotes a Logical Page Number (LPN) from the file system. The switch merge is the cheapest merge operation. As shown in Figure 1(a), the FTL simply erases the data block only with invalid pages and changes the log block to the new data block: it requires only one block erasure with no page copies. The switch merge is performed only when all the pages in the data block are updated *sequentially*, starting from the first logical page (i.e., the page 0 in Figure 1(a)) to the last logical one (i.e., the page 3). The partial merge is similar to the switch merge, but it requires extra page copies from the data block to the log block, as depicted in Figure 1(b). After all the valid pages are copied (i.e., the page 3 in Figure 1(b)), the FTL performs the switch merge. The partial merge is typically observed when *semi-sequential* writes occur that are sequential but not long enough to fill up the entire block.

The full merge is the most expensive operation and is typically observed when logical pages are *randomly* updated. Figure 1(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 two free blocks and copies all the valid pages from LB_0, LB_1, DB_0, and DB_1 to the free blocks. The data blocks DB_0

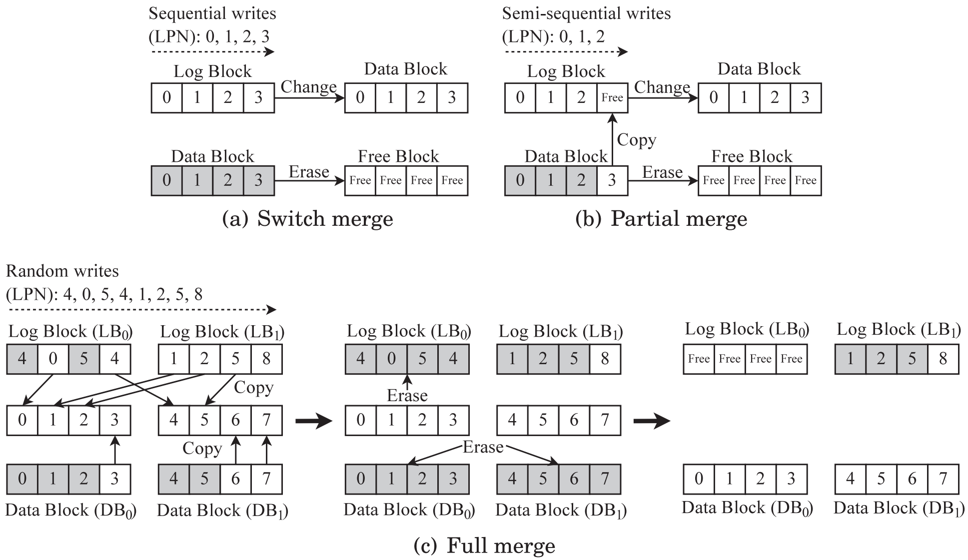


Fig. 1. Three types of merge operations.

and DB_1 are called *associated data blocks* of LB_0 because they have the invalid pages for the valid pages in the victim block LB_0 (i.e., the valid pages 0 and 4 in Figure 1(c)). The number of the associated data blocks can be increased up to the number of pages per block. After copying all the valid pages, the free block becomes the new data block, and DB_0, DB_1, and LB_0 are erased. As a result, the FTL gets one free block after the full merge. The full merge requires many extra copies and block erasures. In this example, eight pages are copied and three blocks are erased. In particular, the number of associated data blocks of a log block, which we call *an association degree*, decides full merge costs [Lee et al. 2007; Cho et al. 2009].

The hybrid FTL has been widely used for mobile devices such as MP3 players and digital cameras. Many mobile applications mostly issue sequential writes for storing multimedia files, along with a small number of random writes for metadata. For this reason, cheap switch merges are frequently performed whereas full merges are rarely conducted. Another benefit of the hybrid FTL is great robustness against sudden power failures. The hybrid FTL stores logical-to-physical mapping information in dedicated blocks called *map blocks*. Map blocks keep track of physical locations of log and data blocks and are used to reconstruct mapping information at boot time. In the hybrid FTL, updates of map blocks are performed in a single atomic write operation. This assures that mapping information stored in map blocks is always consistent [Kim et al. 2002]. Moreover, physical locations of log and data blocks are only changed after a block merge operation is performed, so extra I/Os required to manage map blocks are very small [Kim et al. 2002].

In spite of these advantages, the hybrid FTL has serious limitations. First, the hybrid FTL exhibits low performance in general-purpose systems like desktop PCs and laptops. Unlike mobile systems, general-purpose systems run complicated applications that issue lots of random writes to SSDs. This results in a large number of full merge operations. Second, hybrid FTLs have been designed for single-channel SSDs. Thus, they do not effectively support recent high-performance SSDs with multiple channels. Even though there have been several efforts to use the hybrid FTL in multichannel SSDs [Shim et al. 2012; Park et al. 2009], they still exhibit limited performance because of a relatively low channel utilization and a high merge cost over fine-grain mapping FTLs like the page-level FTL.

2.3. Demand-based FTL

The demand-based FTL is based on the pure page-level FTL. This allows the demand-based FTL to effectively exploit device-level parallelism, exhibiting better performance than the hybrid FTL. Moreover, since full merge operations do not occur, the demand-based FTL yields smaller garbage collection overheads in comparison with the hybrid FTL. To reduce the DRAM requirement, it maintains popular mapping entries in DRAM. In-memory mapping entries are usually managed by an LRU-based replacement algorithm, and only nonpopular mapping entries are evicted to NAND flash. Evictions of nonpopular entries incur extra writes, but those do not seriously affect overall storage performance because of the relatively high write hit ratio of an LRU cache [Gupta et al. 2009].

Unfortunately, the demand-based FTL has some serious drawbacks. First, the demand-based FTL is not robust enough because important mapping entries maintained in DRAM are easily lost when power failures or system crashes happen. To recover from a crash, the entire NAND flash space has to be fully scanned, which inevitably takes a very long time. One feasible solution that reduces recovery time while assuring reasonable data integrity is to employ a method that stores changes of mapping information in NAND flash, for example, periodically writing mapping information to NAND flash. Even when sudden power failures occur, SSDs are brought back to a consistent state by reading the latest mapping information kept in NAND flash. This approach, however, not only incurs many extra writes, but also causes extra garbage collection overheads. In our observation, the demand-based FTL performs more poorly than the hybrid FTL even when a relatively loose consistency method is used. Second, the demand-based FTL usually exhibits slower read performance, which has a higher impact on end-users' experiences. To read a flash page whose mapping entry is not available in DRAM, the demand-based FTL has to read a mapping entry from NAND flash after evicting existing entries from DRAM. This incurs additional I/O operations, thus degrading read performance.

3. RELATED WORK

In this section, we first review well-known FTL schemes and explain enhancements over our previous study in this field.

Review of previous FTL schemes: Kim et al. proposed the first hybrid FTL scheme that uses Block Associative Sector Translation (BAST) [Kim et al. 2002]. In BAST, one data block is associated with one log block. If a page in a data block is overwritten, its new data are written to a log block that is mapped to that data block. The block merge is triggered when there is no free log block that accommodates a newly updated page. BAST exhibits efficient garbage collection for consumer devices where sequential writes are mainly observed. However, the space utilization of log blocks gets worse with random writes. This is because even a single page update of a data block requires a whole log block. When a large number of small random writes are issued from the file system, most log blocks are selected as victim blocks with only a small portion of blocks being utilized. This phenomenon is called a *log block thrashing problem* [Lee et al. 2007]. Since all underutilized log blocks have to be merged by full or partial merges, the merge cost is greatly increased.

To overcome this shortcoming of BAST, Fully Associative Sector Translation (FAST) [Lee et al. 2007] has been proposed. In FAST, one log block is shared by several data blocks: up-to-date pages are written to any log blocks regardless of their data blocks. The block merge is performed only when all available free pages in log blocks are exhausted. This approach effectively removes the block thrashing problem, increasing the garbage collection efficiency for random writes. The problem of FAST is its

expensive full merge cost. One log block is associated with several data blocks in FAST, so as the association degree between log and data blocks increases, the cost of full merges linearly increases. For example, if a log block is associated with 4 data blocks (i.e., the association degree is 4) and the number of pages per block is 128, 512 pages have to be copied, and 5 blocks must be erased to create only one free block.

A SUPERBLOCK scheme [Kang et al. 2006] has been proposed to overcome the limitations of both BAST and FAST. Similar to FAST, SUPERBLOCK allows up-to-date pages from several data blocks to be stored in a log block, but it limits the maximum number of data blocks that can share the same log block. This not only reduces the overall full merge cost, but also mitigates the log-block thrashing problem. SUPERBLOCK employs page-level mapping inside a superblock, which is a set of consecutive logical blocks. Using this page-level mapping information, it separates hot pages from cold ones, further reducing the overall full merge cost. However, SUPERBLOCK does not effectively exploit temporal localities of I/O references because of its superblock-based address management. Another shortcoming of the SUPERBLOCK scheme is that page-level mapping information has to be stored in the spare area that is used for keeping error-correction codes.

Gupta et al. first presented a demand-based FTL scheme, called DFTL [Gupta et al. 2009]. DFTL is different from hybrid FTL in that it uses pure page-level mapping to manage the whole NAND flash. DFTL completely removes full merge operations, and, because of the flexibility of page-level mapping, it is also more suitable to exploit the I/O parallelism of multichannel SSDs. DFTL is also not affected by the block-thrashing problem. Despite all those benefits, the inability of DFTL to cope with power failures seriously limits its use in real-world applications. The penalty caused by slow read performance also could outweigh its advantages over the hybrid FTL.

Many other studies improve hybrid or demand-based FTLs. Lim et al. proposed an improved version of FAST, called FASTer, which exploited temporal localities of I/O references to reduce block merge costs [Lim et al. 2010]. Cho et al. presented an enhanced version of FAST, called KAST, for real-time systems. By limiting an association degree between log and data blocks [Cho et al. 2009], KAST guaranteed the worst-case merge time for real-time applications, thus providing nonfluctuating I/O performance. They did not, however, consider the efficient adoption of their FTLs in multichannel SSDs. Park et al. developed a convertible flash translation layer, CFTL, which improved the read performance of DFTL [Park et al. 2010]. By employing a small block-level mapping table (in addition to a page-level mapping table), CFTL handled random read-oriented workloads more effectively, showing better read performance than DFTL. Jiang et al. and Thontirawong et al. improved the mapping table management policy of DFTL to accomplish a high hit ratio with limited DRAM cache size by exploiting localities of workloads [Jiang et al. 2011; Thontirawong et al. 2014]. Similarly, Xu et al. presented a compact address mapping scheme for DFTL, which packed consecutive logical mapping entries into a single entry, thereby improving the effective capacity of DRAM cache [Xu et al. 2012]. Unfortunately, all those techniques focused on improving the performance of DFTL and did not take into account the data integrity issue with DFTL.

As mentioned earlier, many FTL schemes have been proposed, but almost all of them are based on hybrid or DFTL. For this reason, we compare the performance of LAST++ with well-known hybrid FTLs (including BAST, FAST, and SUPERBLOCK) and DFTL in this study.

Enhancements over our previous study: We showed that the exploitation of localities of I/O references could greatly improve the performance of the hybrid FTL scheme [Lee et al. 2008]. Our previous study has some serious limitations that we address in this work. First, our earlier version of LAST++ was designed for the single-channel architecture that was rarely used in recent high-performance SSDs. In this

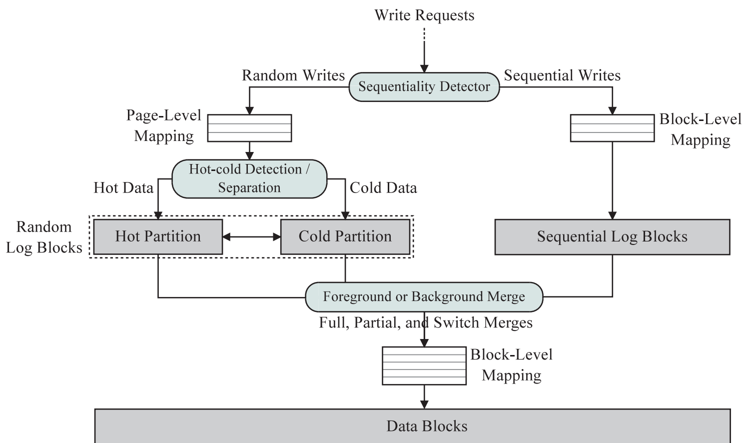


Fig. 2. The overall architecture of LAST++.

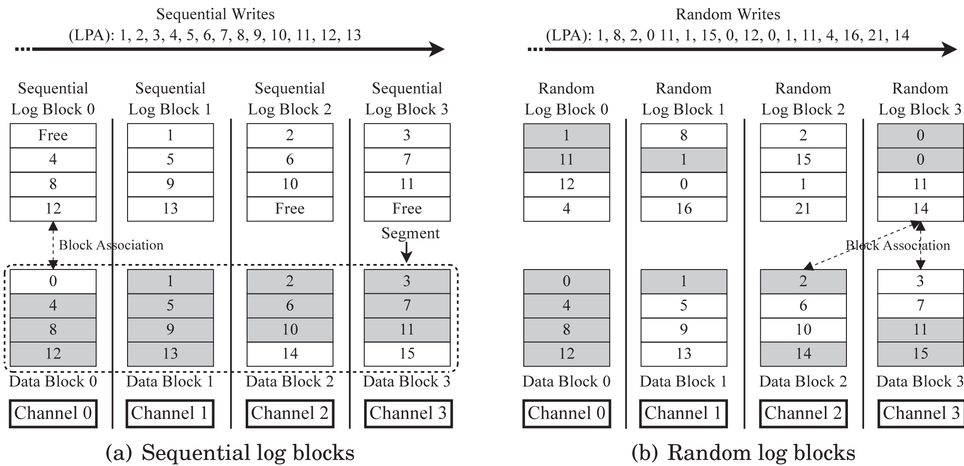
study, we improve the LAST++ scheme so that it effectively works for the multichannel architecture of modern SSDs. The organization of log blocks, logical-to-physical mapping algorithms, and block merge processes are modified to support multichannel SSDs. Second, by leveraging sequential and temporal localities of I/O references, the earlier version of LAST++ greatly reduced the cost of block merge operations. However, it still required high merge costs for cold data that was randomly written to log blocks. LAST++ resolves this problem by performing block merge operations in the background. To minimize the lifetime penalty caused by premature block merges, LAST++ carefully performs background merges only for cold data that remain valid for a long time. Third, data integrity (which is considered an important issue in designing the FTL) was not taken into account in our previous study. In this work, we develop a simple but efficient recovery scheme for LAST++. We also show that LAST++ is more durable than the demand-based FTLs, exhibiting better I/O performance. Finally, the previous version of LAST++ had several tunable parameters. Even though it would be beneficial to offering better performance, it increased the overall design complexity. All those tunable parameters are eliminated or simplified in our new design without greatly sacrificing performance.

4. LOCALITY-AWARE SECTOR TRANSLATION

LAST++ is designed to overcome the limitations of hybrid FTL while preserving its advantages over demand-based FTL. Localities of I/O references typically observed in general-purpose systems are a key consideration that LAST++ uses to resolve the limitations of the existing FTL solutions. In this section, we explain how LAST++ reorganizes log and data blocks of the hybrid FTL and how it manages mapping information to maximally exploit I/O localities taking full advantage of multichannel SSDs.

4.1. Overall Architecture

Figure 2 shows the overall architecture of LAST++. Similar to hybrid FTL, LAST++ divides all flash blocks into two groups: data blocks and log blocks. Data blocks are used as an ordinary storage space offered to end-users, whereas log blocks are used as a write buffer that temporarily stores incoming data. Log blocks are also divided into sequential and random log blocks. A sequentiality detector finds sequential write requests and sends them to sequential log blocks. Other requests are regarded as random and are destined for random log blocks. This separation of sequential writes



(a) Sequential log blocks (b) Random log blocks

Fig. 3. The management of sequential and random log blocks in LAST++.

from random ones avoids useless full merges for sequential requests. Random log blocks are divided into hot and cold partitions. Frequently updated data (i.e., hot data) are written to the hot partition, whereas infrequently updated data (i.e., cold data) are sent to the cold partition. This hot/cold separation further reduces full merge costs by reducing an association degree between log and data blocks. Data temporally stored in log blocks is evicted to data blocks using block merge operations (i.e., full, partial, and switch merges) in a foreground or background manner.

LAST++ manages data blocks and sequential log blocks using block-level mapping. Figure 3(a) shows an example of how LAST++ manages sequential log blocks. Unless otherwise stated, in this article, we assume that the number of channels is 4 and the number of pages per block is 4. We also assume that 13 pages (whose logical page addresses are 1, . . . , 13) are sequentially written. Unlike conventional block-level mapping, LAST++ *statically* maps adjacent logical pages to different channels and writes them together. For example, the logical page 1 is written to the sequential log block 1 in the channel 1, and, at the same time, the logical page 2 is written to the log block 2 in the channel 2. Since all write requests sent to sequential log blocks are sequential, this static mapping allows us to maximally exploit I/O parallelism. A sequential log block is associated with only one data block, and the page offsets of logical pages within those blocks are fixed. Three kinds of merge operations, including switch, partial, and full merges, occur in sequential log blocks, but cheap switch and partial merges are mostly performed.

In LAST++, data blocks are grouped by a *segment*. A segment is a fixed set of blocks, one per channel. For example, in Figure 3(a), the data blocks 0, 1, 2, and 3 on different channels are grouped into one segment. Logically consecutive pages are mapped to the same segment in a zigzag manner. For instance, in Figure 3(a), logical pages 0, 1, . . . , 15 belong to the same segment. The zigzag arrangement of logical pages in the segment enables us to perform partial and switch merges between data blocks and sequential log blocks.

Random log blocks are managed by page-level mapping. Figure 3(b) shows how LAST++ manages random log blocks when 16 pages are randomly written. Incoming write requests can be written to any locations, regardless of their logical page addresses, so LAST++ accomplishes high I/O parallelism even for random writes. For example, the logical pages 1, 8, 2, and 0 are written to four different channels simultaneously.

Similar to hybrid FTL, however, each random log block can be associated with the maximum N data blocks, where N is the number of pages per block. In Figure 3(b), the random log block 3 is associated with two data blocks, the data blocks 2 and 3. This results in expensive full merges.

4.2. Separation of Sequential Writes from Random Writes

LAST++ detects sequential and temporal localities of I/O requests and separates them into different types of log blocks (i.e., sequential and random log blocks). This separation is not only useful for reducing the number of full merges, but also is effective for preventing the block thrashing problem. If sequential writes are written to random log blocks, they must be evicted to data blocks by full merge operations. These full merge operations are actually useless: If they were written to sequential log blocks, switch and partial merges would be applied. On the other hand, if random writes are written to sequential log blocks, it causes the log-block thrashing problem like BAST: If they were written to random log blocks, the log-block thrashing would not occur.

Figure 4(a) illustrates write access patterns of a real user on a desktop computer where several applications like a web browser, a word processor, and games run. Microsoft's Windows XP with the NTFS file system is used for trace collection. Note that we borrowed this trace from the authors of Kang et al. [2006]. As labeled in Figure 4(a), write requests with temporal localities (labeled as ①) and sequential localities (labeled as ②) are commonly observed. There are also random writes that have no temporal and sequential localities (labeled as ③).

In LAST++, the sequential and temporal localities of I/O requests are detected by referring to the size of a write request that arrives at the device, which is simply called a *device-level request size*. Figure 4(b) shows a relationship between an update frequency and a device-level request size. The update frequency of the request of size S is the average number of updates over all the write requests of size S. The unit of a request size is a sector (512 bytes). The higher the update frequency, the higher a temporal locality is. As the size of a write request becomes shorter, a temporal locality is strongly observed. Figure 4(c) shows a relationship between the size of an application-level write request and the size of a device-level request. Here, the application-level write request is a write request issued by applications to the file system. As shown in Figure 4(c), short device-level writes mostly come from short application-level writes, and long device-level writes are likely to be a part of long sequential writes. We can thus safely assume that a short write request usually has a high temporal locality; on the other hand, a long write request has a relatively high sequential locality. Note that similar observations were also reported by Chang [2010].

Based on our observations in Figure 4, we propose a threshold-based locality detection policy that decides the types of localities of incoming write requests by comparing their sizes with a threshold value. If the size of a write request is larger than the threshold value, it is regarded as having a strong sequential locality and is sent to sequential log blocks. Otherwise, it is written to random log blocks. This threshold value must be carefully determined. If the threshold is too short, a large amount of small data is written to sequential log blocks, which causes the block thrashing problem. If the threshold is too long, many sequential writes are forwarded to random log blocks, and this increases the number of full merge operations.

As illustrated in Figure 4, device-level writes that are longer than 128 sectors belong to long application-level writes whose sizes are 2–3 MB on average. Other requests belong to short application-level requests whose sizes are several kilobytes (0.5K–400K). Considering that the segment size is several MB (e.g., 4 MB in our configuration with 128 4 KB pages and 8 channels), sending device-level requests larger than 128

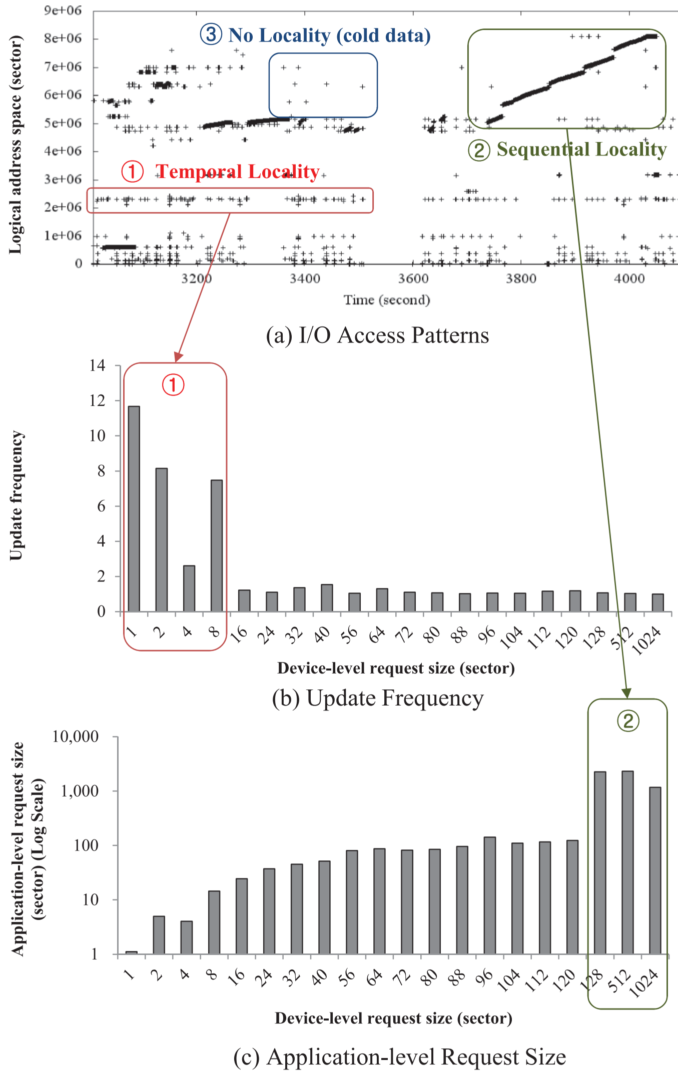


Fig. 4. The characteristics of write requests depending on their sizes.

sectors to sequential log blocks is the best choice. This decision is well supported by our experiments.

As astute readers may notice, LAST++ sends short or middle-sized random writes with no localities (see the label ③) to random log blocks. It prevents the log block thrashing problem, but since cold data are mixed with hot data in random log blocks, it results in many full merges. To more effectively deal with such cold data, LAST++ employs hot/cold partitioning and background merge techniques, which will be discussed in Section 4.4 in detail.

4.3. Management of Sequential Log Blocks

LAST++ sends write requests whose lengths are longer than the threshold value to sequential log blocks. Algorithm 1 shows how LAST++ handles write requests for sequential log blocks. When a new write request comes, LAST++ divides a write request

ALGORITHM 1: Write a Page to Sequential Log Blocks

```
   Input: Logical Page Address (LPA)
   Output: Boolean
 1 channel := getChannelNumber (LPA);                              // from Eq. (1)
 2 segment := getSegmentNumber (LPA);                              // from Eq. (1)
 3 page := getPageOffset (LPA);                                    // from Eq. (1)
 4 seg_entry := getEntryFromSeqBlockMappingTable (segment);        // See Figure 5
 5 if seg_entry.chls[channel] = NULL then
 6 |   phyBlkAddr := getFreeBlock (channel);
 7 |   if phyBlkAddr = NULL then
 8 |   |   doBlockMerge ();                    // trigger a block merge operation
 9 |   |   phyBlkAddr := getFreeBlock (channel);
10 |   end
11 |   seg_entry.chls[channel].phyBlkAddr := phyBlkAddr;   // initialize a mapping entry
12 |   seg_entry.chls[channel].PST := 0;
13 |   seg_entry.SeqID := SeqID++;
14 else
15 |   phyBlkAddr := seg_entry.chls[channel].phyBlkAddr;
16 end
17 if seg_entry.chls[channel].PST < (1<<page) then
18 |   writePage (channel, phyBlkAddr, page);
19 |   seg_entry.chls[channel].PST[page] := 1;
20 |   return TRUE;
21 end
22 return FALSE;                                // write a page to random log blocks
```

into several logical pages. For each logical page, LAST++ gets a channel number, a segment number, and a page offset using its Logical Page Number (LPA) as follows:

$$\begin{aligned} \text{Channel number} &= \text{LPA} \ \% \ \#\,\text{of channels} \\ \text{Segment number} &= \text{LPA}/\#\,\text{of pages per segment} \\ \text{Page offset} &= \text{LPA} \ \% \ \#\,\text{of pages per segment}/\#\,\text{of channels} \end{aligned} \tag{1}$$

where the number of pages per segment is 16 (i.e., 4 pages per block $\times$ 4 channels).

Using the segment number, LAST++ finds a segment entry in a block mapping table to which a logical page belongs. For a fast lookup, LAST++ uses a hash table (a more detailed explanation of the hash table is described in Section 4.6.1). Then, using the channel number, LAST++ finds a channel entry that points to the location of the physical block address in the corresponding channel. The channel entry also maintains a Page Status Table (PST) that keeps the status of pages in individual sequential log blocks. The size of the PST is N bits, where N is the number of pages per block. Each bit of the PST indicates whether the corresponding page is empty ('0') or has up-to-date data ('1'). A Seq-ID is a unique segment ID and increases by one when a new segment is allocated to the block mapping table. The Seq-ID is used to select a victim block later. Figure 5 depicts the block mapping table for sequential log blocks and shows how LAST++ handles write requests in sequential log blocks. This example illustrates the situation in the example of Figure 3(a), where the logical pages 4 and 5 arrive.

If the channel entry does not point to any physical block, it means that a physical block is not mapped yet. LAST++ has to obtain a free physical block from a free block list in the corresponding channel. If a free block is not available in the channel, LAST++ performs a merge operation to create free space. A more detailed explanation of block merge operations on sequential log blocks is discussed later. LAST++ then writes the new page to the page offset in the block. The corresponding position of the PST is

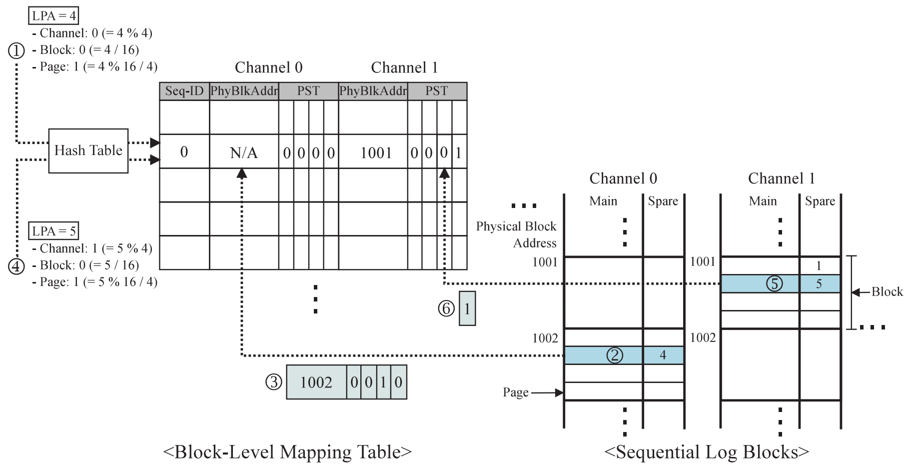


Fig. 5. An example of how LAST++ handles write requests in sequential log blocks. It shows the situation in the example of Figure 3(a), where the logical pages 4 and 5 arrive. ① A write request for the logical page 4 arrives. LAST++ first obtains the channel number, the segment number, and the page offset using the LPA, which are 0, 0, and 1, respectively. The corresponding entry of the block-level mapping table does not point to any physical block. ② LAST++ gets a free physical block whose address is 1002 in the channel 0. Then, it writes the page data to the second page (whose page offset is 1) in the physical block. The logical page address is also written to the spare area. ③ LAST++ updates the mapping entry to point to the new physical block. ④ A write request for the logical page 5 arrives. The channel number, the segment number, and the page offset are 1, 0, and 1, respectively. The corresponding mapping entry is already mapped to the physical block 1001 in the channel 1. ⑤ LAST++ writes the page data to the second page in the physical block, along with its LPA. ⑥ Finally, the PST of the mapping entry is updated.

set to ‘1’ to indicate that the new page is written. If the channel entry points to a physical block, LAST++ sees if the new page can be written to the physical block. If the corresponding position of the PST is ‘0’ and its page offset is the highest, LAST++ writes the data to the page offset in the block. Otherwise, LAST++ sends it to random log blocks because there is no available free space in the block.

LAST++ maintains several sequential log blocks. Maintaining several log blocks not only avoids a number of premature partial merges, but also increases the chance of performing switch merges because it delays the invocation of merge operations until all the free space is used up. In particular, it is also useful to effectively handle multiple sequential write streams that are sent from several user applications simultaneously; LAST++ can accommodate multiple write streams in several sequential log blocks. Figure 6 shows how LAST++ handles write requests, especially when multiple sequential write streams arrive at the SSD.

When sequential log blocks are fully used and there is no free space to accommodate newly updated data, LAST++ triggers block merge operations. LAST++ selects the *least-recently allocated* segment as a victim using the Seq-ID. Then, it performs multiple block merges at once for all the sequential log blocks in the victim segment. For example, if there are four channels in the SSD, four sequential log blocks in different channels (of the same segment) are merged simultaneously. LAST++ spreads sequential writes over all the channels, so if free blocks in one channel are exhausted, free blocks in other channels are also exhausted soon. Furthermore, performing multiple block merges in different channels in parallel is more performance efficient than doing block merges separately because it can exploit parallelism of multiple channels. Figure 7 shows an example of block merges in sequential log blocks.

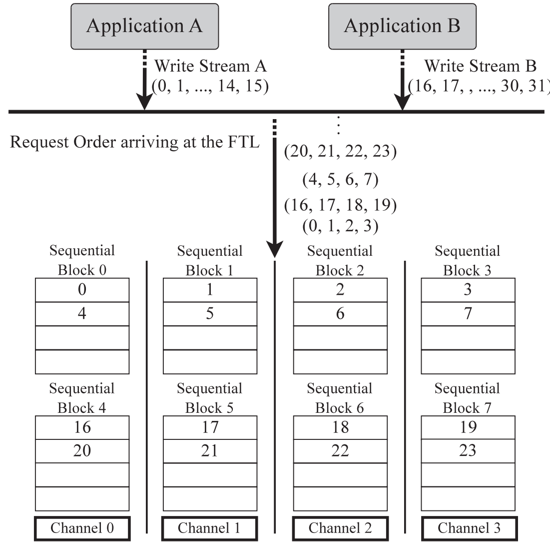


Fig. 6. An example of how LAST++ handles multiple write streams from several applications. Here, we assume that the number of channels is four and the number of pages per block is four. The sequentiality threshold is assumed to be four. There are two applications, `Applications A` and `B`, which issue two sequential write streams, `Write Streams A` and `B`, simultaneously. Each write stream is composed of 16 consecutive logical pages (e.g., `(0, 1,..., 14, 15)` for `Write Stream A` and `(16, 17,..., 30, 31)` for `Write Stream B`). Two write streams are mixed at the level of the FTL (at the level of the device) and arrive in the following order: `(0, 1, 2, 3), (16, 17, 18, 19),..., (28, 29, 30, 31)`. Since LAST++ maintains several sequential log blocks using block-level mapping, two different write streams are automatically isolated in different blocks according to Equation (1). If only one sequential segment is maintained, similar to FAST FTL (i.e., sequential log blocks 0, 1, 2, and 3 are only maintained), a partial merge occurs inevitably because there are no available log blocks to accommodate the pages from `Write Stream B` (i.e., `(16, 17, 18, 19)`).

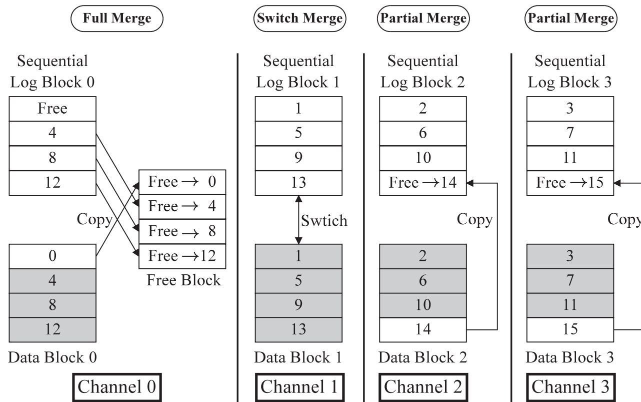


Fig. 7. An example of block merges in sequential log blocks. The initial status of sequential log blocks and data blocks is the same as Figure 3(a). Three different types of block merge operations occur. For channel 0, the full merge is required because the page 0 in data block 0 cannot be copied to sequential log block 0 due to the sequential program restriction. LAST++ allocates a new free block and copies all the valid pages from log and data blocks to the free block. The free block becomes the new data block 0, and the log block and the data block are erased. For the channel 1, the cheapest switch merge is applied. The sequential log block 1 becomes the new data block 1 and the old data block 1 is erased. For channels 2 and 3, partial merges are applied. After copying valid pages 14 and 15 to sequential log blocks 2 and 3, LAST++ erases data blocks 2 and 3. The sequential log blocks become the new data blocks.

ALGORITHM 2: Write a Page to Random Log Blocks

```
   Input: Logical Page Address (LPA)
   Output: NULL
 1 channel = getLRWChannel ();
 2 phyBlkAddr = getPhyBlockAddr (channel);
 3 if a free page is not available in phyBlkAddr then
 4 |  phyBlkAddr = getFreeBlock (channel);        // trigger a block merge operation
 5 |  if phyBlkAddr = NULL then
 6 |  |  doBlockMerge ();
 7 |  |  phyBlkAddr = getFreeBlock (channel);
 8 |  end
 9 end
10 page = getFreePageOffset (phyBlkAddr);
11 writePage (channel, phyBlkAddr, page);        // write the page to random log blocks
12 entry_old = NULL;
13 entry = getEntryFromPageMappingTable (LPA);                          // See Figure 8
14 if entry != NULL then
15 |  entry_old = entry;
16 end
17 entry.channel = channel;                                       // update the hash table
18 entry.phyBlkAddr = phyBlkAddr;
19 entry.page = page;
20 updateMergeCostTable (entry_old, entry);                   // update the merge cost table
```

LAST++ skips unprogrammed pages at the beginning of sequential log blocks. This results in full merges (e.g., see Channel 0 in Figure 7). In our observation, however, sending *sequential writes* to sequential log blocks is more beneficial than writing them to random log blocks even if unprogrammed pages are created. If a large amount of data belonging to sequential write streams is written to random log blocks, other pages (which are likely to update in the near future) must be evicted. Since sequential writes are not frequently updated, they stay in random log blocks for a long time, occupying precious log block space uselessly. Finally, several full merges have to be carried out when they are evicted from random log blocks. On the other hand, if sequential write streams are sent to sequential log blocks, they are separate from random log blocks and evicted to data blocks through full merges. Note that since only sequential writes are sent to sequential log blocks, the block thrashing problem does not occur.

4.4. Management of Random Log Blocks

All write requests that cannot be written to sequential log blocks are sent to random log blocks. Algorithm 2 shows how LAST++ handles write requests for random log blocks. LAST++ divides a write request into several logical pages and distributes them over different channels. To maximize I/O parallelism, LAST++ gets the random log block from the Least-Recently Written (LRW) channel. If the block has no free pages, LAST++ triggers a block merge to create free space in random log blocks. The page data are then written to the free page in the block. After writing the page, LAST++ has to update the page mapping table. To quickly find the physical location of the logical page, LAST++ uses a hash table that points to the corresponding entry of the page-level mapping table. Each mapping entry has a channel number, a block number, and a page offset. The entry also contains a 2-bit update counter that is increased by one whenever the logical page is overwritten (we explain this later in detail). If the entry

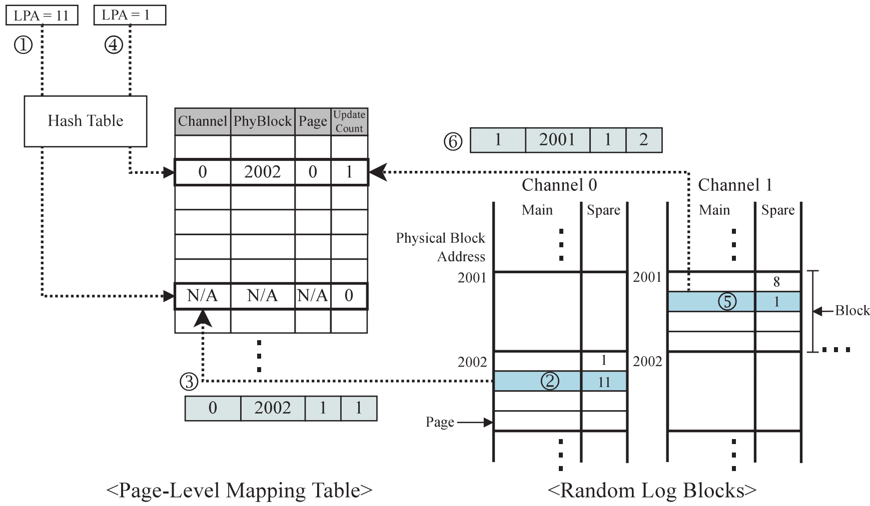


Fig. 8. An example of how LAST++ handles write requests in random log blocks. It shows the situation in the example of Figure 3(b), where logical pages 11 and 1 are written. ① A write request for logical page 11 arrives. Channel 0 is the Least-Recently Written (LRW) channel, and physical block 2002 is being used as a random log block in the channel 0. The mapping entry does not point to any physical location because page 11 was not written before. ② LAST++ writes page 11 to the second page of physical block 2002 in channel 0. The logical page number is also written to the spare area. ③ The mapping entry is updated to point to the physical location. ④ A write request for logical page 1 arrives. Channel 1 is the LRW channel, and physical block 2001 is being used as a random log block. The mapping entry points to the physical location (i.e., channel 0, block 2002, and page 0) where logical page 1 was previously written. ⑤ LAST++ writes page 1 to the second page of block 2001 in channel 1, along with its LPA. ⑥ Finally, the mapping entry is changed to point to the new physical location.

exists in the mapping table, LAST++ keeps the location of the old page to invalidate it later. Figure 8 illustrates the situation in the example of Figure 3(b), where logical pages 11 and 1 are sent to random log blocks after pages 1, 8, 2, and 0 are written.

To keep track of valid and invalid pages in random log blocks, LAST++ uses a merge-cost table. The merge-cost table maintains association degrees between random log blocks and data blocks. When a merge operation is triggered, LAST++ uses the merge-cost table to select a victim log block associated with the smallest number of data blocks. Note that choosing a victim block in this way is not new and has been used by Kang et al. [2006], Lee et al. [2008], and Cho et al. [2009]. The entry of the merge-cost table corresponds to each random log block. It contains a set of data blocks associated with a random log block and the number of valid pages of data blocks stored in the random log block. If the logical page is newly written to a random log block, it has a new associated data block or the number of valid pages of the corresponding data block increases by one. If a logical page becomes invalid in random log blocks, the number of valid pages of the corresponding data block decreases by one. If it reaches 0, that data block is removed from the entry and the number of associated data blocks decreases by one.

Figure 9(a) is an example of the merge-cost table corresponding to the random log blocks in Figure 3(b). The maximum number of data blocks that can be associated with a random log block is decided by the number of pages per block. Since the number of pages per block is assumed to be 4, the number of entries for data blocks is 4. In practice, a block has 128 or 256 pages, so the merge-cost table requires a large DRAM space, and the time taken to search the table could be so high. To avoid this, LAST++

Random Log Block #	The number of data blocks	A set of data blocks (data block #, # of valid pages)							
0	1	0	2	N/A	N/A	N/A	N/A	N/A	N/A
1	2	0	2	4	1	N/A	N/A	N/A	N/A
2	4	2	1	3	1	1	1	5	1
3	2	3	1	2	1	N/A	N/A	N/A	N/A

(a) A merge-cost table

Random Log Block #		Overflow flag				
0	1	0	0	2	N/A	N/A
1	2	0	0	2	4	1
2	2	1	2	1	3	1
3	2	0	3	1	2	1

(b) A reduced merge-cost table

Fig. 9. Examples of two different types of merge-cost tables that correspond to the random log blocks in the example of Figure 3(b). Note that data blocks 4 and 5 are not shown in Figure 3(b).

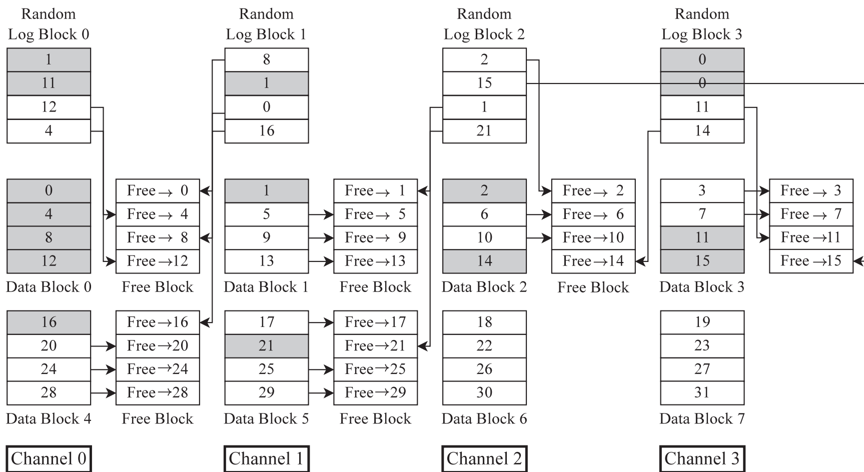


Fig. 10. An example of block merge operations in random log blocks. The initial status of the random log blocks is the same as Figure 3(b). Here, we assume that random log blocks 0, 1, 2, and 3 are selected as victim blocks, and they are associated with six data blocks: the data blocks 0, 1, 2, 3, 4, and 5. LAST++ first allocates six free blocks and copies all valid pages from the log and data blocks to the free blocks. Then, LAST++ erases 10 blocks (4 victim log blocks and 6 data blocks), and the free blocks become the new data blocks. As a result, LAST++ gets four free blocks for individual channels.

uses a reduced merge-cost table that maintains the limited number of associated data blocks for individual random log blocks. Instead, LAST++ adds a one-bit overflow flag to each log block. The maximum number of data blocks is set to 32 for NAND flash with 128 pages per block. If associated data blocks become larger than 32, the overflow flag is set to '1' to indicate that the random log blocks have more than 32 data blocks. If the number of the associated data blocks is reduced to 31, LAST++ still maintains the overflow flag as '1', indicating that it could be associated with more than 31 data blocks. When choosing a victim, LAST++ preferentially chooses a log block with the overflow flag of '0' if there are log blocks associated with the same number of data blocks. Figure 9(b) shows the example of the reduced merge-cost table for Figure 3(b) when the number of associated data blocks is limited to 2.

When available free space in random log blocks is exhausted, LAST++ triggers full merges to create free space. Figure 10 shows the overall steps of a full merge in the example of Figure 3(b). The full merge in LAST++ is similar to that in the existing hybrid FTL, except that it has to perform multiple full merges to create free log blocks in all channels. LAST++ chooses victim blocks and copies valid pages from both random log blocks and data blocks to free blocks. The cost of full merges is much expensive

than that of partial or switch merges. In particular, since LAST++ performs several full merges at once, it incurs a lot of page copies, thus degrading the overall SSD performance. In the example of Figure 10, six data blocks are associated with victim log blocks, so LAST++ has to copy 24 pages and erase 10 blocks.

To reduce full merge costs, LAST++ employs two strategies: *log-block partitioning* and *log-block replacement* techniques. The log-block partitioning technique divides random log blocks into two partitions, a hot partition and a cold partition, and writes incoming pages to different partitions depending on their localities. This separation of hot pages from cold creates many blocks with no valid pages in the hot partition, thus lowering overall full merge costs. The log-block partitioning technique can be more effective when it is combined with the log-block replacement technique. The log-block replacement technique selects a victim block in a way that minimizes full merge costs, and, at the same time, it dynamically resizes the hot and cold partitions so that the hot partition contains enough hot pages adapting to changing workloads.

4.4.1. Log-Block Partitioning Technique. In our observation, a large number of invalid pages occupy random log blocks, and many of them originate from hot pages whose data are updated frequently. Invalid pages are distributed into several log blocks, so random log blocks have both invalid and valid pages. This results in full merges that incur many live page copies. The log-block partitioning technique addresses this problem by partitioning random log blocks into hot and cold partitions. This ensures that a large number of *dead blocks* holding only invalid pages are created in the hot partition. The full merges of dead blocks do not require any page copies. By aggressively evicting dead blocks from the partition, the overall full merge cost is greatly lowered. In addition, this also makes cold pages stay longer in the cold partition, giving more chances for cold pages to be invalid before they are chosen as a victim block.

To detect hot pages, LAST++ uses a 2Q-like approach [Johnson and Shasha 1994]. LAST++ initially writes incoming pages to the cold partition. Then, if a page in the cold partition is updated, the up-to-date data of that page are sent to the hot partition. Once a page is written to the hot partition, it is regarded as a hot page until it is evicted to a data block. Sending all the pages updated in the cold partition to the hot partition, however, often makes a wrong decision because infrequently updated pages are also considered hot pages. To avoid this, LAST++ refers to the update frequency of a newly updated page. Only cold pages that are updated more than four times in the cold partition are sent to the hot partition. To monitor the update frequency of pages, LAST++ uses the 2-bit update flag in the page-level mapping table. Once the update flag reaches 3 (i.e., 11 binary), the logical page is regarded as hot and is sent to the hot partition. If the hot page is evicted from the hot partition, the corresponding mapping entry is reset. If the same logical page is written to random log blocks again, it starts again with the update flag of 0 in the cold partition. This helps LAST++ to adapt to the changing locality.

4.4.2. Log-Block Replacement Technique. The random log-block replacement policy is proposed to provide an intelligent victim block selection. To properly resize the partitions according to input write traffic, LAST++ adjusts the partition sizes while doing log-block replacement. The log-block replacement is composed of three steps: (i) victim partition selection, (ii) victim block selection, and (iii) partition resizing, and it operates differently depending on which partition requires free space to write incoming data. If free space in the hot partition is exhausted, LAST++ first sees if there is a dead block in the hot partition. If so, LAST++ chooses a dead block as a victim log block from the hot partition. The victim block is erased and is inserted into a free block list. LAST++ gets a new free block from the free block list, assigning it into the hot partition. The sizes of the partitions are not changed. If there are no dead blocks in the hot partition,

LAST++ picks up a victim from the cold partition. To reduce full merge costs, LAST++ selects a block with the smallest association degree by referring to the merge-cost table and performs a full merge. The freed block is inserted into the free block list. LAST++ gets a new free block and assigns it to the hot partition. The size of the hot partition is thus increased by one. This increase is necessary: If there are no dead blocks in the hot partition, it means that its size is not large enough to create dead blocks.

LAST++ attempts to select a victim block from the hot partition even when the cold partition requires more free space. If a dead block exists in the hot partition, LAST++ selects it as a victim and erases it to create a free block with no live page copies. Then, LAST++ increases the cold partition by assigning a new free block and writes incoming data to the newly assigned free block. The existence of dead blocks in the hot partition means that it is large enough to contain hot pages, so the decrease of the hot partition (or the increase of the cold partition) is a reasonable choice. On the other hand, if there are no dead blocks in the hot partition, LAST++ performs full merges in the cold partition to create new free space. As expected, a block with the smallest association degree is chosen as a victim. Since a free block created in the cold partition is reassigned to the cold partition, there are no changes in the sizes of the partitions.

LAST++ chooses only a dead block as a victim from the hot partition. This is effective in reducing full merge costs. However, this makes cold pages (which were previously hot but are now not hot) stay in the partition forever, occupying precious log blocks uselessly. To expel those pages, once the hot partition reaches its maximum size, LAST++ selects a victim from the hot partition even if there are no dead blocks. The maximum hot partition size is set proportional to the number of hot pages in random log blocks. For example, if the number of valid pages in random log blocks is 10 and hot pages are 5, the maximum hot partition size is 0.5 * the number of random log blocks. The overall steps of log-block replacement are described in the flowchart of Figure 11.

4.5. Background Merge for Cold Partition

By leveraging sequential and temporal localities, LAST++ mitigates the high merge cost problem in the hybrid FTL. However, cold pages staying in the cold partition have neither sequential nor temporal locality. For this reason, full merge operations in the cold partition often incur lots of live page copies that inevitably delay incoming write requests for a long time, degrading the experience of end-users. One feasible approach that resolve this problem is to perform block merges in background. In this article, we propose a new background merge policy for the proposed LAST++ scheme. Background garbage collection is not new and has been studied by other researchers [Lee et al. 2009; Park et al. 2014]. Our background merge policy is based on those previous studies, but it is different from earlier work in that it is designed to be more suitable for the architecture of LAST++.

As illustrated in Figure 12, LAST++ attempts to conduct full merges in advance during idle periods before expensive foreground full merges have to be performed. LAST++ uses a simple static timeout-based approach that triggers a background merge whenever observed idle time is longer than a fixed threshold value (which is denoted by TO in Figure 12). This simple threshold-based approach is known to be useful for flash storage since it does not incur a serious penalty by misprediction even with a relatively short threshold value [Lee et al. 2009]. Note that more advance triggering policies like a dynamic timeout-based policy also can be used with LAST++ [Park et al. 2014]. From the perspective of hiding full merge overheads from end-users, it would be reasonable to *aggressively* perform as many block merges as possible during available idle periods. This aggressive approach creates a lot of free blocks in the cold partition, so it can delay foreground merges as long as possible, thus minimizing user-perceived I/O latencies.

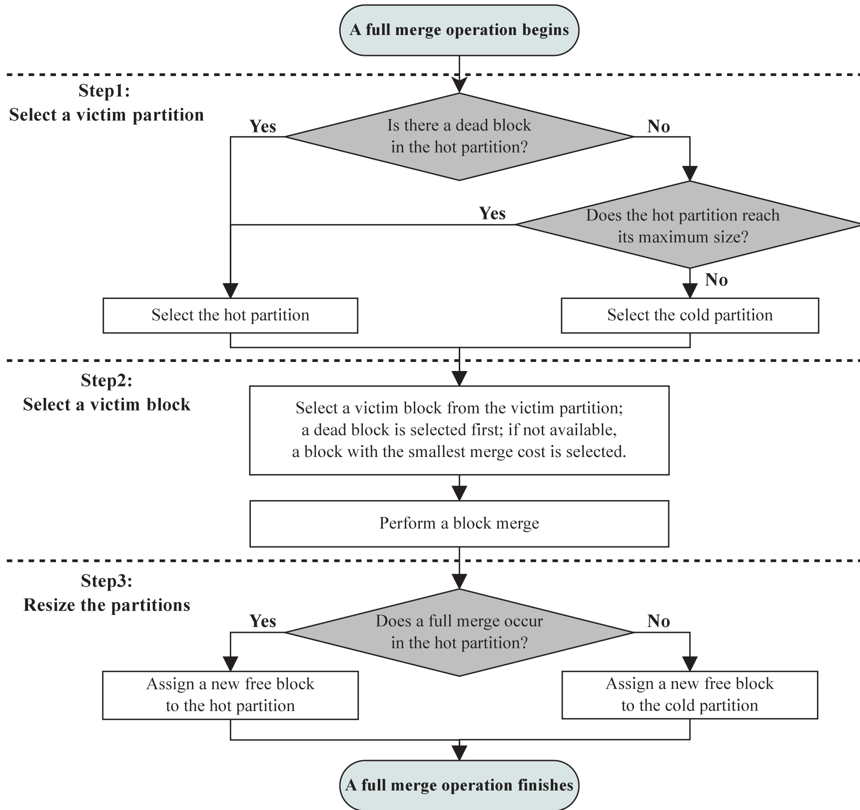
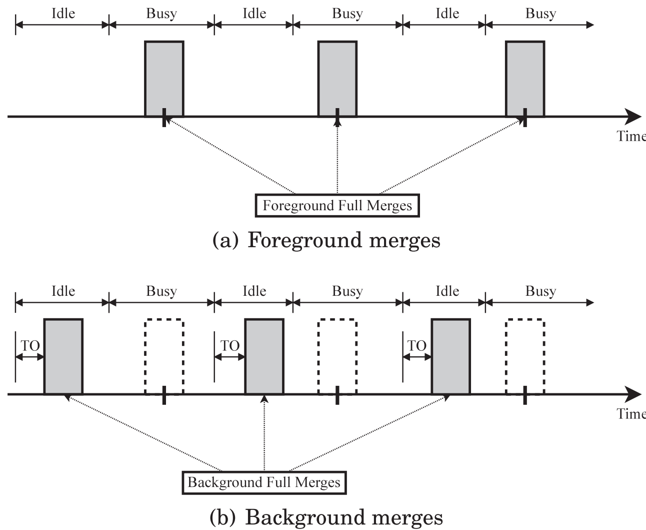


Fig. 11. The flowchart of the log-block replacement steps.

Unfortunately, this approach often incurs lots of premature block merges that move soon-to-be-obsolete pages to data blocks, thus degrading overall SSD lifetime.

To resolve this problem, we propose a new victim selection policy for background merges, one that performs background merges *conservatively* to achieve the same level of SSD lifetimes as foreground merges. LAST++ uses a new data structure called a *sorted-merge-cost list*. The sorted-merge-cost list is a list of random log blocks in the cold partition that are sorted in an ascending order of their merge costs. Figure 13 is an example of the sorted-merge-cost list. The topmost log block of the list is the cheapest one for block merges (e.g., 'Log Block A' in Figure 13). It also has an additional flag, called a latest update ID, which is a timestamp updated whenever the merge cost (i.e., an association degree) is reduced. A timestamp is always increasing, so log blocks with large update IDs are recently updated ones, meaning that their merge costs are reduced in the near past. This indirectly shows that log blocks with larger IDs have more hot pages than other blocks with smaller IDs. For example, in Figure 13, log blocks A, B, C, and E have larger IDs (i.e., 98), so they may have more hot pages than log block D with a smaller update ID (i.e., 95).

Figure 14 illustrates a simple example of how LAST++ selects a victim log block using the sorted-merge-cost list. We start with the same table shown in Figure 13. LAST++ maintains a timestamp, called a *merge sequence ID*, that increases by one whenever a foreground merge is invoked. For example, in Figure 14, there are four foreground merges, so the merge sequence ID is increased to 102 from 99. This sequence ID is used



(a) Foreground merges

(b) Background merges

Fig. 12. Examples of foreground and background merges. Foreground full block merges occur while writing incoming data to random log blocks. LAST++ has to perform full merges while suspending incoming writes to create sufficient free space in random log blocks. By conducting full merges in background, LAST++ can hide from end-users the overheads caused by full merges.

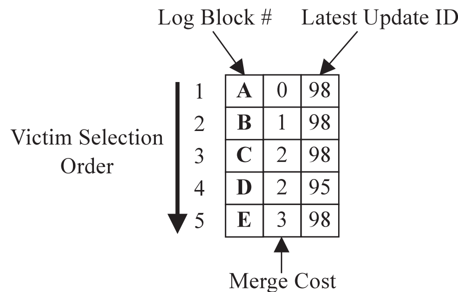


Fig. 13. An example of a sorted-merge-cost list.

as a timestamp for the sorted-merge-cost list. In the preceding example, the current sequence ID is 98. When the 99th full merge starts (i.e., the first full merge in Figure 14), LAST++ selects log block A as a victim because its merge cost is smallest. Then, log block A is removed from the list. The merge sequence ID is set to 99. Before the next foreground merge (i.e., 100th merge) is invoked, the merge costs of the log blocks B and E are decreased by one because some pages in B and E become obsolete. Thus, their latest update IDs are updated to 99. After the next foreground merge is called, LAST++ selects log block B as a victim. The current merge sequence ID now becomes 100. After finishing the 100th merge, a long idle period is detected, so a background merge is called. LAST++ first looks at the sorted-merge-cost list to select a victim. There are three candidates: log blocks C, D, and E. Selecting log block C is the cheapest way. However, since its merge cost was reduced after the latest block merge (i.e., after the merge sequence ID is 99), LAST++ expects that the merge cost of log block C is likely to reduce soon. For the same reason, log block E is not selected. On the other hand, the latest update ID of log block D is 95—log block D was not updated for the past five

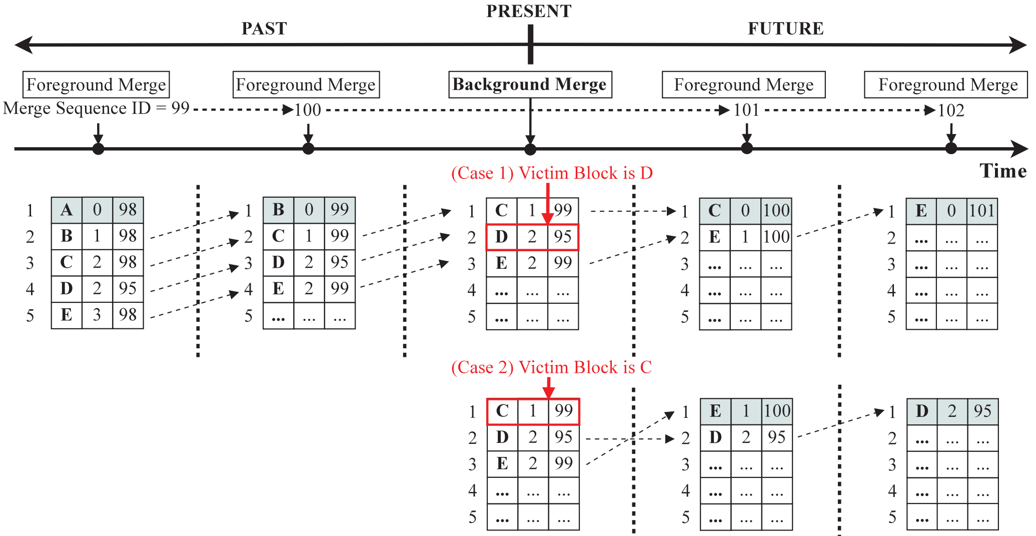


Fig. 14. An example of victim selection for background merges.

block merges. Therefore, it is unlikely that the merge cost of D is reduced in the near future. As a result, LAST++ selects log block D as victim.

Figure 14 shows two different cases where log block C or D is selected as a victim, respectively. If log block D is selected, only two log-block erasures are required in the future. This is because log blocks C and E become dead blocks when the foreground merges (i.e., 101th and 102th) are invoked. In the case where log block C is selected, LAST++ has to erase five blocks, including two log blocks and three data blocks, because of premature victim selection.

We now can generalize our victim selection policy for conservative background merges. LAST++ selects a victim log block using following two metrics: (i) a position in the sorted merge-cost list and (ii) a distance between the current merge sequence ID and log block's latest update ID. Whereas the position metric indicates how soon a log-block is selected as a victim for foreground merges in the future, the distance metric shows the likelihood of when the merge cost of a log block is reduced in the future. Back to the example in Figure 14, LAST++ gets two metrics using the sorted merge-cost table available when the background merge is called. For log block D, the position and the distance are 2 and 5 (i.e., 100–95), respectively. Using the position metric, LAST++ expects that log block D is selected as a victim when the 102th foreground merge is invoked (i.e., 100 + 2). Using the distance metric, LAST++ also expects that there will be no changes in the merge cost in the future five merges because the merge cost was not reduced for the past five merges. Based on this, LAST++ predicts that log block D will be selected as a victim and be merged by foreground garbage collection before its merge cost is reduced. Therefore, there will be no penalty to perform a background merge for log block D. On the other hand, in the case of log block E, its position and distance metrics are 3 and 1, respectively. These show that the merge cost of log block E will be reduced before the next foreground merge (i.e., 101th merge), but it will be selected as a victim much later (i.e., 103th merge). Therefore, selecting log block E could incur premature merges. In that sense, LAST++ only selects a log block whose distance metric is larger than its position metric. If there are no log blocks that meet this condition, it does not perform background merges.

Finally, we talk about the management issues of the sorted merge-cost table. For explanation purposes, we displayed the sorted merge-cost tables for the foreground merges in Figure 14. However, it is not necessary to use the sorted merge-cost tables for foreground merges because LAST++ can select the cheapest log block using the existing merge cost table. Instead, when the background merge is invoked, LAST++ creates the sorted merge-cost table on demand by referring to the existing merge cost table. To support background merges, we just need to add the latest update ID to the merge cost table. Building the sorted merge-cost table could take some time, but since it is built in background during idle times, it would not seriously affect I/O performance.

4.6. Computational Complexity and Memory Requirements

4.6.1. Computational Complexity. In order to quickly find the physical location of a logical page, LAST++ maintains two hash tables for sequential and random log blocks, respectively. Theoretically, the computational complexity of the hash table is O(1), but a large number of memory accesses could occur for a single hash lookup depending on the number of items in the hash. In the current implementation, LAST++ uses simple linear-probing to build the hash tables [Morris 1968]. According to Heileman and Luo [2005], linear-probing usually exhibits good performance with a load factor of less than 0.8. To maintain a reasonable load factor, LAST++ carefully decides the number of buckets in the hash tables. In case of sequential log blocks, the number of hash buckets is set to '# of sequential log blocks / # of channels $\times$ 2'. The number of entries in the block-level mapping table is the same as '# of sequential log blocks / # of channels', so LAST++ maintains the load factor of 0.5. For random log blocks, the number of buckets in the hash table is the same as the number of pages per random log blocks. Based on our observation, valid pages account for 50% of the total pages in random log blocks. Only 50% of hash buckets point to entries of the page-level mapping table, so the load factor of the hash table is maintained about 0.5. As will be discussed in the experimental section, the number of memory references per hash lookup is about 5.5.

The victim selection of LAST++ could incur computational overheads. As mentioned in Sections 4.3 and 4.4, LAST++ selects the least-recently allocated blocks as a victim block for sequential log blocks. LAST++ also chooses the block associated with the smallest data blocks as a victim for random log blocks. LAST++ maintains several thousands of flash blocks (e.g., 1,024 blocks) for sequential and random log blocks. For this reason, whenever it selects a victim block in sequential or random log blocks, LAST++ has to check all the entries in the block-level mapping table or the merge cost table. This problem can be overcome by applying some runtime optimizations. Unlike normal reads or writes that must be handled as soon as possible on demand from the host system, choosing a victim log block can be done in the background or processed in a pipelined manner with other operations. For example, LAST++ selects victim log blocks for future merge operations while performing the current merge operation. The cheapest block merge operation (e.g., a switch merge or a dead block merge) requires at least one block erasure that takes several milliseconds (e.g., 3.5ms), but checking all the entries (e.g., 1,024 entries when 1,024 log blocks are used) in both the block-level mapping table and the merge-cost table just requires several microseconds (e.g., 117 μs = 1,024 entries $\times$ 114 ns for a single DRAM latency [Leibowitz et al. 2010]). As a result, by overlapping computation and I/O operations, LAST++ completely hides the overheads for victim selection.

4.6.2. Memory Requirements. LAST++ maintains the block-level mapping table, the page-level mapping table, the merge-cost table, and two hash tables. The table sizes are different depending on the SSD capacity and the number of log blocks. We assume the SSD of 256 GB with 8 channels ($= 2^3$). The size of a page is 4 KB, and the number

Table I. A Summary of the Table Sizes of LAST++

Table	Sequential log blocks	Random log blocks	Merge-cost table	Data blocks	Hash tables
# of table entries	32	98,304	768	524,288	64 (sequential) 98,304 (random)
Entry size (bit)	1,157 bits	28 bits	854 bits	19 bits	64 bits
Table size (KB)	4.52 KB	336 KB	80 KB	1,216 KB	768.5 KB
Total table size (MB)	2.34 MB				

Table II. A Comparison of the Mapping Table Size of Different FTL Schemes

FTL Scheme	Block-level FTL	Page-level FTL	Hybrid FTL (BAST/FAST/SUPERBLOCK)	LAST++
Table Size	1.18 MB	208 MB	2.09 MB	2.34 MB

of pages per block is 128 (= 2^7). There are a total of 524,288 (= 2^{19}) blocks in the SSD and 65,536 (= 2^{16}) blocks per each channel. 1,024 blocks are used as log blocks: 256 for sequential log blocks and 769 for random log blocks. Table I summarizes the sizes of the tables in LAST++.

—*Sequential log blocks:* LAST++ maintains 32 total entries for the block-level mapping table for sequential log blocks (i.e., 256 sequential log blocks / 8 channels). As depicted in Figure 5, each segment entry is composed of the Seq-ID (5-bit), 8 physical block addresses belonging to different channels (16-bit each), and 8 page status tables (128-bit each). The block-level mapping table is thus 4.52 KB.
—*Random log blocks:* LAST++ maintains a total of 98,304 entries (i.e., 768 blocks × 128 pages per block) for the page-level mapping table. As depicted in Figure 8, each mapping entry has the channel number (3-bit), the block number in the channel (16-bit), the page offset (7-bit), and the update flag (2-bit). The page-level mapping table size becomes 336 KB.
—*Merge-cost table:* For the individual entries of the merge-cost table, LAST++ keeps the number of associated data blocks (5-bit) and the overflow bit (1-bit), as illustrated in Figure 9. Each entry of the merge-cost table also contains a list of 32 data blocks, each of which consists of the data block number (19-bit) and the number of valid pages (7-bit). To support background merges, each entry of the merge-cost table also has a 16-bit flag for the latest update ID to keep the merge sequence ID. There are 768 random log blocks, so the size of the merge-cost table is 80 KB.
—*Data blocks:* For the block-level mapping table for data blocks, LAST++ maintains the 19-bit data block number for individual entries. The number of entries is 524,288 (i.e., 524,288 flash blocks – 1,024 log blocks). Thus, its size is 1,216 KB.
—*Hash table:* Regarding the hash tables, finally, if a hash entry is 8 bytes, the sizes of the hash tables for the block-level mapping table and the page-level mapping table are 0.5 KB (= 64 × 8 bytes) and 768 KB (= 98,304 × 8 bytes), respectively.

As a result, the total amount of the memory space for the tables is 2.34 MB.

Table II compares the memory requirements of different FTL schemes, including page-level, block-level, BAST, FAST, SUPERBLOCK, and LAST++ FTLs. As expected, the block-level FTL requires the smallest memory space (1.18 MB), whereas the page-level FTL requires the largest memory space (208 MB). The mapping table size of BAST, FAST, and SUPERBLOCK FTLs is 2.09 MB. LAST++ requires 12% more memory space than other hybrid FTLs because it maintains the hash table to quickly find the physical location of a logical page. However, considering a high capacity of a recent DRAM chip (e.g., 32–128MB), a 12% larger mapping table of LAST++ would be not a serious obstacle to its adoption.

4.7. Reliability Issues

For fast startup, LAST++ stores the snapshot of block-level and page-level mapping in NAND flash when the SSD is normally turned off. This is a common way to support an instant booting and is widely used in most FTL schemes, including the hybrid and demand-based FTLs. Unfortunately, if system crashes or power failure occur, the snapshot information cannot be stored, so LAST++ has to recover the mapping information by scanning the NAND flash medium.

LAST++ maintains map blocks to keep track of physical locations of log and data blocks, and this allows LAST++ to recover the mapping information. The management of map blocks is exactly the same as in other hybrid FTLs [Kim et al. 2002; Lee et al. 2007]. The block-level mapping table for data blocks can be quickly built by reading map blocks. To construct the hash tables and the block- and page-level mapping tables, however, LAST++ has to scan entire pages in log blocks to read logical page addresses from spare areas. This could take very long time if the number of log blocks is large. For example, suppose that the capacity of the SSD is 256 GB and the size of log blocks is 25.6 GB (which is 10% of the total capacity). Further suppose that the maximum read bandwidth is 320 MB/s with eight NAND channels [Agrawal et al. 2008]. The recovery time taken to scan the entire log blocks is about 81.92 seconds (= 25.6 GB/320 MB/s).

To address this problem, we propose a simple recovery technique that keeps page-level mapping of individual log blocks in their last pages via a summary page. This allows us to quickly build a page-level mapping table by reading only one page per log block. LAST++ only needs to scan all the pages in a log block in the worst case where a summary page is not completely committed to that log block. Note that a similar scheme was introduced by Birrell et al. [2007] for page-level mapping. Back to the previous example: 52,429 log blocks are required for 25.6 GB. LAST++ needs to read 52,429 pages, which is 204.8 KB. Thus, the recovery time is about 0.64 seconds. The last pages of individual log blocks are reserved for summary pages, so the effective capacity of random log blocks is inevitably reduced by 1/128 (= 0.78%) if the number of pages per block is 128. This reserved log-block space is not so huge, so its effect on performance is negligible in our observation. We will show this in our experimental section.

4.8. Handling of Read Requests

The handling of read requests is straightforward in LAST++. For each page read request coming from the host system, LAST++ checks whether the page is stored in random log blocks or not by searching the hash table. LAST++ reads the page data from random log blocks if it is available. If random log blocks do not have the recent version of the page, LAST++ looks at the block-level mapping table to see if sequential log blocks have that page. If it is, LAST++ reads the data from sequential log blocks. If the page does not exist in random and sequential log blocks, the page in data blocks is transferred to the host system.

4.9. Wear-Leveling Issue

In LAST++, hot pages are kept in the hot partition, so random log blocks belonging to the hot partition are intensively erased. On the other hand, random log blocks in the cold partition are rarely overwritten; thus, their erasure counts become much smaller than those in the hot partition. The address mapping and garbage collection of LAST++ works independently of existing wear-leveling mechanisms. Therefore, this uneven wear problem can be resolved by employing well-known wear-leveling algorithms like hot/cold swap algorithms [Chang 2007]. It must be noted that inappropriate integration of LAST++ and wear-leveling algorithms could badly affect their performance. Thus, it

Table III. Key Parameters of NAND Flash Memory

NAND Flash Memory Organization	Block size	512 KB
	Page size	4 KB
	Number of pages per block	128
Operation Latency	Page read	50 usec
	Page write	900 usec
	Block erasure	3500 usec

Table IV. Descriptions of Benchmarks

Trace	Description	Write	Read	Duration
`Desktop1`	Collected from desktop/laptop PCs where several applications like editors, games, web browsers, and messengers ran.	6.1 GB	5.28 GB	8 hrs
`Desktop2`		3.5 GB	1.7 GB	9 hrs
`Laptop`		5.7 GB	7.02 GB	97 hrs
`Postmark`	Emulated the behaviors of mail and netnews services. 200K transactions were performed and 30K files with 4–16 KB were created.	6.1 GB	N/A	37 mins
`Iozone`	Performed writes/re-writes and reads/re-reads on a 1 GB file. The I/O flush was enabled, and the stripped access was disabled.	6.0 GB	N/A	73 mins
`Tiobench`	Created 1 GB files from eight threads that wrote 4K blocks randomly and sequentially.	1.2 GB	N/A	3 mins
`Bonnie++`	Performed different types of file system operations. Several files/directories were sequentially and randomly written.	1.0 GB	N/A	2 mins
`Financial1`	Collected from OLTP applications running at financial institutions.	12.5 GB	2.2 GB	10 hrs
`Proxy1`	Collected from web-proxy servers.	53.3 GB	99.8 GB	12 hrs
`Msnfs`	Collected from MSN storage file servers.	50.5 GB	110 GB	120 hrs

is necessary to investigate the impact of combining LAST++ and wear-level schemes on both lifetime and performance in detail. We leave this issue for future investigation.

5. EXPERIMENTAL RESULTS

5.1. Experimental Setting

To evaluate the performance of the proposed LAST++ scheme, we developed a trace-driven FTL simulator. We compared LAST++ with four existing FTL schemes: BAST [Kim et al. 2002], FAST [Lee et al. 2007], SUPERBLOCK [Kang et al. 2006], and DFTL [Gupta et al. 2009]. NAND flash parameters used in our simulation were based on Micron's MT29F16G08 NAND flash memory [Micron Technology Inc. 2012] and are listed in Table III.

Table IV summarizes 10 traces used for our evaluations. `Desktop1`, `Desktop2`, and `Laptop` were I/O traces collected from desktop and laptop PCs. Except for `Laptop`, which used the FAT32 file system, the NTFS file system was used to collect I/O traces. `Postmark`, `Iozone`, `Tiobench`, and `Bonnie++` were obtained while running well-known file-system benchmarks on Microsoft's Windows XP with the NTFS file system. `Financial1`, `Proxy1`, and `Msnfs` were taken from SNIA's trace repository [SNIA 2015].

The total capacity of the SSD was set to 256 GB (=2^{38} bytes), excluding log blocks for hybrid FTLs and an overprovisioning area for DFTL. To evaluate the effect of garbage collection algorithms on performance, the number of log blocks was set differently depending on the working-set size of benchmarks. For two small I/O traces, `Bonnie++` and `Tiobench`, we used 256 log blocks. For middle-sized traces, `Desktop1`, `Desktop2`, `Laptop`, `Postmark`, and `Iozone`, 1,024 log blocks were used. Because of a large working-set size, we used 4,096 log blocks for `Financial1` and `Proxy1`, while 16,384 log blocks

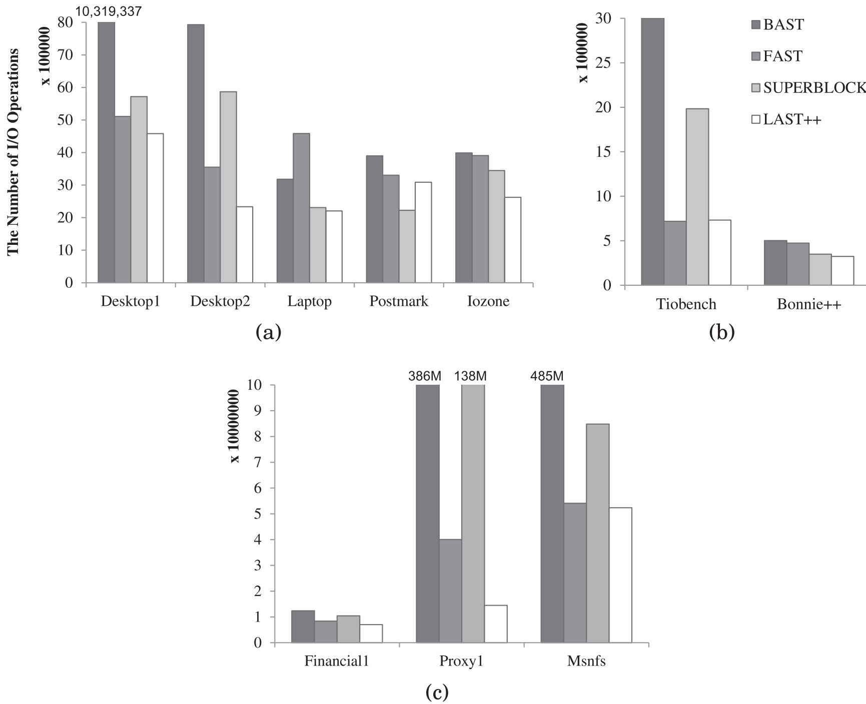


Fig. 15. A comparison of the number of I/O operations.

were assigned to `Msnfs`. For LAST++, 25% of the total log blocks were used for sequential log blocks, while other blocks were used for random log blocks. The number of channels was set to 8 by default. All data blocks were initially filled with valid pages to mimic aged SSDs.

BAST, FAST, and SUPERBLOCK FTLs were not designed for multichannel SSDs. For this reason, we used a design method, called FTL-MM, which enabled the hybrid FTL designed for single-channel SSDs to exploit the parallelism of multichannel SSDs [Shim et al. 2012]. In FTL-MM, individual flash chips were separately managed by independent instances of the single-channel–based hybrid FTL. To exploit I/O parallelism of multiple channels, FTL-MM distributed logically continuous pages over multiple chips. For DFTL, a page-level stripping policy was employed with a greedy garbage collection policy [Agrawal et al. 2008; Gupta et al. 2009]. Note that the DRAM cache size of DFTL was set the same as the DRAM requirement of LAST++.

5.2. Experimental Results with Hybrid FTLs

We first compare the performance of LAST++ with three hybrid FTL schemes: BAST, FAST, and SUPERBLOCK. We compare LAST++ with DFTL in Section 5.3 in detail. Figure 15 shows the number of I/O operations. LAST++ exhibits the best performance among all the FTL schemes; it reduces the number of I/O operations by 299%, 39%, and 70%, on average, over BAST, FAST, and SUPERBLOCK, respectively. BAST performs the largest I/O operations because of high merge costs, and this is mainly caused by the block thrashing problem. The log block utilization of BAST is 14%; on the other hand, LAST++ exhibits a log block utilization of 86%, the highest among all the FTL schemes (see Figure 19).

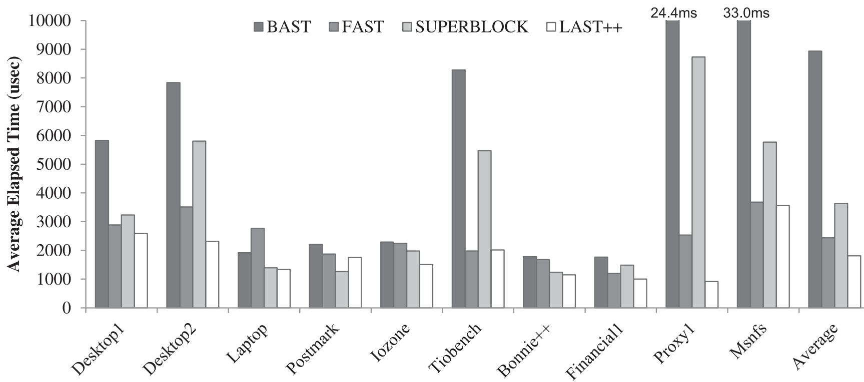
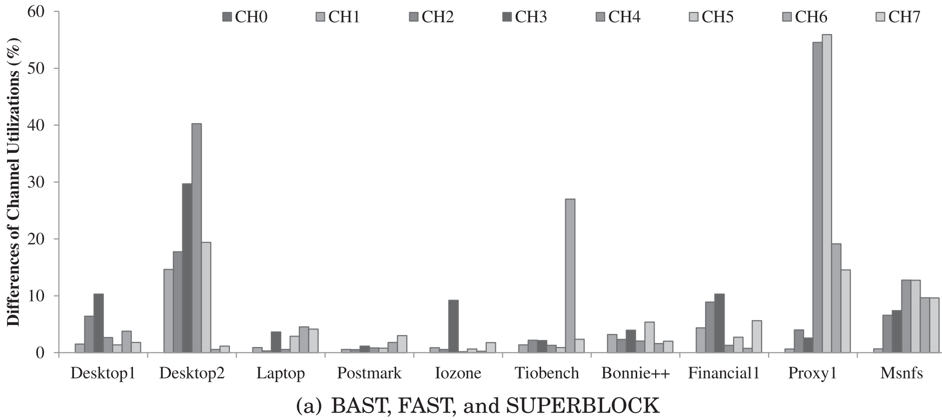


Fig. 16. Average elapsed time (μsec).

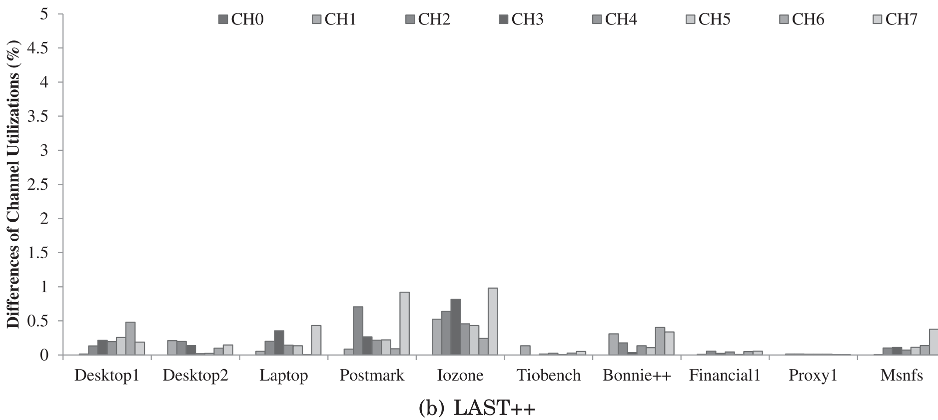
By allowing a single log block to be shared by several data blocks, FAST mitigates the block thrashing problem. However, it cannot outperform LAST++ for the following reasons. FAST maintains only one log block for sequential writes, which is called a sequential log block. In general-purpose systems, several sequential write streams are sent to SSDs simultaneously, competing for one sequential log block. For this reason, one sequential write stream often expels the other sequential streams stored in the sequential log block, thus incurring many partial merges. Unlike FAST, LAST++ maintains many sequential log blocks; thus, it can accommodate multiple sequential write streams so that they are evicted to data blocks by cheap switch merges. FAST often sends sequential writes to random log blocks, and they have to be evicted by full merges later. LAST++ sends only sequential writes to sequential log blocks, increasing the probability of performing partial or switch merges, which are much cheaper than full merges. Unlike FAST, that does not exploit the temporal locality of random writes, LAST++ increases the number of dead blocks by separating hot and cold pages in random log blocks.

Similar to FAST and LAST++, SUPERBLOCK allows several data blocks to share the same log block. To reduce the association degree, it limits the maximum number of data blocks that can be associated with a single log block. Even though it helps us to limit the maximum full merge costs, the block thrashing block problem cannot be completely avoided. For example, SUPERBLOCK performs worse than FAST for `Tiobench`, `Desktop1`, `Desktop2`, `Proxy1`, and `Msnfs`, where block thrashing is frequently observed. SUPERBLOCK cannot effectively reduce full merge costs, exhibiting a higher log-block association degree than LAST++ (see Table V). SUPERBLOCK attempts to separate hot pages from cold pages, but this hot/cold separation is only applied to pages in the same superblock because of its superblock-based mapping policy. Unlike SUPERBLOCK, LAST++ detects and separates hot and cold pages regardless of their locations in NAND flash. This allows LAST++ to generate a large number of dead blocks in the random log blocks, further reducing the overall association degree.

Figure 16 shows the elapsed time for writing a single page to the SSD. As shown, LAST++ exhibits the smallest elapsed time over all the FTL schemes; LAST++ outperforms BAST, FAST, and SUPERBLOCK by 255%, 41%, and 73%, respectively, on average. The overall elapsed time is highly related to the number of I/O operations depicted in Figure 15. As the cost of block merges increases, more valid pages have to be copied to free blocks before servicing incoming write requests from the host system. This inevitably increases the overall write response times.



(a) BAST, FAST, and SUPERBLOCK

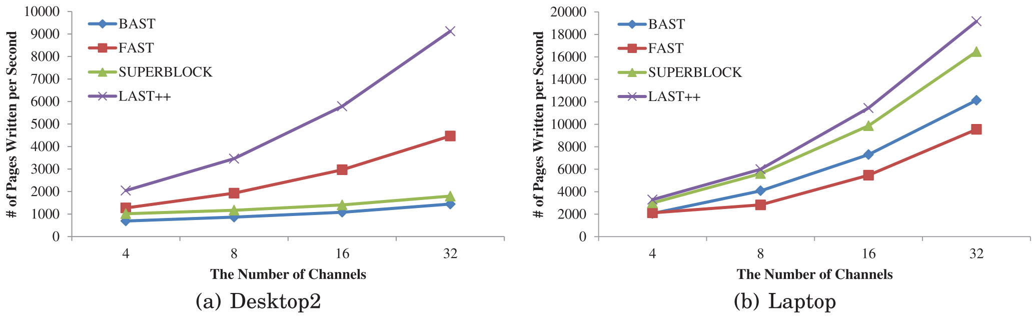


(b) LAST++

Fig. 17. The differences of the utilizations of eight channels (%).

Another key factor that highly affects write response times is channel utilization. Figure 17 shows the differences of channel utilizations among eight channels. We use channel 0 as a reference point, so its value is always 0. If the difference of a certain channel is 10%, 10% more or less requests are served in that channel than channel 0. BAST, FAST, and SUPERBLOCK use the same stripping policy proposed in FTL-MM [Shim et al. 2012], so they exhibit the same channel utilizations. As depicted in Figure 17, LAST++ shows much higher channel utilization than other FTL schemes. In the case of random log blocks, LAST++ fully utilizes the I/O parallelism of multiple channels because of flexible page-level mapping. Even though block-level mapping is used, LAST++ also exhibits high channel utilizations for sequential log blocks because only sequential write requests are sent to sequential log blocks. Unlike LAST++, other FTL schemes distribute incoming page writes across different channels according to their logical page addresses. For this reason, the overall channel utilization is decided by the patterns of incoming write requests.

Figure 18 shows how much the number of channels affects performance. We select two traces, `Desktop2` and `Laptop`, which show different channel utilizations. As illustrated in Figure 17(a), BAST, FAST, and SUPERBLOCK show a relatively even channel utilization for `Laptop`, but exhibit an uneven utilization for `Desktop2`. LAST achieves high utilizations for both. We measure the number of pages written per second while varying the number of channels from 4 to 32. As expected, the overall write



(a) Desktop2 (b) Laptop

Fig. 18. The effect of channel numbers on performance.

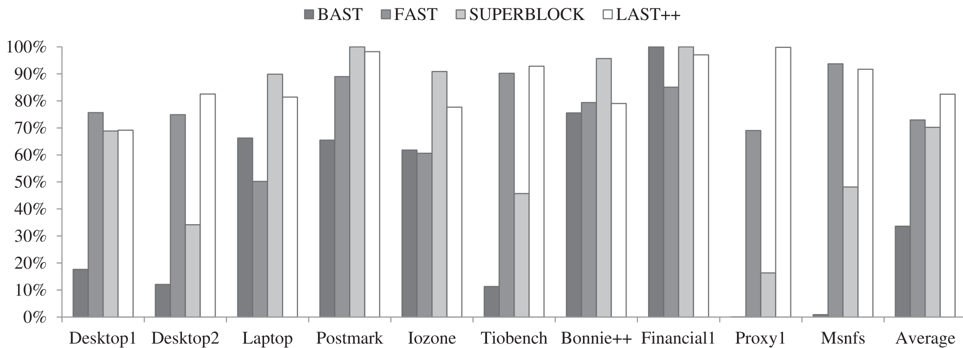


Fig. 19. Log block utilization (%).

performance is improved in proportion to the number of available channels. In the case of `Desktop2`, the performance improvements of BAST, FAST, and SUPERBLOCK are seriously limited because of their low channel utilizations. For `Laptop`, where BAST, FAST, and SUPERBLOCK show good utilization, the performance scales very well as the number of available channels increases. Regardless of the benchmarks, LAST++ shows good performance scalability.

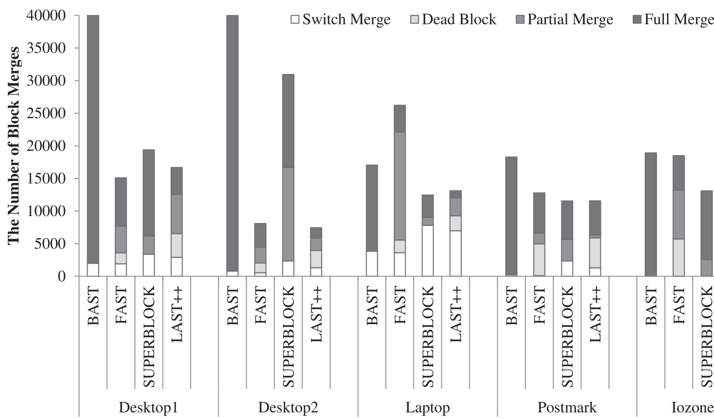
Figure 19 shows the log-block utilization of four different FTL schemes. As pointed out earlier, BAST exhibits the lowest utilization for all I/O traces. SUPERBLOCK also shows low utilizations for `Tiobench`, `Desktop1`, `Desktop2`, `Proxy1`, and `Msnfs`. FAST achieves a log-block utilization of 100% for random log blocks. However, because of frequent partial merges in the sequential log block, its overall block utilization is reduced to 73%. Similar to FAST, LAST++ also exhibits 100% utilization for random log blocks. By maintaining multiple sequential log blocks and sending only sequential writes to them, it prevents many sequential log blocks from being evicted to data blocks with a low utilization. For this reason, LAST++ shows the highest log-block utilization.

Table V compares the average association degrees during full merges for 10 I/O traces. As expected, LAST++ shows the smallest association degree among all the FTL schemes. This benefit mainly comes from a large number of dead blocks created in random log blocks. In LAST++, 70% of victim blocks are selected as dead blocks from random log blocks for full merges, and this reduces the overall association degree. Note that, for BAST, the association degree is always fixed to 1.

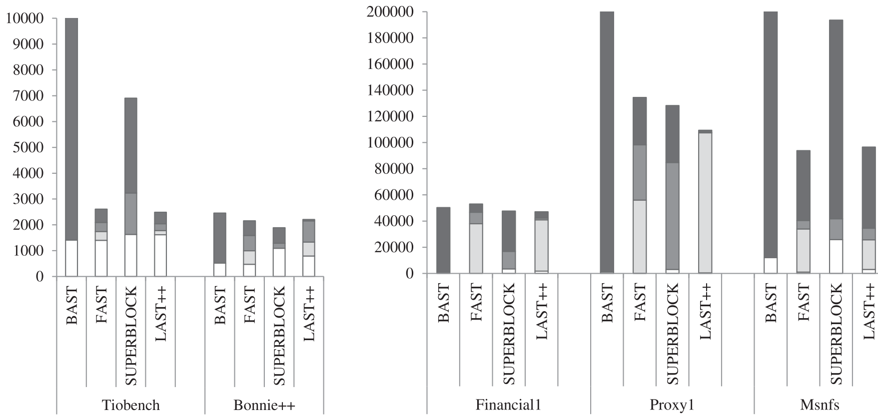
Figure 20 shows the number of block merges according to their types. BAST shows the largest number of block merges among all the FTL schemes because of block thrashing. For `Tiobench`, `Desktop1`, `Desktop2`, `Proxy1`, and `Msnfs`, SUPERBLOCK is also

Table V. A List of the Average Association Degrees During Full Merges

Trace	BAST	FAST	SUPERBLOCK	LAST++
Desktop1	1	2.6	2.6	2.5
Desktop2	1	3.7	1.8	2.4
Laptop	1	1.4	2.2	1.2
Postmark	1	1.1	2.0	1.0
Iozone	1	1.0	2.2	0.8
Tiobench	1	4.0	3.2	5.1
Bonnie++	1	1.2	2.1	0.3
Financial1	1	0.2	1.8	0.1
Proxy1	1	1.7	1.0	0.02
Msnfs	1	4.1	3.3	3.6
Average	1	1.5	2.1	0.76



(a)



(b) (c)

Fig. 20. The number of block merges according to their types.

affected by the block thrashing problem, incurring a larger number of block merges than FAST and LAST++. For Laptop, Postmark, Iozone, Bonnie++, and Financial1, SUPERBLOCK shows a smaller or similar number of block merges compared with LAST++. However, it cannot outperform LAST++ except for Postmark, as depicted

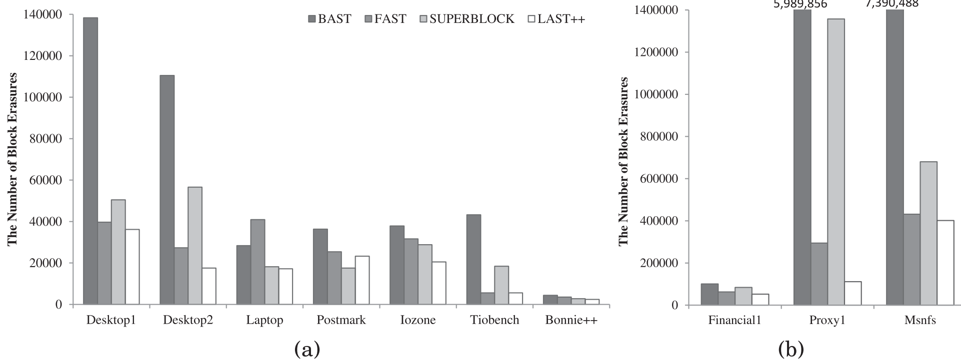


Fig. 21. The number of block erasures.

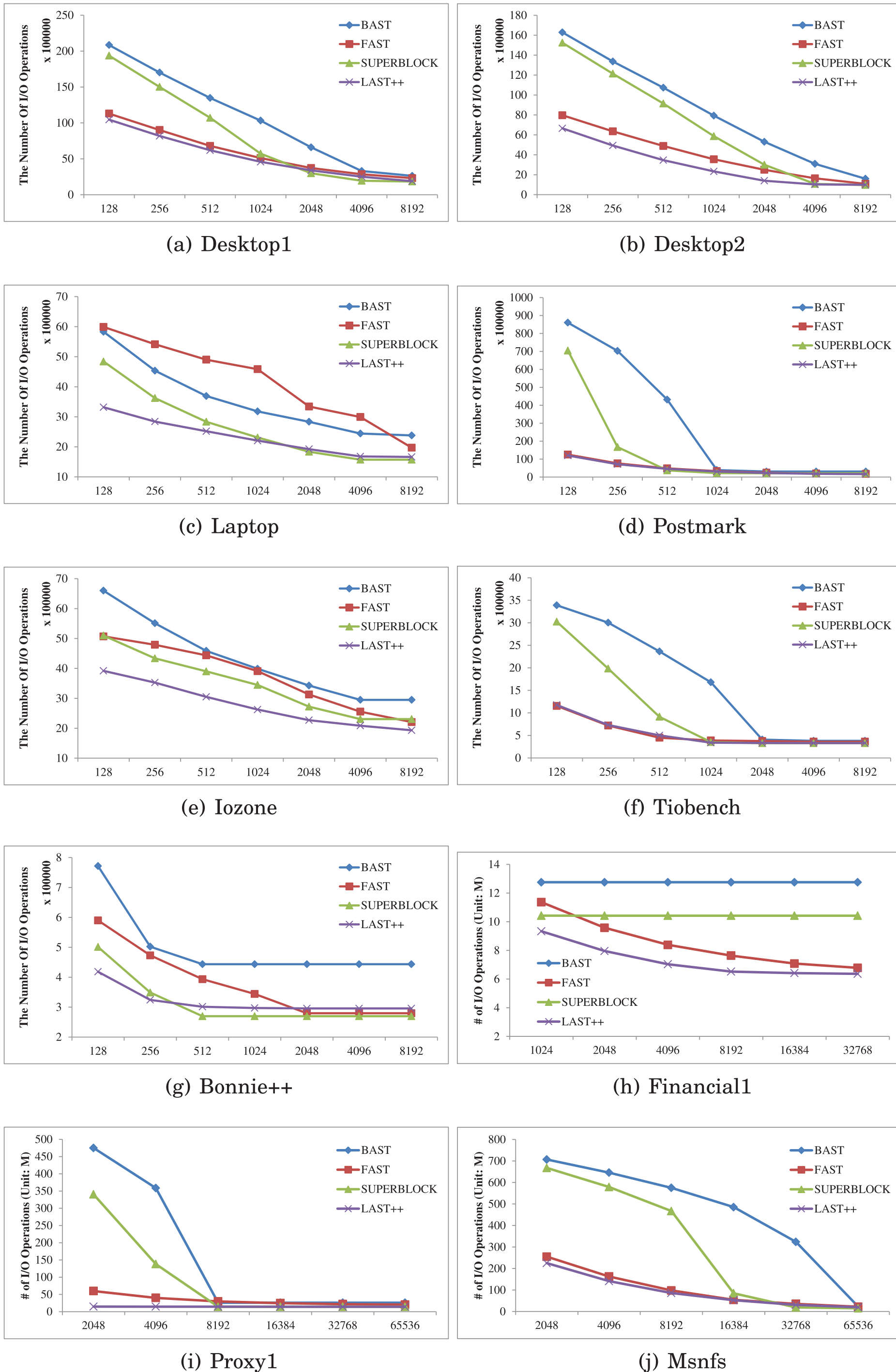
in Figure 15. This is because LAST++ performs more switch merges and dead-block merges with smaller full merges. Compared with FAST, LAST++ requires much smaller full merges. LAST++ generates a larger number of dead blocks than FAST by separating hot and cold pages in random log blocks. By sending only sequential writes to sequential log blocks, furthermore, the ratio of switch merges to total block merges is much higher than FAST.

Figure 21 shows the number of block erasures performed while running 10 I/O traces. The number of block erasures is closely related to the number of I/O operations depicted in Figure 15. LAST++ reduces the number of block erasures by 282%, 40%, and 51% over BAST, FAST, SUPERBLOCK, respectively. This means that LAST++ improves the lifetime of the SSD by the same amount.

We evaluate the effect of the number of log blocks on write performance. As shown in Figure 22, as the number of log blocks increases, the number of page writes decreases. With a larger number of log blocks, the FTL keeps more data in log blocks, which increases the probability that valid pages become invalid until they are evicted from log blocks. In particular, the performance of BAST and SUPERBLOCK greatly improves because the block thrashing problem disappears with a larger number of log blocks. However, regardless of the number of log blocks, LAST++ exhibits the best performance. Note that once the capacity of log blocks becomes larger than the working-set size of the benchmark (i.e., the amount of data written by the benchmark), all the FTL schemes exhibit similar performance because block merges rarely occur and normal I/O requests (sent from the host) become a dominant part of total I/O operations.

5.3. Experimental Results with DFTL

Unlike the hybrid FTL, DFTL could cause serious data integrity problems because it keeps logical-to-physical mapping information in DRAM all the time. Therefore, it is not practical to use DFTL directly without any methods that ensure data integrity. For this reason, we evaluate DFTL with four different data integrity methods, `NOFLUSH`, `PAGE`, `TIMEOUT`, and `REQ`. `NOFLUSH` is the same as the original DFTL scheme proposed in Gupta et al. [2009]; it does not write any mapping information to NAND flash until the DRAM cache becomes full and some mapping entries have to be evicted to NAND flash. `PAGE` writes a corresponding mapping entry to NAND flash after writing a single page. `PAGE` shows the strongest data integrity, but it incurs lots of extra writes to NAND flash. If K pages are newly written, additional K pages containing their mapping entries have to be written because the eviction of one mapping entry in DRAM requires one flash page write. `TIMEOUT` periodically writes dirty mapping entries to NAND flash. The timeout threshold is set to 30 seconds, similar to a policy used in the Linux kernel. `TIMEOUT`



(a) Desktop1 (b) Desktop2

(c) Laptop (d) Postmark

(e) Iozone (f) Tiobench

(g) Bonnie++ (h) Financial1

(i) Proxy1 (j) Msnfs

Fig. 22. The number of I/O operations with various log blocks.

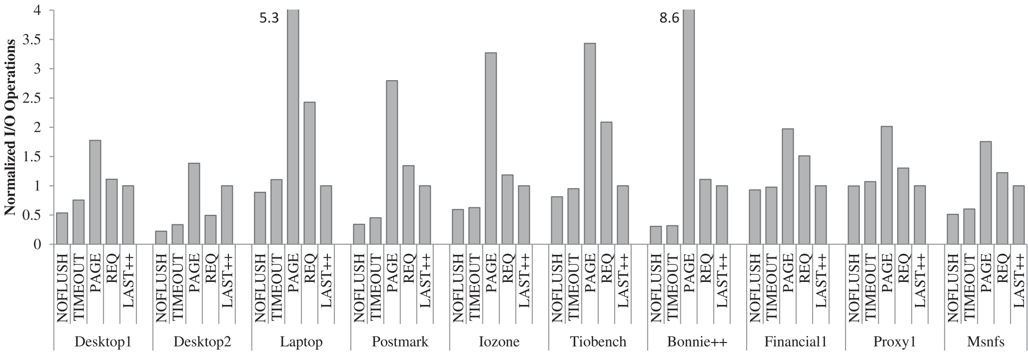


Fig. 23. A comparison of LAST++ with four different versions of DFTL: NOFLUSH, PAGE, TIMEOUT, and REQ. All results are normalized to LAST++.

is more durable than NOFLUSH, requiring smaller extra I/Os than PAGE, but it loses important mapping information if a power failure occurs between two flush periods. REQ writes mapping entries based on the unit of a write request. If a write request is composed of 512 pages, it writes all the pages to NAND flash, updating corresponding mapping entries in DRAM. Then, it writes the updated mapping entries to NAND flash. REQ not only guarantees the atomicity of a write request, but also reduces many extra writes over PAGE because it writes a bunch of updated mapping entries at once.

Figure 23 compares the performance of LAST++ with four different versions of DFTL. Experimental results are normalized to LAST++. All the experimental settings, such as the number of channels, are the same as those used in experiments with the hybrid FTL. As expected, NOFLUSH shows better performance than LAST++. However, when a sudden power failure occurs, NOFLUSH has to scan the entire NAND flash to reconstruct the page-level mapping table. PAGE shows the worst performance among all the FTLs because of lots of extra write traffic to NAND flash. TIMEOUT shows better performance than LAST++, except for Laptop. Compared with REQ, LAST++ exhibits better performance for all the benchmarks. According to our experimental results, REQ may be a feasible solution for DFTL because it exhibits relatively high performance with good data integrity. Considering that LAST++ outperforms REQ, offering the same level of data integrity as PAGE, LAST++ would be a better FTL solution in environments where high data integrity and quick recovery are required. Finally, our experiment results show that, even though DFTL is receiving lots of attention from academia because of its superb performance, it could be impractical or could perform more poorly than hybrid FTLs without a proper data integrity method. The development of a data integrity model suitable for DFTL is highly desirable.

Figure 24 shows the number of page read operations for DFTL and LAST++. For our evaluation, we choose six real-world traces—Desktop1, Desktop2, Laptop, Financial1, Proxy1, and Msnfs—because the I/O traces collected from micro-benchmarks do not contain read requests. LAST++ requires 2.34MB DRAM for the mapping table. For Desktop1, Desktop2, Laptop, and Financial1, we use the same size of cache (i.e., 2.34 MB) because they have relatively small read working-set sizes. However, this DRAM cache size is too small for Proxy1 and Msnfs, considering their large read working-set sizes. To prevent performance distortion by such a small cache size, we use a larger DRAM cache for Proxy1 and Msnfs, which is 32 MB. We roughly decide the cache size so that about the top 30% unique hot-mapping-entries could be kept in the DRAM; that is, mapping entries for frequently accessed data could stay in the DRAM cache. We employ the REQ method for DFTL.

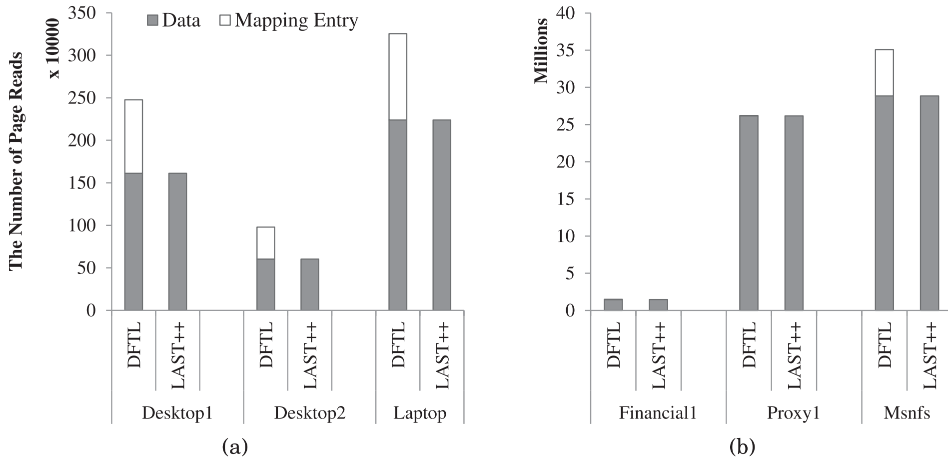


Fig. 24. The number of page read operations for DFTL and LAST++.

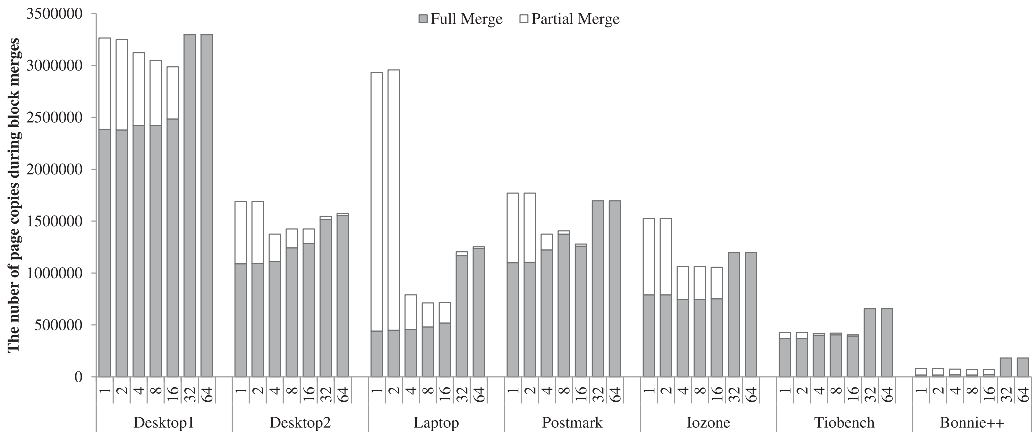


Fig. 25. The number of page copies during partial and full merges with various threshold values.

Unlike LAST++, that keeps the entire mapping entries in DRAM, DFTL holds only popular entries in DRAM. For this reason, DFTL often incurs extra page read operations to fetch mapping entries from NAND flash. The number of extra reads is different depending on the read access patterns of benchmarks, but it accounts for a relatively large proportion of the total read operations: 18–35%, except for `Financial1` and `Proxy1`. This inevitably increases overall read latencies that highly affect overall user-perceived I/O performance. As expected, LAST++ does not require any extra page reads. In the cases of `Financial1` and `Proxy1`, only a few reads for on-flash mapping entries are observed because of their high read localities.

5.4. Detailed Experiments with Various Design Parameters

We evaluate the performance of LAST++ in detail while changing several design parameters. We first evaluate the impact of a threshold value for sequentiality detection on performance. As depicted in Figure 25, when the threshold value is 16 pages, LAST++ shows the best performance. When the threshold value is small (e.g., 1–8 pages), a lot of random writes are sent to sequential log blocks. This incurs the block thrashing

S. Lee et al.

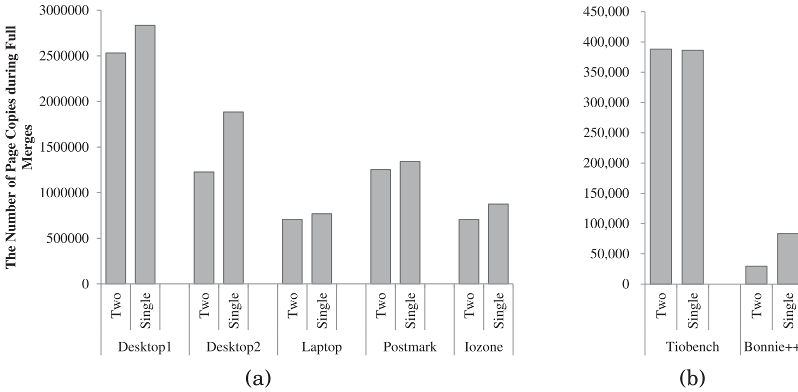


Fig. 26. The number of page copies with a single partition or two partitions.

Table VI. The Number of Memory References Per Hash Lookup

Desktop1	Desktop2	Laptop	Postmark	Iozone	Tiobench	Bonnie++	Average
6.8	5.4	6.6	6.8	6.5	3.9	2.8	5.5

problem and requires many partial merges. On the other hand, if the threshold value is large (e.g., 32–64 pages), many sequential writes are sent to random log blocks. This reduces the chance of switch merges while increasing full merge costs.

We evaluate the effect of the hot/cold separation in random log blocks by comparing the number of page copies of LAST++ with two partitions and with a single partition. As shown in Figure 26, LAST++ with two partitions reduces 25% of live page copies during full merges over LAST++ with a single partition. The effect of the hot/cold separation is quite effective for the I/O traces having high temporal locality (e.g., `Desktop1`, `Desktop2`, and `Bonnie++`. However, for I/O traces with low temporal locality, like `Tiobench`, its effect is very limited.

Reducing the searching cost of the mapping tables is also one of the important issues in designing LAST++. We measure how many memory references are required when LAST++ searches the physical location of a logical page. Table VI shows the number of memory references per hash lookup. As shown in this table, LAST++ requires 5.5 memory accesses per hash lookup, on average. This is very small compared to FAST FTL, which requires 65,536 accesses with a simple linear search.

To understand how the reduced merge table affects performance, we compare the performance of LAST++ with the reduced merge table and with the full-length merge table. Whereas the reduced merge table maintains only 32 entries for associated data blocks, the full-length merge table keeps 128 entries for data blocks. Figure 27 shows our experimental results. Even though the maximum number of data blocks in the merge table is limited to 32, the actual number of associated data blocks is much smaller than 32. For this reason, using the reduced merge table does not badly affect overall performance, incurring only 7% extra overheads for full merge operations.

We evaluate the effect of the background merge policy on the performance and lifetime of the SSD. Figure 28 shows our experimental results. We carried out a series of experiments with three different policies of LAST++: `FG`, `BG(AGGR)`, and `BG(CONS)`. `FG` is the LAST++ scheme with foreground merges. LAST++ with `BG(AGGR)` uses aggressive background merges that trigger full merges whenever idle times are available.

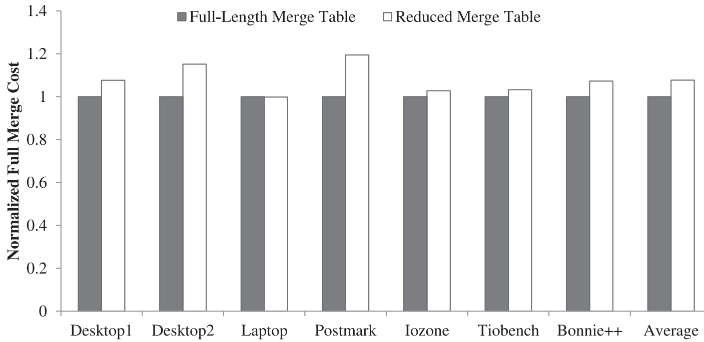
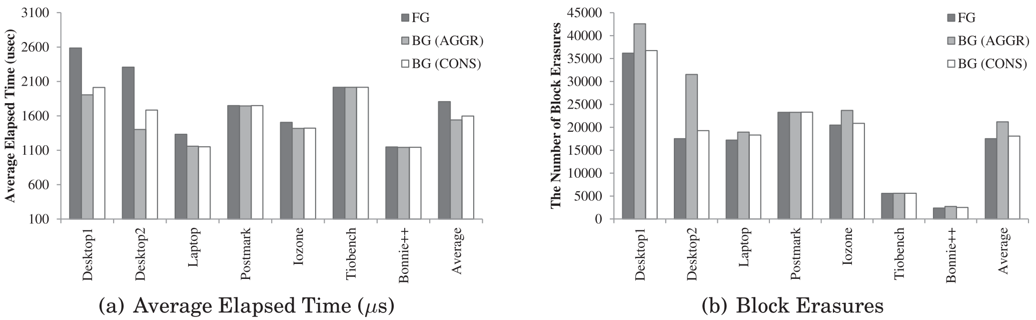


Fig. 27. A comparison of full merge costs with the full-length merge table and the reduced merge table.



(a) Average Elapsed Time (μs) (b) Block Erasures

Fig. 28. A comparison of the foreground merge policy and the background merge policy.

LAST++ with BG(CONS) conservatively performs background merges only when there are log blocks whose merge costs would not be changed in the near future. As illustrated in Figure 28(a), BG(AGGR) shows 15% shorter elapsed time because it maximally exploits available idle times to hide overheads caused by foreground block merges. Since BG(AGGR) often selects a victim block whose pages are likely to be invalid soon, however, it performs 21% more block erasures than FG. Unlike BG(AGGR), BG(CONS) carefully selects a victim log block holding many cold pages that will not be obsolete before being evicted to data blocks. For this reason, the increase in the number of block erasures is limited to 3.2%, but it improves overall I/O elapsed time by 12%, on average. Our background merge policy is less effective for Postmark, Iozone, Tiobench, and Bonnie++. Those traces are collected from micro-benchmarks that intensively issue a lot of reads and writes to the SSD. Due to very short idle times, background merges are infrequently triggered.

Finally, we assess the effect of summary pages on performance. As mentioned in Section 4.7, LAST++ keeps mapping information in reserved pages of log blocks (one page per log block). This enables quick recovery, but reduces the effective capacity of log blocks. Since summary pages account for a trivial proportion of the total log-blocks space (i.e., 1/128), its effect on performance is negligible, as depicted in Figure 29.

6. CONCLUSION

In this article, we proposed a new locality-aware FTL scheme called LAST++, which greatly improved the performance and lifetime of flash-based SSDs with small memory requirements. By exploiting the sequential and temporal localities of I/O references that were typically observed in general-purpose computing systems, LAST++ resolved

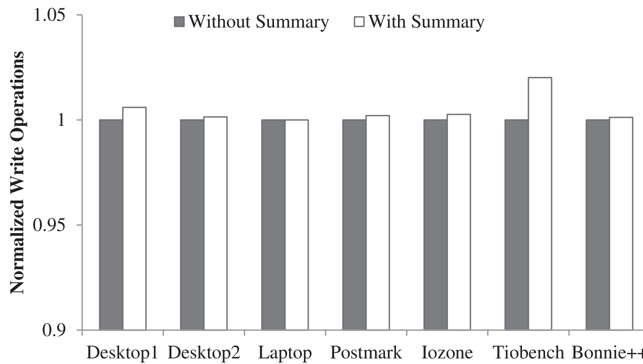


Fig. 29. A comparison of two versions of LAST++ with or without summary pages.

the low channel utilization and high garbage collection problems of the hybrid FTL scheme, improving overall SSD performance. This work also showed that the well-designed hybrid FTL could outperform DFTL in terms of performance and data integrity. Our experimental results showed that LAST++ exhibited 27% higher write performance and 7% better read performance, on average, than DFTL while ensuring higher data integrity against system crashes and/or sudden power failures. LAST++ also improved write performance and storage lifetime by 39% and 40%, respectively, over the FAST FTL.

REFERENCES

Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, 57–70.

Amir Ban. 1995. Flash file system. (April 4 1995). US Patent 5,404,485.

Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. 2007. A design for high-performance flash disks. *SIGOPS Operating Systems Review* 41, 2 (April 2007), 88–93. DOI:http://dx.doi.org/10.1145/1243418.1243429

Li-Pin Chang. 2007. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing*. ACM, 1126–1130.

Li-Pin Chang. 2010. A hybrid approach to NAND-flash-based solid-state disks. *IEEE Transactions on Computers* 59, 10 (Oct 2010), 1337–1349. DOI:http://dx.doi.org/10.1109/TC.2010.14

M.-L. Chiang and R.-C. Chang. 1999. Cleaning policies in mobile computers using flash memory. *Journal of Systems and Software* 48, 3 (Nov. 1999), 213–231. DOI:http://dx.doi.org/10.1016/S0164-1212(99)00059-X

Hyunjin Cho, Dongkun Shin, and Young Ik Eom. 2009. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09)*. European Design and Automation Association, Leuven, Belgium, 507–512.

Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, 229–240. DOI:http://dx.doi.org/10.1145/1508244.1508271

Gregory L. Heileman and Wenbin Luo. 2005. How caching affects hashing. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*. 141–154.

S. Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu Chen. 2011. S-FTL: An efficient address translation for flash memory by exploiting spatial locality. In *Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST 2011)*. 1–12. DOI:http://dx.doi.org/10.1109/MSST.2011.5937215

Theodore Johnson and Dennis Shasha. 1994. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of 20th International Conference on Very Large Data Bases (VLDB'94), September 12–15, 1994, Santiago de Chile,* Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann, 439–450.

Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software (EMSOFT'06)*. ACM, New York, NY, 161–170. DOI:http://dx.doi.org/10.1145/1176887.1176911

Han-joon Kim and Sang-goo Lee. 1999. A new flash memory management for flash storage system. In *Proceedings of the 23rd International Computer Software and Applications Conference (COMPSAC'99)*. IEEE Computer Society, Washington, DC, 284.

Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. 2002. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics* 48, 2 (May 2002), 366–375. DOI:http://dx.doi.org/10.1109/TCE.2002.1010143

George Lawton. 2006. Improved flash memory grows in popularity. *Computer* 39, 1 (2006), 16–18.

Sungjin Lee, Keonsoo Ha, Kangwon Zhang, Jihong Kim, and Junghwan Kim. 2009. FlexFS: A flexible flash file system for MLC NAND flash memory. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (USENIX'09)*. USENIX Association, Berkeley, CA, 9–9.

Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Operating Systems Review* 42, 6 (Oct. 2008), 36–42. DOI:http://dx.doi.org/10.1145/1453775.1453783

Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computer Systems* 6, 3, Article 18 (July 2007). DOI:http://dx.doi.org/10.1145/1275986.1275990

B. Leibowitz, R. Palmer, J. Poulton, Y. Frans, S. Li, J. Wilson, M. Bucher, A. M. Fuller, J. Eyles, M. Aleksic, T. Greer, and N. M. Nguyen. 2010. A 4.3 GB/s mobile memory interface with power-efficient bandwidth scaling. *IEEE Journal of Solid-State Circuits* 45, 4 (April 2010), 889–898. DOI:http://dx.doi.org/10.1109/JSSC.2010.2040230

Sang-Phil Lim, Sang-Won Lee, and B. Moon. 2010. FASTer FTL for enterprise-class flash memory SSDs. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*. 3–12. DOI:http://dx.doi.org/10.1109/SNAPI.2010.9

Micron Technology Inc. 2012. MT29F16G08 MLC NAND Flash Memory Data Sheet.

Sungup Moon, Sang-Phil Lim, Dong-Joo Park, and Sang-Won Lee. 2010. Crash recovery in FAST FTL. In *Proceedings of the 8th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems (SEUS'10)*. Springer-Verlag, Berlin, 13–22.

Robert Morris. 1968. Scatter storage techniques. *Communications of the ACM* 11, 1 (Jan. 1968), 38–44. DOI:http://dx.doi.org/10.1145/362851.362882

Dongchul Park, Biplob Debnath, and David Du. 2010. CFTL: A convertible flash translation layer adaptive to data access patterns. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'10)*. ACM, New York, NY, 365–366. DOI:http://dx.doi.org/10.1145/1811039.1811089

Sang-Hoon Park, Dong gun Kim, Kwanhu Bang, Hyuk-Jun Lee, Sungjoo Yoo, and Eui-Young Chung. 2014. An adaptive idle-time exploiting method for low latency NAND flash-based storage devices. *IEEE Transactions on Computers* 63, 5 (May 2014), 1085–1096. DOI:http://dx.doi.org/10.1109/TC.2012.281

Sang-Hoon Park, Seung-Hwan Ha, Kwanhu Bang, and Eui-Young Chung. 2009. Design and analysis of flash translation layers for multi-channel NAND flash-based storage devices. *IEEE Transactions on Consumer Electronics* 55, 3 (August 2009), 1392–1400. DOI:http://dx.doi.org/10.1109/TCE.2009.5278005

Gyudong Shim, Sung Kyu Park, and Kyu Ho Park. 2012. MNK: Configurable hybrid flash translation layer for multi-channel SSD. In *Proceedings of the IEEE 15th International Conference on Computational Science and Engineering (CSE'12)*. 445–452. DOI:http://dx.doi.org/10.1109/ICCSE.2012.68

SNIA. 2015. Storage Networking Industry Association. Retrieved from http://www.snia.org/.

P. Thontirawong, M. Ekpanyapong, and P. Chongstitvatana. 2014. SCFTL: An efficient caching strategy for page-level flash translation layer. In *Proceedings of the 2014 International Computer Science and Engineering Conference (ICSEC)*. 421–426. DOI:http://dx.doi.org/10.1109/ICSEC.2014.6978234

Zhiyong Xu, Ruixuan Li, and Cheng zhong Xu. 2012. CAST: A page-level FTL with compact address mapping and parallel data blocks. In *Proceedings of the 2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, 142–151. DOI:http://dx.doi.org/10.1109/PCCC.2012.6407747

Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the robustness of SSDS under power fault. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, Berkeley, CA, 271–284.

Received October 2014; revised July 2015; accepted November 2015