

Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices

Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha

Abstract—While NAND flash memory is used in a variety of end-user devices, it has a few disadvantages, such as asymmetric speed of read and write operations, inability to in-place updates, among others. To overcome these problems, various flash-aware strategies have been suggested in terms of buffer cache, file system, FTL, and others. Also, the recent development of next-generation nonvolatile memory types such as MRAM, FeRAM, and PRAM provide higher commercial value to Non-Volatile RAM (NVRAM). At today's prices, however, they are not yet cost-effective. In this paper, we suggest the utilization of small-sized, next-generation NVRAM as a write buffer to improve the overall performance of NAND flash memory-based storage systems. We propose various block-based NVRAM write buffer management policies and evaluate the performance improvement of NAND flash memory-based storage systems under each policy. Also, we propose a novel write buffer-aware flash translation layer algorithm, optimistic FTL, which is designed to harmonize well with NVRAM write buffers. Simulation results show that the proposed buffer management policies outperform the traditional page-based LRU algorithm and the proposed optimistic FTL outperforms previous log block-based FTL algorithms, such as BAST and FAST.

Index Terms—Nonvolatile RAM, flash memory, write buffer, flash translation layer, solid-state disk, storage device.

1 INTRODUCTION

THE rapid progress of NAND flash technology has enabled mobile devices to provide rich functionalities to users at a low price. For example, users can take pictures or listen to MP3 files using state-of-the-art cellular phones instead of digital cameras or MP3 players. As a result of large capacity flash memory, users can store many high-quality photos and MP3 files in their cellular phones. Recently, a notebook that is equipped with a NAND flash-based solid-state disk (SSD), which has tens of gigabytes of capacity, has emerged in the marketplace. Also, it is a common prediction that the SSD will replace the hard disk in the notebook market because of the low power consumption, high speed, and shock resistance, among other advantages. However, flash memory has a few drawbacks, such as the asymmetric speed of read and write operations, inability to in-place updates, very slow erasure operation, among others. Much

effort has been undertaken to overcome these drawbacks and most of these have a common objective: to reduce the number of write and erase operations. One of the approaches to achieve the objective is by exploiting the buffer cache in volatile memory to delay write operations [1], [2]. However, delaying write operations using the buffer cache in volatile memory may lead to data loss when an accidental failure (for example, power outage) occurs.

Using nonvolatile random access memory (NVRAM) as a write buffer for a slow storage device has long been an active research area. In particular, many algorithms that manage NVRAM write buffers for hard disks have been proposed [3], [4], [5]. Also, some works have proposed using NVRAM as a write buffer for the NOR-type flash memory-based storage system [6], [7].

For the past decade, next-generation nonvolatile memory has been under active development. The next-generation nonvolatile memory types such as phase-change RAM (PRAM), ferroelectric RAM (FeRAM), and magnetic RAM (MRAM) are known not only to be as fast as DRAM in terms of both read and write operations, but also as in-place updating is possible. The access times for DRAM, next-generation NVRAM, and NAND flash memory are summarized in Table 1 [8], [9].

Recently, leading semiconductor companies announced the development of 512 Mbits-capacity PRAM module [10] and a 4-states MLC (Multi-Level Cell) PRAM technology [11]. Hence, it is expected that a commercial product will be produced in the near future. However, previous researches on using NVRAM as a write buffer for slow storage devices [3], [4], [5] are not suitable to be used for NAND flash memory-based storage systems since they do not consider not only the characteristics of the NAND flash memory, but

• S. Kang is with the Division of Information and Communications, Hanyang University, Haengdang-dong, Seongdong-gu, Seoul 133-791, Korea. E-mail: sykang@hanyang.ac.kr.

• S. Park is with the Division of Electronics and Computer Engineering, Hanyang University, Seongdong-gu, Seoul 133-791, Korea. E-mail: syrilo@hanyang.ac.kr.

• H. Jung and H. Shim are with the Division of Electronics and Computer Engineering, Hanyang University, Haengdang-dong, Seongdong-gu, Seoul 133-791, Korea. E-mail: {horong, dahlia}@hanyang.ac.kr.

• J. Cha is with the Division of Information and Communications, Hanyang University, Seongdong-gu, Seoul 133-791, Korea. E-mail: chajh@hanyang.ac.kr.

Manuscript received 7 Aug. 2008; revised 10 Nov. 2008; accepted 9 Dec. 2008; published online 19 Dec. 2008.

Recommended for acceptance by N. Ranganathan.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-08-0401.

Digital Object Identifier no. 10.1109/TC.2008.224.

Authorized licensed use limited to: Fish & Richardson PC. Downloaded on April 28, 2026 at 05:00:04 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Characteristics of Storage Media

Media	Access time		
	Read	Write	Erase
MRAM	10~50ns (2B)	10~50ns (2B)	N/A
FeRAM	30~100ns (2B)	30~100ns (2B)	N/A
PRAM	20~80ns (2B)	20~80ns (2B)	N/A
DRAM	20ns (2B)	20ns (2B)	N/A
NAND Flash	25μs (2KB)	200μs (2KB)	1.5ms

also the behavior of the flash translation layer (FTL) that enables the use of flash memory-based storage as an ordinary block device such as a hard disk. To provide another point of view, current FTL algorithms [12], [13], [14], [15], [16], [17] are designed without any consideration of the existence of a write buffer. Since the write request pattern to the flash memory greatly changes when the requests are filtered from the write buffer, the existence of the write buffer greatly affects the performance of FTL algorithms, which necessitates a write buffer-aware FTL algorithm.

The contribution of our work is twofold. First, we present various NVRAM write buffer management algorithms for NAND flash memory-based storage systems, which use well-known FTL algorithms and show a performance gain that can be achieved using a write buffer. Though the term “NVRAM” in this paper means the next-generation NVRAM, our scheme can also be used with the traditional NVRAM such as battery-backed DRAM. Second, we develop a novel write buffer-aware flash translation layer (*Optimistic FTL*) for a NAND flash memory-based storage system.

The rest of this paper is organized as follows: in Section 2, we present background knowledge of our approach including NAND-type flash memory, FTL algorithms, among others. In Section 3, we introduce various algorithms for using NVRAM as a write buffer for NAND-type flash memory. In Section 4, we show the feasibility of using NVRAM as a write buffer for NAND-type flash memory through various trace-driven simulation results. In Section 5, we propose a write buffer-aware FTL scheme (*Optimistic FTL*) and evaluate its performance. Finally, in Section 6, we draw conclusions from this study.

2 BACKGROUNDS

2.1 Characteristics of the NAND Flash Memory

NAND flash memory consists of blocks, each of which consists of pages. In small-block NAND flash memory, there are 32 pages in a block and a page consists of a 512-byte data area and a 16-byte spare area. In large-block flash memory, there are 64 pages in a block and a page consists of a 2,048-byte data area and 64-byte spare area. The spare area contains Error Correction Code or metadata for the page or block. Table 2 compares the device architectures between a 1 Gbit small-block flash memory and a 1 Gbit large-block flash memory. In this paper, we assume that only large-block NAND flash memory is used since small block architecture is not used for large capacity (> 1 Gbit) NAND flash memory because of its relatively lower performance.

TABLE 2
Small-Block and Large-Block NAND Flash Memories

	1Gb small block NAND flash	1Gb large block NAND flash
Page	(512+16) bytes	(2K+64) bytes
Block	(512+16) bytes × 32 pages = (16K+512) bytes	(2K+64) bytes × 64 pages = (128K+4K) bytes
Device	(16K+512) bytes × 8192 blocks = 1056Mb	(128K+4K) bytes × 1024 blocks = 1056Mb

There are three basic operations for a NAND flash memory: read, write (program), and erase. In the NAND flash memory, unlike NOR flash memory, the read and write operations take place on a page basis rather than on a byte or word basis—the size of data I/O register matches the page size. In the read operation, a page is transferred from the memory into the data register for output. In the write operation, a page is written into the data register and then transferred to the flash array. The erase operation takes place on a block basis. In a block erase operation, a cluster of consecutive pages in a block are erased in a single operation.

While NAND flash memory is used in a variety of end-user devices, it has a few drawbacks, which are summarized as follows [9], [18]:

- *Asymmetric operations speed*: As we can see from Table 1, the write and erase operations are slower than the read operation by about 8 and 80 times, respectively.
- *Inability to in-place update*: Flash memory is write-once and overwriting is not possible. The memory must be erased before new data can be written.
- *Limited lifetime*: The number of erasures possible on each block is limited, to 100,000 or 1,000,000 times.
- *Random page write prohibition within a block*: Within a block, the page must be written consecutively from the least significant bit (LSB) page of the block to most significant bit (MSB) page of the block. In this case, the definition of the LSB page is the LSB among the pages to be written. Therefore, the LSB page does not need to be page 0.

We can easily find that the above drawbacks are directly or indirectly related to the characteristics of the write operation in the NAND flash memory. Hence, we carefully conjecture that the use of an NVRAM write buffer can overcome these drawbacks and therefore dramatically improve the overall performance of the NAND flash memory-based storage systems.

2.2 Flash Translation Layer

To use flash memory as a storage device we can either use dedicated file systems [19], [20] for flash memory or emulate block device with flash memories using Flash Translation Layer (FTL). FTL is a widely used software technology, enabling general file systems to use flash memory-based storage device in the same manner as a generic block device such as a hard disk. FTL can be either implemented in the host system (e.g., Smart media) or packaged with the flash device (e.g., Compact flash, USB memory). To emulate a block device using flash memory,

FTL provides a few core functionalities such as address mapping, bad block management, and ECC check. FTL also provides wear leveling functionality to extend the lifetime of the flash memory. The address mapping functionality maps two address domains: the logical block address (LBA) and the hardware address. Since the mapping scheme greatly affects the number of read, write, and erase operations that actually occur in the flash memory, the overall performance of the flash memory-based storage system highly depends on the mapping scheme.

We can categorize address mapping schemes, according to the mapping granularity, into block mapping schemes and page mapping schemes. Page mapping schemes maintain a mapping table in which each entry represents the mapping information between single logical page and single physical page. While the page-level mapping provides great flexibility to space management, it requires a large memory space to store mapping table itself, which makes it infeasible to be used in practical applications. Block mapping schemes associate logical blocks with physical blocks, which requires far less entries in the mapping table. For example, while the size of a page mapping table is about 6 Mbytes for a 32-Gbit large block flash memory, the size of the block mapping table is only about 128 Kbyte. However, in the block mapping scheme, page offsets in the logical block and the corresponding physical block should remain the same.

The performance of the FTL depends on its implementation algorithm. There are many FTL algorithms such as Mitsubishi, NFTL, AFTL, and log block Schemes (BAST and FAST) [13], [14], [15], [16], [17]. In this paper, we only consider log block schemes (BAST and FAST) since they are known to outperform others [17].

2.3 Log Block-Based Address Mapping

Log block-based address mapping schemes (log block schemes) exploit a mixed use of page mapping and block mapping. They reserve a small fixed number of blocks as log blocks and other blocks are used as data blocks. Data blocks hold ordinary data and are managed at the block-level. Log blocks are used as temporary storage for small-size write to data blocks and are managed at the finer page-level.

There are two representative log block schemes: BAST and FAST. While the BAST algorithm associates one log block to a data block, the FAST algorithm associates multiple log blocks to multiple data blocks. In the BAST algorithm, overall space utilization decreases due to the limited association that allows only one data block to a single log block. The FAST algorithm overcomes this limitation by associating a log block to multiple data blocks. By sharing log block, however, FAST algorithm shows unstable merge operation latency, and the performance of the flash memory system decreases when a merge operation occurs because all data blocks should be merged accordingly. Also, in comparison with the BAST algorithm, more entries in the map table should be modified when a log block is merged with multiple data blocks.

When a write operation to a dirty page in a data block is issued, log block schemes perform the operation to a page in the corresponding log block. When there is no available

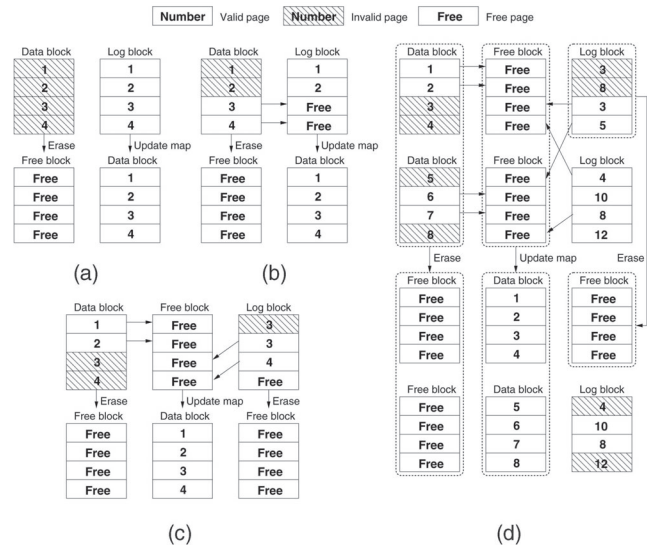


Fig. 1. Three kinds of merge operations in BAST and FAST.

log block, they select a victim log block and merge it with its corresponding data block(s)—a *merge* operation. While executing the merge operation, multiple page copy operations (to copy valid pages from both the data block and log block to the newly allocated data block) and erase operations (to create a free block) are invoked. Therefore, merge operations seriously degrade the performance of the storage system because of these *extra operations*.

Fig. 1 shows three different forms of merge operations, which are as follows [21]:

- *Switch merge*: When every page in data block is updated sequentially, the page sequence in the log block is the same as that in the data block. In this case, the log block is changed into the data block and the original data block is erased. *Switch merge* is the most economical merge operation.
- *Partial merge*: When a contiguous subset of pages in a data block is updated sequentially and other pages are not updated, the updated pages in the log block preserve the page sequence in the data block. Then, the remaining valid pages in the data block are copied to the log block and the log block is changed into a data block and the original data block is erased.
- *Full merge*: When a noncontiguous subset of pages in a data block is updated, the page sequence in the data block and in the log block is different. In this case, a new data block is allocated and valid pages in both the original data block and log block are copied to the new data block. Next, both the original data block and log block are erased. *Full merge* is the least economical merge operation.

While both the *switch merge* and *partial merge* operations are not different according to the algorithms, the *full merge* operation in each algorithm are different. In the BAST algorithm, the *full merge* operation merges only one data block with the victim log block, while multiple data blocks are needed to be merged with the victim log block in the FAST

algorithm. Generally, large sequential write operations can induce *switch merge* operations while random write operations induce *full merge* operations. Therefore, if random write operations occur frequently, the performance of the flash memory system decreases.

3 NVRAM WRITE BUFFER MANAGEMENT POLICIES FOR FLASH MEMORY

3.1 Design Considerations

Write buffer management schemes for hard disk have been developed over the past decade. According to those schemes, the performance of the write buffer management scheme for a hard disk depends on the following two factors:

- *Total number of destages to the hard disk:* The total number of destages to the hard disk is the number of write operations actually issued to the hard disk. As more write requests are accommodated by the write buffer, less requests are issued to the hard disk. In the LRW (Least Recently Written page) scheme [3], [4], [5], they exploited the temporal locality of the data access pattern to increase the write buffer hit ratio and decrease the number of destages to the hard disk.
- *Average access cost of each destage:* Since read and write operations in a hard disk show a symmetric operation speed, the access cost in a hard disk can be modeled as the sum of the seek time, rotational delay, and transfer time. By exploiting the spatial locality of the data access pattern, SSTF, LST [4], and CSCAN [5] attempted to decrease the seek time and rotational delay.

The stack model in [4] used the LRW and LST schemes, simultaneously, to exploit both the temporal locality and spatial locality, and WOW [5] used LRW and CSCAN, simultaneously, to exploit both localities. These two factors also make sense when using NAND flash memory instead of a hard disk. First, the number of destages to flash memory should be decreased to increase the overall performance. To decrease the number of destages to flash memory, the write buffer hit ratio should be increased. Therefore, we can use traditional buffer management schemes that exploits the temporal locality of the data access pattern. However, the access cost factor makes it necessary to devise a novel write buffer management scheme. While the access costs for data blocks that are stored in physically different locations in the hard disk vary, the physical location of a data block in the flash memory does not affect the access time to the block. The spatial locality is no longer important factor for flash memory. Therefore, instead of the seek time and rotational delay, another important factor should be considered to estimate the access cost for flash memory: *extra operations* issued by FTL.

As described in Section 2.3, FTL issues extra read, write, or erase operations internally to efficiently manage the storage space in flash memory, and the number of those extra operations depends both on the data access pattern from the upper layer and the algorithm used for address

mapping. Hence, consideration of the FTL algorithm is inevitable for designing a write buffer management scheme for flash memory. For example, since most of the FTL algorithms use block-level mapping, the write buffer management scheme should be designed to decrease the number of merge operations (each of which consists of multiple copies of valid data pages and a maximum of two block erase operations) that can be invoked internally while processing write operations.

To decrease the number of extra operations, the write buffer management scheme is required to 1) decrease the number of merge operations by clustering pages in the same block and destaging them at the same time, 2) destage pages such that the FTL may invoke *switch merge* or *partial merge* operations which show relatively low cost rather than the *full merge* operation, which is very expensive, and 3) detect sequential page writes and destage those sequential pages preferentially and simultaneously.

Fig. 2 shows the necessity of the block-level management of the write buffer. There are 10 pages in the NVRAM, and the flash memory consists of more than four data blocks, including blocks A, B, C, and D, and only two log blocks (L1 and L2). Each block in the flash memory consists of four physical pages. Data blocks A, B, C, and D contain corresponding data pages: block A contains pages A1, A2, A3, and A4; block B contains B1, B2, B3, and B4; block C contains C1, C2, C3, and C4; and block D contains D1, D2, D3, and D4. NVRAM is currently filled with pages A3', A4', B1', B2', C1', D1', A1', A2', A2', and C2' (each page is the newer version of the corresponding page in the flash memory), and among these, page A3' is the oldest page (LRU page) and page C2' is the newest page (MRU page). While four data blocks in the flash memory are fully used, no page in the log blocks is used yet. When subsequent write requests are issued to NVRAM, the write buffer manager selects victim pages according to the replacement algorithm and evicts them to the flash memory, one after another. The figure shows the sequence of operations to the flash memory when victim pages are evicted, assuming that BAST is used for the FTL algorithm.

Assume that buffer pages are managed in a page-level scheme and the victim selection sequence is A3', A4', B1', B2', C1', D1', A1', A2', A2', and C2'. In this case, FTL writes pages A3' and A4' into log block L1 and B1' and B2' into log block L2. When page C1' is evicted to the flash memory, since there is no remaining log block, FTL merges (*full merge*) data block A and log block L1 into a new data block and erases A and L1 to acquire an empty log block, and then writes C1' into the erased L1. At this time, the merge operation consists of two erase operations, four read operations, and four write operations.

When buffer pages are managed in a block-level scheme, pages in the NVRAM are clustered by corresponding block number in the flash memory. Assume that the victim cluster selection sequence is Cluster A, Cluster B, Cluster C, and Cluster D. Since a page cluster is selected as a victim, all pages in the victim page cluster are evicted to the flash memory simultaneously. Hence, the sequence of evicted pages becomes A1', A2', A3', A4', B1', B2', C1', C2', D1', and D2'. In this case, FTL writes pages A1', A2', A3', and A4'

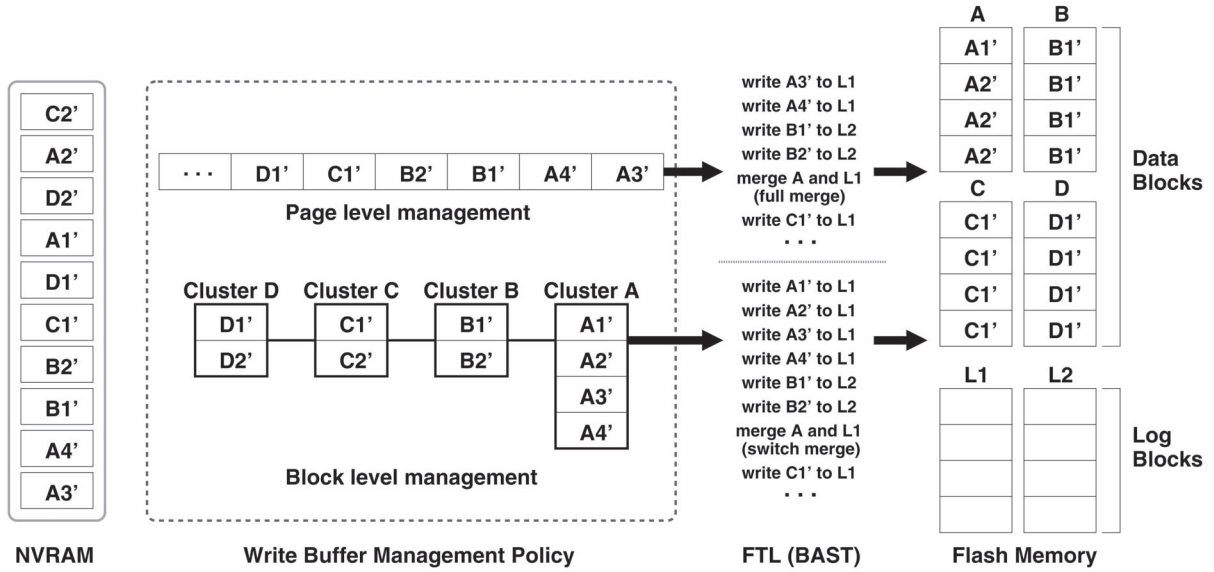


Fig. 2. Comparison between page-level buffer management and block-level buffer management.

into log block L1 and B1' and B2' into log block L2. When pages C1' and C2' are evicted to the flash memory, since there is no remaining log block, FTL merges (*switch merge*) data block A and log block L1 and erases L1, and then writes C1' and C2' into the erased L1. At this time, the merge operation consists of only one erase operation.

In this manner, we can easily obtain the total number of extra operations when all pages in NVRAM are evicted to the flash memory. Using page-level management policy, 11 read, 11 write, and 5 erase operations are invoked while 2 read, 2 write, and 2 erase operations are invoked using block-level management policy. As we can see from this example, block-level buffer management policy not only invokes relatively fewer merge operations than page-level buffer management policy but also invokes *switch merge* or *partial merge* rather than *full merge* for merge operation. Block level buffer management policy has one drawback: since it evicts multiple pages at a time even though only one page replacement is needed, the utilization of the NVRAM pages can be decreased, which results in a lower buffer hit ratio than that of page-level buffer management policy. However, since the benefit from the reduced cost for extra operations by block-level buffer management policy is much greater than the drawback, the block-level buffer management policy shows better overall performance than the page-level buffer management policy.

3.2 Write Buffer Management Policies

In this section, we propose four write buffer management policies, among which three are block-level buffer management policies and one is a page-level buffer management policy. Page-level buffer management policy is introduced only for comparison purposes.

We assumed that the block-level address mapping is used in FTL since the page-level address mapping is not used widely in practical situation. Hence, data movement between NVRAM and flash memory is done according to the block mapping algorithm in FTL. Since the page size in

the large block NAND flash memory is 2 KB, we assumed the page size in the NVRAM is also 2 KB. For block-level buffer replacement policies, we clustered pages in the NVRAM by the block number in the flash memory and those page clusters are maintained through a linked list. In each page cluster, pages with the same block number in flash memory are residing through a linked list. The size of a cluster is defined as the number of pages in the cluster, which varies from 1 to 64. Fig. 3 shows the data structure of write buffers for block-level buffer management policies.

3.2.1 Least Recently Used Page (LRU-P) Policy

In LRU-P policy, the replacement unit is a page and the least recently used (written) page in the buffer is selected as a victim. Since the reference to the NVRAM write buffer is done in a page unit, LRU-P policy most precisely reflects the "hotness" of pages, which results in a relatively high page hit ratio in the buffer. To handle sequential page writes, LRU-P regards 64 or more consecutive page writes as a sequential page write and maintains those sequential pages with a sequential page list. Pages in the sequential page list are selected preferentially as victim pages and if the list is empty, the least recently used page is selected as a victim.

3.2.2 Least Recently Used Cluster (LRU-C) Policy

In LRU-C policy, the replacement unit is a page cluster and the least recently accessed cluster is selected as a victim. Access to a cluster means either modifying an already

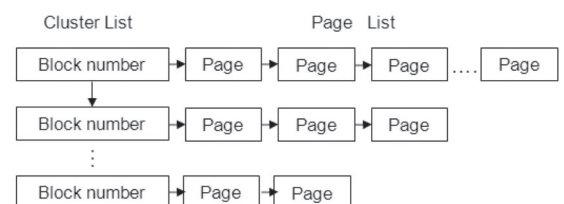


Fig. 3. Data structure for block-level write buffer management.

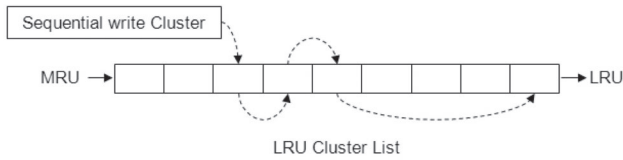


Fig. 4. Data structure for LRU-C policy.

existing page in the cluster or writing a new page in the cluster. Fig. 4 shows the data structure for the LRU-C policy.

To handle sequential page writes, LRU-C regards 64 consecutive page writes into a cluster as a sequential page write and maintains those clusters with a sequential write cluster list. Clusters in the sequential write cluster list are selected as victim clusters preferentially and if the list is empty, the least recently accessed cluster is selected as a victim.

BPLRU [22] is a variant of LRU-C policy. It can be seen as a LRU-C policy with additional block padding scheme. It exploits block padding so as not to invoke full merge operation, which is very expensive, in the underlying FTL. In this way, BPLRU could improve the random write performance of the flash memory-based storage devices. However, since BPLRU is based on the LRU-C policy, it could not help inheriting the drawback of the LRU-C policy: Since pages are clustered together, a page cluster can contain both hot and cold pages at the same time. Also, since the recency of a cluster largely depends on the recency of the hot pages in the cluster, cold pages in a cluster remain for the same duration as hot pages in the cluster, which can decrease the overall page hit ratio in the buffer.

3.2.3 Largest Cluster (LC) Policy

As more pages are clustered and destaged together, the number of merge operations is expected to decrease. Hence, if we select the largest page cluster as a victim, we can expect an increased overall performance of the storage system. This is the rationale of the LC policy.

In the LC policy, the replacement unit is a page cluster and the page cluster with the largest cluster size is selected as a victim. It maintains LRU cluster lists for every cluster size (from 1 to 64), as shown in Fig. 5, and the victim is selected from the LRU position of the cluster list whose cluster size is 64. If the list is empty, then the cluster list whose cluster size is 63 is searched from the LRU position to the MRU position.

A DRAM buffer replacement policy, flash-aware buffer replacement (FAB) [2] is similar to the LC policy. It selects the block having the largest number of pages in it as a victim block for replacement. It maintains pages not only for write requests but also for read requests. If we modify FAB such that it only considers write requests, the modified FAB can also be used as a write buffer management policy. However, it does not consider page pinning which is necessary to cope with the sequential page writes.

When a page cluster is currently being accessed for writing, it is necessary to protect the cluster from being selected as a victim. Hence, the page cluster is pinned when a write request to the cluster arrives and unpinned when a write request to another cluster arrives. Only when a cluster

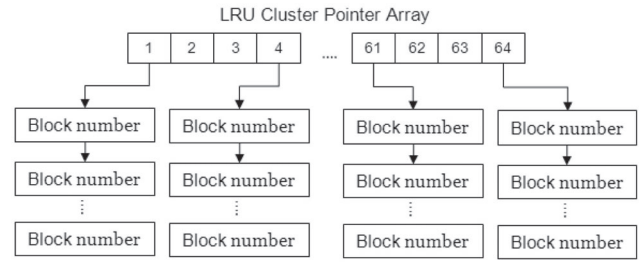


Fig. 5. Data structure for LC policy.

changes its state from pinned to unpinned, the cluster is moved from the previous position to the MRU position of the cluster list. If the size of the cluster is changed because of the addition of new pages, it moves to another cluster list of which the size is equal to the changed cluster size. In this manner, sequential writes to a page cluster invoke at most one location change in the cluster lists.

As sequential page writes in a cluster make the size of the cluster relatively larger, the cluster comes to have a high probability for selection as a victim. Hence, it is not required to deal with sequential page writes, separately, in LC policy.

However, the LC policy has a few drawbacks. Since the victim selection sequence in the LC policy largely depends on the cluster size rather than the recency of the cluster or page, the temporal locality of page access is not sufficiently exploited in the LC policy, which can decrease the page hit ratio in the buffer. More seriously, when the buffer is mostly filled with small-sized cold clusters, hot clusters come to be selected as victims before they are sufficiently matured (i.e., before the size of each hot cluster is increased sufficiently).¹ Consequently, the entire buffer comes to be filled with only small-sized page clusters, in which case the overall performance of the storage system greatly degrades because the size of the victim cluster is always small.

3.2.4 Cold and Largest Cluster (CLC) Policy

In the CLC policy, both the temporal locality and cluster size are considered simultaneously. The replacement unit of the CLC policy is also a page cluster and the page cluster with the largest cluster size among the cold clusters is selected as a victim. To accommodate both the temporal locality and cluster size, it maintains two kinds of cluster lists: 1) size-independent LRU cluster list and 2) size-dependent LRU cluster list for each cluster size. The size-independent LRU cluster list is the same as the LRU cluster list in LRU-C policy, and the size-dependent LRU cluster lists for each cluster size are from the LC policy.

Fig. 6 shows the data structure for CLC policy. When a page cluster is initially generated, it is inserted into the MRU position of the size-independent LRU cluster list, and whenever the cluster is accessed, it moves to the MRU position of the list. When the size-independent LRU cluster list is full and a new page cluster arrives, the page cluster in the LRU position of the list is evicted from the list and inserted into the size-dependent LRU cluster list with a

1. "Cold cluster" means a page cluster in which page access does not occur frequently. Since the size of the cold cluster rarely changes, small-sized cold clusters remain in the buffer for a long time until all large-sized clusters are evicted to the flash memory.

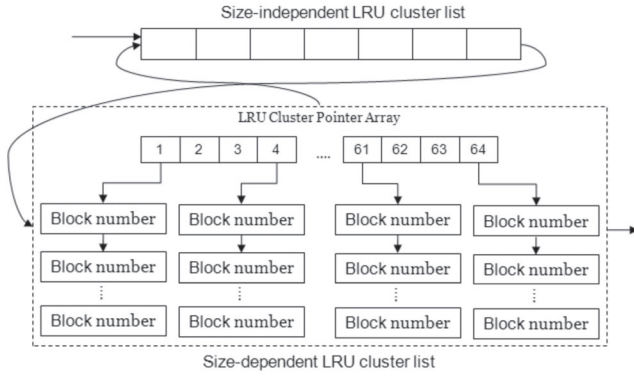


Fig. 6. Data structure for CLC policy.

corresponding cluster size. If a page cluster residing in any of the size-dependent LRU cluster lists is accessed, the page cluster is moved to the MRU position of the size-independent LRU cluster list. In this manner, hot clusters come together in the size-independent LRU cluster list and cold clusters in the size-dependent LRU cluster list. The victim cluster is selected, in the same way as the LC policy, from the size-dependent LRU cluster list. Therefore, only a cold and large cluster is selected as a victim.

The sequential page writes are detected in the same way as in the LRU-C policy. When 64 consecutive page writes in a page cluster occur, the CLC policy moves the cluster into the size-dependent LRU cluster lists of size 64. Hence the cluster is preferentially selected as a victim.

The buffer space partitioning between the two kinds of lists is determined based on the number of page clusters, not the physical size, using a partition parameter α ($0 \leq \alpha \leq 1$). If $\alpha = 0.1$, then 10 percent of the total number of page clusters in write buffer are maintained with the size-independent LRU cluster list and the remaining 90 percent of the page clusters are maintained with the size-dependent LRU cluster lists, regardless of the size of each page cluster.² Hence, the CLC policy subsumes both the LRU-C and LC policies - when $\alpha = 1$, it becomes LRU-C and when $\alpha = 0$, it becomes LC.

4 PERFORMANCE EVALUATION

In this section, we evaluate the performance of each buffer management policy. We used BAST and FAST for the underlying FTL algorithm and a Samsung K9NBG08U5A 32-Gbit large block flash memory for the storage device. We used I/O traces (FAT and NTFS) provided from [23]. While the traces contain both the read and write requests, we used only write requests. The characteristics of the traces are shown in Table 3.

4.1 Characteristics of the Write Buffer Management Policies

Fig. 7a shows page hit ratio in the write buffer for each buffer management policy when the NVRAM write buffer size varies from 512 KB to 16 MB. As we can see from the figure,

2. The partition parameter, α , does not represent the physical size of each LRU cluster list in the buffer. Actually, the physical size of each list varies dynamically according to the size distribution of page clusters.

TABLE 3
Characteristics of Disk I/O Trace

File system	FAT32, NTFS
Running application	Web surfing, sending/receiving e-mail, movie playback and downloading, document typesetting, and gaming
Sector size	512 bytes
Duration	One month
Final capacity utilization	FAT: 71% (initially empty), NTFS: N/A
Total data written	FAT: 18,284 MB, NTFS: 118,550 MB
Mean write size	FAT: 5.7 sectors, NTFS: N/A
Write locality	FAT: 65-32, NTFS: 70-17 (A-B: A% of total requests access B% of total LBA's)

recency-based replacement policies, LRU-P, LRU-C, and CLC show higher page hit ratio than size-based replacement policy, LC. Interestingly, LRU-P, which reflects temporal locality in a page unit, does not show higher page hit ratio than LRU-C and CLC, which reflect temporal locality in a block unit. This means that the page clustering not only can preserve temporal locality but also can accommodate spatial locality of page accesses. As a result, we can expect an improved overall performance through merely clustering pages and enlarging replacement granularity. The LC policy, which prefers a larger cluster rather than a cold cluster as a victim, shows the worst page hit ratio.

Fig. 7b compares the number of destaged page clusters among three cluster-based buffer management policies. The LC policy evicts the largest cluster regardless of the "hotness" of the cluster. Since the evicted hot cluster is likely to be accessed and therefore introduced in the buffer again in the near future, it invokes another eviction of a victim cluster. Therefore, considering only the cluster size for victim selection may result in the large number of total cluster evictions. Furthermore, hasty eviction of a "hot and relatively large" cluster prevents the cluster from becoming even larger.

Fig. 8 shows the variation of average cluster size until an initial 10,000 victim clusters are selected. The x -axis in the figure represents the sequence of victim cluster selections. We examined the average size of clusters in the write buffer at each time a victim cluster is selected. The y -axis represents the examined average cluster size. Hence, the unit of y -axis is the number of pages in a cluster, in average. We can see from the figures that small-sized clusters occupy the write buffer most of the time in the LC and CLC policies. In particular, the average cluster size in LC policy approaches 1, as time goes on, in which case the buffer is filled with clusters consisting of only one page. From that time, hot clusters come to have no chance to increase their sizes because they are selected as a victim whenever their size is increased. As a result, the LC policy cannot exploit the effect of page clustering while degrading the page hit ratio. Hence, the overall performance of the LC policy is expected to be worse than other policies. In the CLC policy, since the newly generated page cluster is inserted into the size-independent cluster list, hot clusters can be accessed again, and therefore, increase their size in the size-independent cluster list. Hence, the average cluster size can increase even when the buffer is mostly filled with small-sized cold clusters in the CLC policy.

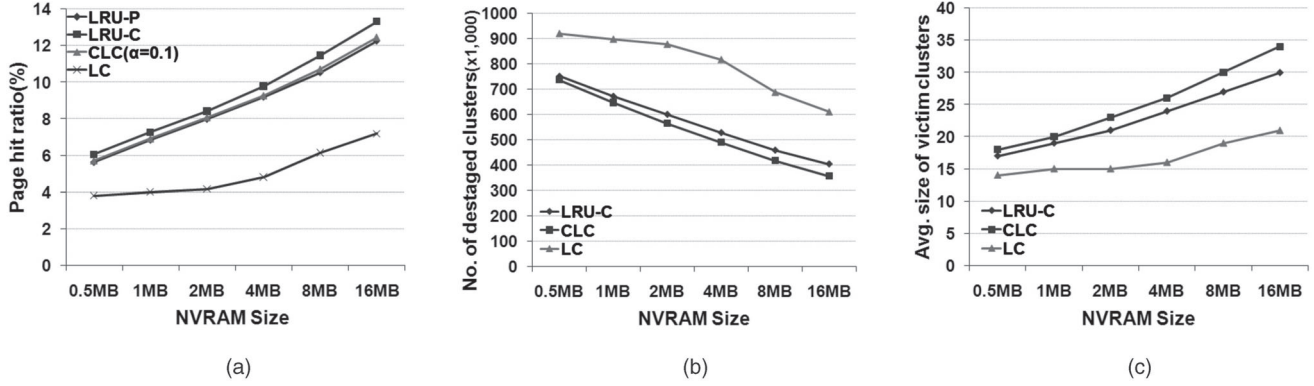


Fig. 7. Characteristics of write buffer management policies (FAT trace). (a) Page hit ratio, (b) number of destaged clusters, and (c) average size of victim clusters.

Preserving hot clusters while evicting cold clusters can provide a chance to page clusters in the buffer to form a larger spectrum of cluster sizes, which can enlarge the average size of the evicted clusters and resultantly decrease the total number of destaged clusters. As we can see from Fig. 7b, the number of destaged clusters in the CLC policy is much smaller than that in the LC policy.

Fig. 7c shows the average size of the victim clusters in each policy. The result shows, fairly well, the relationship between the total number of destaged clusters and the average size of the victim clusters. As the average victim cluster size increases, the number of destaged clusters decreases. Hence, as we increase the victim cluster size, we can decrease the total number of extra operations in FTL, which, as a result, increases the overall I/O performance of the flash memory-based storage system. As we can see from the figure, the CLC and LC policies show the largest and smallest average size of the victim clusters, respectively. While the average cluster size in LRU-C policy is much larger than that in CLC policy (Fig. 8), the average size of the victim clusters in LRU-C policy is smaller than that in CLC policy. This is because the CLC policy selects the largest cluster, from the size-dependent LRU cluster lists, as a victim, while the LRU-C policy does not consider the cluster size.

4.2 Performance Comparison

Fig. 9 shows the overall performance of each write buffer management policy. The *extra overhead* is used as the performance metric. Extra overhead is the time overhead induced by the *extra operations*. Extra operations occur while

merge operation is performed, and a merge operation consists of *valid page copies* and *erases*. Hence, we itemized the *extra overhead* into *valid page copy overhead* and *erase overhead*, each of which is the time overhead for each operation. The *y*-axis of the figure represents the normalized extra overhead and the *x*-axis represents the write buffer size.

We can figure out the effect of page clustering by comparing the performances of the LRU-P and LRU-C policies. The overall performance of the LRU-C policy is about 11-43 percent (both in FAT and NTFS) higher in the BAST case and 12-40 percent (in FAT) and 18-47 percent (in NTFS) higher in the FAST case. It shows that clustering pages in the same erasure unit (i.e., block) can decrease the number of valid page copy and erase operations. Also, the effect of page clustering increases as the write buffer size increases, since a cluster can stay in the buffer for a longer time during which more pages can be gathered in the cluster. Hence, the performance gap between LRU-P and LRU-C increases as the buffer size increases.

LRU-C shows far better page hit ratio than the LC policy (Fig. 7a), and the overall I/O performance of the LRU-C policy is better than that of the LC policy. Figs. 7b and 7c show that the average size of victim pages in the LC policy is smaller than that of the LRU-C policy and the number of destaged clusters in the LC policy is larger than that in the LRU-C policy. Writing a small-sized cluster may invoke, with higher probability than writing a large-sized cluster, the *full merge* operation, which requires a greater number of valid page copies and erasures rather than other types of merge operations (*switch merge* or *partial merge*). Hence, frequent writing of small-sized clusters makes the overall performance of the LC policy worse even than that of the LRU-P policy. Therefore, considering only the size of the page cluster can be the worst choice for victim selection.

While the CLC policy shows a slightly lower page hit ratio than that of the LRU-C policy (Fig. 7a), not only the number of destaged clusters in the CLC policy is smaller than that in the LRU-C policy (Fig. 7b), but also the average size of the victim clusters is larger than that in LRU-C policy (Fig. 7c). The CLC policy could harvest those affirmative effects only by reserving part of the buffer space for a pure LRU cluster list (size-independent LRU cluster list). We can

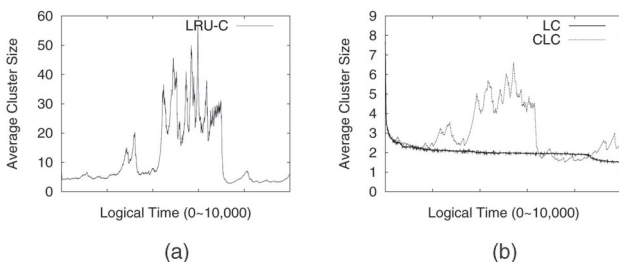


Fig. 8. Average cluster size in the buffer (FAT trace). (a) LRU-C and (b) LC and CLC.

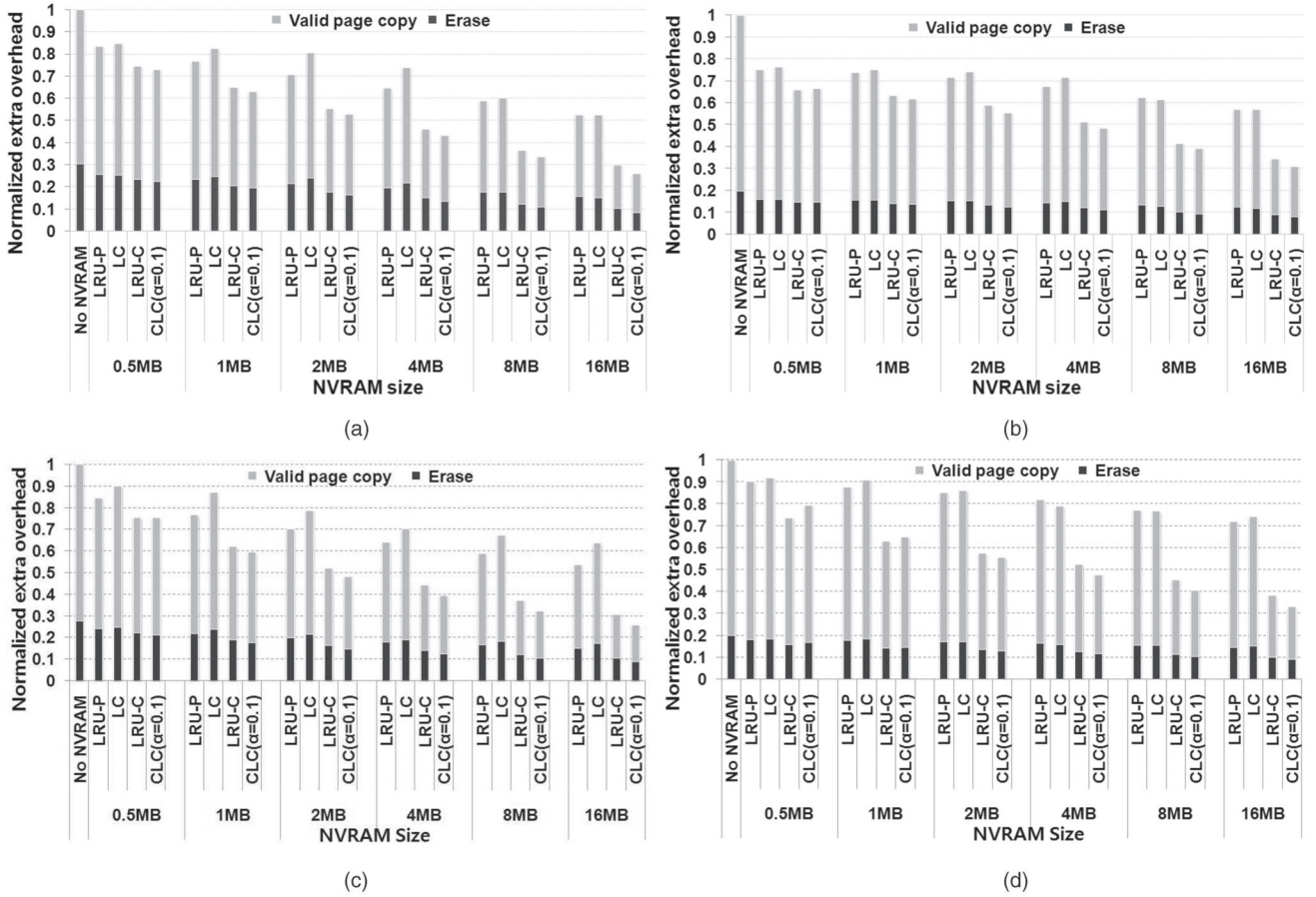


Fig. 9. Performance comparison: Extra overhead is normalized such that the overhead with no NVRAM write buffer is 1. (a) BAST: number of log blocks = 16 (FAT trace), (b) FAST: number of log blocks = 16 (FAT trace), (c) BAST: number of log blocks = 16 (NTFS trace), and (d) FAST: number of log blocks = 16 (NTFS trace).

see the effect of the pure LRU cluster list from Fig. 9, where the CLC policy outperforms others in all cases.

Fig. 10 shows the overall performance of the CLC policy with various proportions of size-independent LRU cluster list in the total NVRAM buffer space. The time is normalized such that the execution time is 1 when $\alpha = 0.1$ and the buffer size is 1 MB. The CLC policy is the same as the LC policy when $\alpha = 0$ and the LRU-C policy when $\alpha = 1$. It shows the best performance when $\alpha = 0.1$, which means that maintaining 10 percent of the total number of

page clusters for hot clusters (in the size-independent LRU cluster list) can sufficiently exploit the temporal locality of the storage access.

5 OPTIMISTIC FTL

5.1 Motivation

Since previous FTL algorithms did not consider the existence of the NVRAM write buffer, their design policy to “efficiently accommodate different write patterns (e.g., random small writes or sequential large writes) to the flash memory” added considerable complexity to the FTL. For example, BAST and FAST algorithms use log blocks to cope with random small writes. A page is not updated but is invalidated and the new version of the page is written in the log block. To keep a trace of the up-to-date pages, log block-based FTL algorithms maintain a sector mapping table for log blocks, and when a read/write request to a page arrives, they must first search the page mapping information in the mapping table. If it exists in the table, the corresponding page in the log block is accessed; otherwise, the original page in the data block is accessed. In this way, the erase operation is delayed until a fairly large number of updated pages are written to the log block. However, until the data block and log block are merged, all page accesses require a mapping table lookup. Moreover, small random

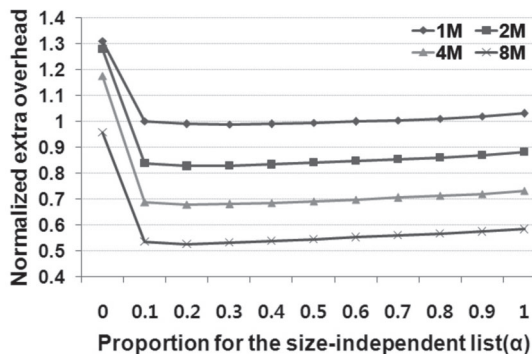


Fig. 10. The effect of the proportion for size-independent LRU cluster list in the buffer (FAT trace).

writes can invoke the *full merge* operation when merging the data block and log block, which is relatively expensive.

Using an NVRAM write buffer for page clustering, small random writes can be transformed into sequential writes to the flash memory. The sequential writes to the flash memory decrease the necessity of sector mapping for log blocks. If we can keep the ordered sequence of pages in a log block, without large overhead, we can remove the sector mapping table, which requires not only much memory space but also page search overhead. Also, it can simplify the complicated merge process by removing the *full merge* operation.

5.2 Optimistic FTL: A Write Buffer-Aware FTL

In this section, we propose a novel write buffer-aware FTL, Optimistic FTL, which exploits the characteristics of the block-level write buffer management policies. It assumes that pages in the buffer are clustered according to the block number in the flash memory and a page cluster is selected as a victim for replacement when a new write buffer page is needed. Also, it assumes that all pages in the victim cluster are passed to the FTL to be written to the flash memory, in the order of page number. The term *Optimistic* is based on this assumption.

The Optimistic FTL associates one log block to each modified data block, like BAST. It maintains a block mapping table (BMT) and a block association table (BAT) which associates modified data blocks with corresponding log blocks. These two data structures are mandatory for log block-based FTLs. The Optimistic FTL always maintains *complete* and *sequential* log blocks. Let P_i denote the page with index i ($1 \leq i \leq 64$) and P_{last} be the page with the highest page index in a log block. The *complete* log block means that there is no missing page between P_1 and P_{last} . The *sequential* log block means that all pages in the log block are stored sequentially in increasing order of indices. By maintaining *complete* and *sequential* log blocks, the Optimistic FTL not only provides stable merge latencies but also decreases the page mapping overhead. In Optimistic FTL, different from BAST or FAST, a sector mapping table is not used. Instead, a new field called *Last Page Index (LPI)* is added to the BAT. *LPI* stores the index of the last page stored in the log block.

Fig. 11 shows the log block management scheme in Optimistic FTL. Let N_v denote the number of pages in the victim cluster, and I_{min} and I_{max} be the smallest and largest page index in the victim cluster, respectively. Let N_B be the block size in number of pages. For large block flash memories, $N_B = 64$.

If all pages in the victim cluster have larger indices than the *LPI* of the associated log block (Fig. 11a), pages whose indices lie between the *LPI* and I_{min} are copied from the data block to the log block, and then pages in the victim cluster are written, sequentially, to the log block. If pages in the victim cluster are not sequential, (for example, there are pages 5 and 7 in the victim cluster) the missing pages (page 6, in the example) are copied from the data block. After writing all pages in the victim cluster, the *LPI* is modified to the last page index in the log block (in Fig. 11a, *LPI* becomes 6). We call this an *append* operation. In the *append* operation, assuming large block Flash, 62 valid page

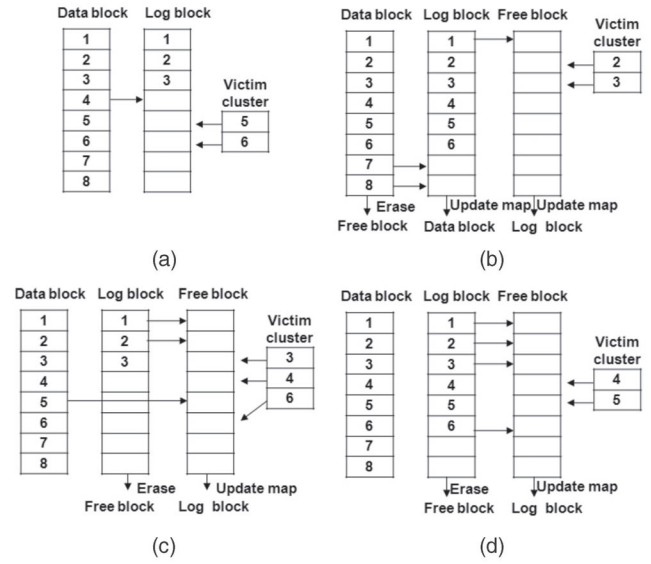


Fig. 11. Log block management in Optimistic FTL. (a) *Append*, (b) *Data block switch*, (c) *Log block switch*, and (d) *Log block switch*.

copies (from data block to log block) and one page write operation can occur in the worst case, where there is only one page in the log block and the victim page contains only the last page (whose index is 64). No erase operation is needed for the *append* operation. When a victim cluster, whose corresponding data block has no associated log block, is evicted from the write buffer, a new log block is assigned to the data block with $LPI = 0$. Optimistic FTL performs the *append* operation for the victim cluster. When the log block becomes full after the *append* operation, the data block is erased and the log block is *switched* to the data block.

When only part of the page indices in the victim cluster are smaller than or equal to the *LPI* (Fig. 11c), the Optimistic FTL assigns a new log block. Pages whose indices are smaller than I_{min} are copied from the old log block to the new log block. Then, pages in the victim cluster are written sequentially to the new log block. In the mean time, missing pages in the victim cluster are copied from either the data block or the old log block. In this case, all pages which are in the old log block but are not in the victim cluster are copied to the new log block. After writing all pages to the new log block, the old log block is erased and the *LPI* of the new log block is set to the last page index in the block (in Fig. 11b, *LPI* becomes 6). We call this a *Log block switch* operation. In this operation, 62 valid page copies (from either the data block or the old log block to the new log block), two page write and one block erase operations can occur in the worst case, where the victim cluster contains only two pages: 1) the page whose index equals the *LPI* of the old log block and 2) the last page in the block. When the victim cluster contains the last page in the block, the log block becomes full after the *log block switch* operation. Then, the data block is erased and the log block is *switched* to the data block.

The case where all pages in the victim cluster also exist in the log block (i.e., the largest page index in the victim cluster is smaller than or equal to the *LPI* of the log block) can be further divided into two detailed cases according to

the cost for making a new *complete and sequential* log block. To make a *complete and sequential* log block, pages that are not in the victim cluster but in the old log block should be copied to the new log block (Fig. 11d). The *Log block switch* operation can be used for this case and total $LPI - N_v$ valid page copies are necessary, which can be very expensive when LPI is large. When the old log block contains a large number of pages (large LPI), instead of making a new log block through a large number of valid page copies, switching the old log block to the data block can be more efficient (Fig. 11b). To make a data block from the old log block, missing pages in the log block should be copied from the original data block. Since LPI of the log block is large, the number of missing pages is small. Also, the new log block is enough to store only those pages whose indices are smaller than or equal to I_{max} . We call this a *Data block switch* operation.

The *Data block switch* operation consists of four steps:

1. valid page copies for pages whose indices are larger than LPI , from the original data block to the old log block (total $N_B - LPI$ copies),
2. valid page copies for pages whose indices are $1, \dots, I_{min} - 1$ from the old log block to the new log block,
3. writing N_v pages from the victim cluster to the new log block (in this step, valid page copies for missing pages in the victim cluster, whose indices lie between I_{min} and I_{max} , also occur), and
4. erasing the original data block and updating BMT and BAT.

The *Log block switch* operation, in the above case, consists of four steps where the first two steps are the same as steps 2 and 3 in the *Data block switch* operation. The remaining steps are: 3) valid page copies for pages whose indices are $I_{max} + 1, \dots, LPI$, from the old log block to the new log block (total $LPI - I_{max}$ copies), and 4) erasing the old log block and updating BAT. Assuming that the cost for updating BMT is small, the cost difference between the two operations arises from step 1 in the *Data block switch* operation and step 3 in the *Log block switch* operation. Hence, when $(N_B - LPI) > LPI - I_{max}$, then the *Log block switch* operation is more efficient than the *Data block switch* operation, and vice versa. In case of Fig. 11c, $LPI - I_{max} < 0$. Hence, we can combine two cases, Figs. 11c and 11d, into one case in which $(N_B - LPI) > LPI - I_{max}$ is satisfied.

Algorithm 1. Log Block Management Algorithm

Notations:

$\mathcal{VC}$: Victim cluster LB : log block
 DB : Data block LPI_{new} : new LPI

- 1: **if** the size of $\mathcal{VC}$ == block size **then**
- 2: DB_{new} = newly assigned data block;
- 3: write pages in the $\mathcal{VC}$ to DB_{new} ;
- 4: erase old DB and update BMT;
- 5: **return**;
- 6: **end if**
- 7: **if** corresponding LB does not exist **then**
- 8: allocate a LB ; // selecting victim log block can be needed

- 9: update BAT with $LPI = 0$;
- 10: **end if**
- 11: **if** $I_{min} > LPI$ **then** // do append
- 12: $LPI_{new} = I_{max}$;
- 13: **for** $i, i = LPI + 1, \dots, LPI_{new}$ **do**
- 14: **if** $page[i] \in \mathcal{VC}$ **then**
- 15: write $page[i]$ to $LB[i]$;
- 16: **else**
- 17: copy $DB[i]$ to $LB[i]$;
- 18: **end if**
- 19: **end for**
- 20: **else**
- 21: LB_{new} = newly assigned log block;
- 22: // selecting victim log block can be needed
- 23: **if** $(N_B - LPI) > (LPI - I_{max})$ **then**
- 24: // do Log block switch
- 25: $LPI_{new} = \max\{LPI, I_{max}\}$;
- 26: **for** $i, i = 1, \dots, LPI_{new}$ **do**
- 27: **if** $page[i] \in \mathcal{VC}$ **then**
- 28: write $page[i]$ to $LB_{new}[i]$;
- 29: **else**
- 30: **if** $page[i] \in LB$ **then**
- 31: copy $LB[i]$ to $LB_{new}[i]$;
- 32: **end if**
- 33: **else**
- 34: copy $DB[i]$ to $LB_{new}[i]$;
- 35: **end if**
- 36: **end for**
- 37: erase LB ; update BAT;
- 38: **else** // do Data block switch
- 39: $LPI_{new} = I_{max}$;
- 40: **for** $i, i = 1, \dots, LPI_{new}$ **do**
- 41: **if** $page[i] \in \mathcal{VC}$ **then**
- 42: write $page[i]$ to $LB_{new}[i]$;
- 43: **else** // the page is in the LB
- 44: copy $LB[i]$ to $LB_{new}[i]$;
- 45: **end if**
- 46: **end for**
- 47: **for** $i, i = LPI + 1, \dots, N_B$ **do** // fill up log block
- 48: copy $DB[i]$ to $LB[i]$;
- 49: **end for**
- 50: erase DB ; update BMT and BAT;
- 51: **end if**
- 52: update BAT such that $LPI = LPI_{new}$;

The log block management scheme is formalized in Algorithm 1. If the victim cluster contains all pages for the block, it is not necessary to maintain the log block. Hence, the Optimistic FTL replaces the corresponding data block with the new data block and updates BMT (steps 1-6 in Algorithm 1).

When the victim cluster overlaps with the current log block (step 20 in Algorithm 1), a new log block is assigned to replace the old one. At the time, if there is no free log block, a victim log block is selected to be merged with the corresponding data block. Since all log blocks in the Optimistic FTL are *complete and sequential*, a *partial merge* (Fig. 1b) operation is performed.

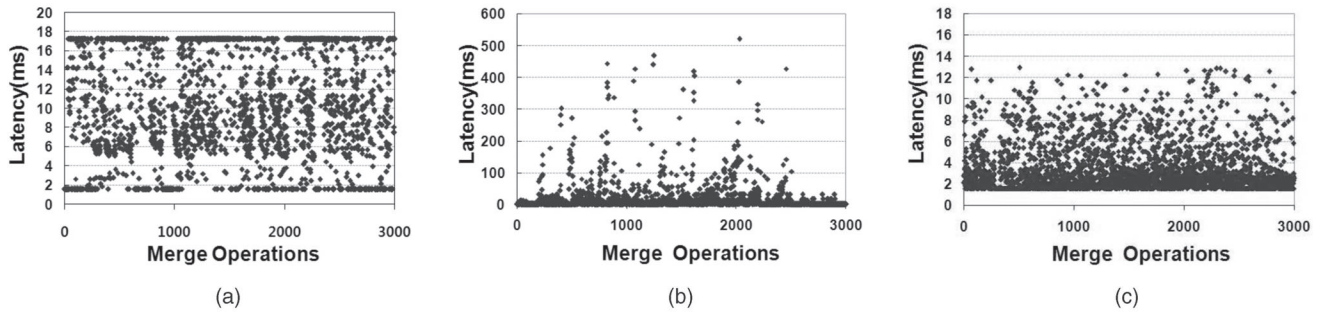


Fig. 12. Merge latencies in each FTL algorithm: in Optimistic FTL, latencies for *log block switch*, *data block switch*, and *partial merge* are plotted. (FAT trace). (a) BAST. (b) FAST. (c) Optimistic FTL.

The Optimistic FTL is much simpler than previous log block-based FTL algorithms, such as BAST and FAST. It maintains only BMT and BAT in the memory. The BAST and FAST algorithms maintain not only BMT and BAT but also a sector mapping table for the log block. Optimistic FTL does not maintain a sector mapping table since it does not use sector mapping for the log block. Also, Optimistic FTL uses *partial merge*, *append*, *Data block switch*, and *Log block switch* operations, which are far cheaper than *full merge* operations. When a *full merge* operation occurs, in BAST, while merging a data block and a log block, 64 read, 64 write and 2 erase operations are always performed, assuming a large block flash memory. In FAST, each page in the log block should be merged with its original data block. Hence, assuming the worst case, where all pages in the log block are from different data blocks, 4,096 (64×64) read, 4,096 write and 65 erase operations are performed. However, in Optimistic FTL, 63 read, 64 write and 1 erase operations are performed even in the worst case of *Log block switch* or *Data block switch* operations.³

Fig. 12 shows the latencies of merge operations in each FTL algorithm. The *x*-axis represents the sequence of 3,000 merge operations and the *y*-axis shows the latency of each merge operation. Since the merge operation greatly affects the response time to the write request from the above layer (file system), the merge latency is one of the important factors in designing FTL algorithm. Since the latency of a merge operation in FAST largely depends on the number of corresponding data blocks for each page in the log block, latencies of merge operations in FAST show a large deviation (Fig. 12b). As we can see from Figs. 12a and 12c, BAST and Optimistic FTL show very stable merge latency, and the average merge latency in Optimistic FTL is much lower than that in BAST.

Fig. 13 shows two important considerations for making elaborate Optimistic FTL. In this experiment, we used CLC for the write buffer management policy and Optimistic FTL with 128 log blocks is used for the underlying FTL algorithm. Fig. 13a shows the cumulative distribution of the interference interval of each victim cluster. The *x*-axis represents the time that a victim cluster evicted from the write buffer has to stay in the log block until a new victim cluster with the same corresponding data block will be

evicted from the write buffer. If the time is small, the probability that an *append* operation will occur becomes large, which enables efficient use of log blocks by Optimistic FTL. Interval 0 means that no future victim cluster having the same corresponding data block with the current victim cluster will be evicted from the write buffer. For those victim clusters, not only *append* but also *data block switch* or *log block switch* operations will occur. Hence, assuming an extreme case where the intervals of all victim clusters are 0, only a *partial merge* operation will be performed to secure free log blocks. We can see from Fig. 13a that the interference interval becomes large as the write buffer

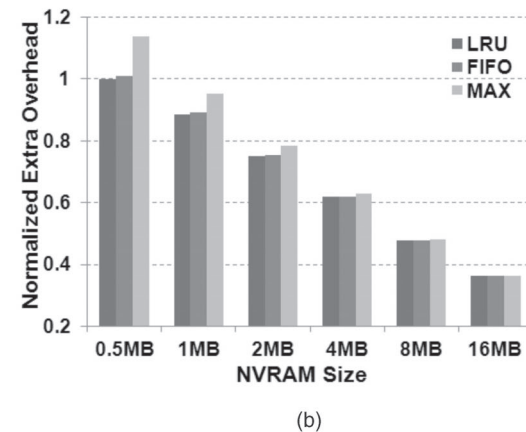
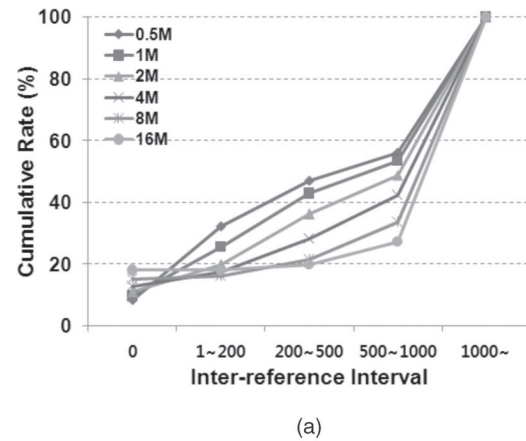


Fig. 13. Considerations for tuning Optimistic FTL (FAT trace). (a) Distribution of the interference intervals (b) Effect of the log block replacement scheme.

3. As described earlier, one more erase operation is needed if the log block becomes full after an *append* or *Log block switch* operation. However, this does not occur frequently.

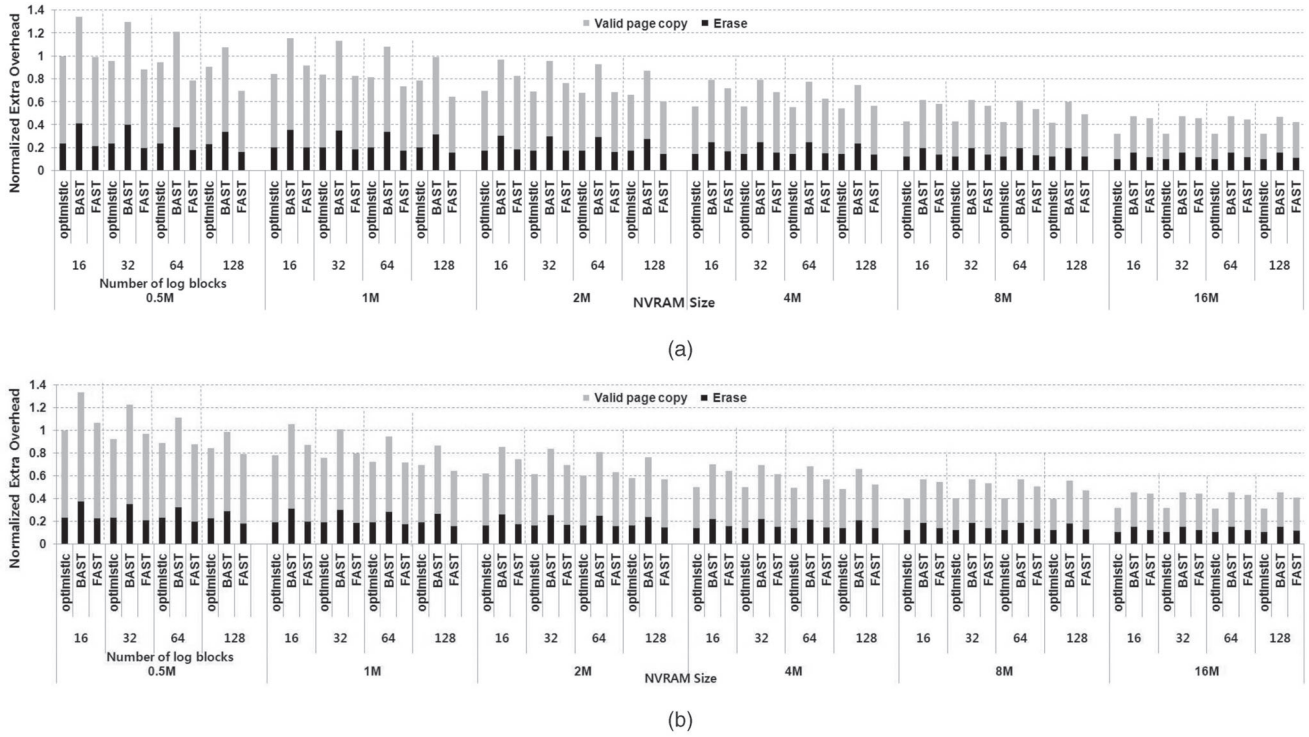


Fig. 14. Extra overhead in each FTL algorithm. (a) FAT trace. (b) NTFS trace.

size increases. As the write buffer size increases, the size of the page clusters in the buffer also increases because a larger portion of rereferences to each cluster are absorbed in the write buffer. Hence, the probability that a victim cluster evicted from the write buffer will be referenced again decreases as the write buffer size increases.

When all log blocks are associated with data blocks, it is necessary to select and merge a victim log block to make a free log block. At this time, the victim selection scheme can affect the overall performance of Optimistic FTL. Fig. 13b shows the performance of Optimistic FTL with three different victim selection schemes. In MAX, the log block that has the largest *LPI* value is selected for the victim. Since MAX selects the log block which has the largest number of pages in it, the average number of valid page copies in each merge operation is minimized. However, it has the same drawbacks as the LC write buffer management policy, which results in more erase operations than others. LRU and FIFO showed similar performance in all cases, which means that the destaged victim clusters from the write buffer show very weak temporal locality. Also, we can see from the figure that the performance gap among the three schemes decreases as the write buffer size increases. As the write buffer size increases, the probability that a victim cluster will be appended to an existing log block decreases. Hence, the effect of the victim selection scheme on the overall performance also decreases.

Fig. 14 shows the performance of three log block-based FTL algorithms with various numbers of log blocks for each NVRAM buffer size. The CLC policy is used for write buffer management. We measured the time for extra operations (valid page copies and erase operations), which are invoked

by merge operations (BAST and FAST) or *append*, *log block switch*, and *data block switch* operations (Optimistic FTL). The extra overhead is normalized such that the overhead for Optimistic FTL with 16 log blocks and 512 KBytes NVRAM is 1. As we can see from the figure, Optimistic FTL outperforms BAST in all cases and FAST when the NVRAM size is larger than or equal to 2 Mbytes (except when the number of log blocks is 128). While FAST looks to be competitive when the number of log blocks are large, the high complexity of *merge* operation in FAST makes its merge latencies very high and unstable, which can be a critical problem for flash memory-based storage devices. Also, when the NVRAM size is large, the number of log blocks in FAST does not largely affect the overall performance. We can conclude, based on the results, that using NVRAM as a write buffer for a flash memory-based storage system not only necessitates a write buffer-aware FTL algorithm for performance improvement but also can simplify the FTL algorithm. Optimistic FTL can be an efficient write buffer-aware FTL algorithm.

Fig. 15 compares the performance of the proposed scheme (CLC + Optimistic FTL) with BPLRU + BAST. When a victim cluster is selected, BPLRU reads pages which are not in the victim cluster from the data block and combines them with the pages in the victim cluster to make a full block. Then, it flushes the full block to the FTL, which, in turn, performs a switch merge operation. The overhead for all operations is exactly the same as with the *partial merge* operation in traditional log block based FTL algorithms. The underlying FTL has nothing to do except to perform a switch merge operation. Hence, the performance of the BPLRU is not affected by the underlying FTL algorithm. Actually, the performance of BPLRU + FAST was identical

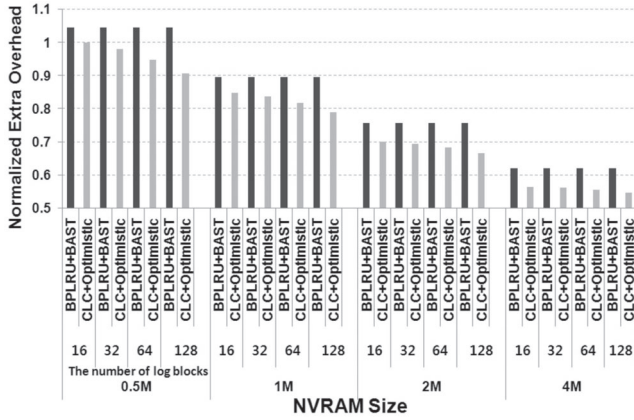


Fig. 15. Performance comparison between the proposed scheme and BPLRU (FAT trace).

with BPLRU + BAST. Also, since the BPLRU does not use log blocks, the number of log blocks does not affect the performance of the BPLRU, as we can see from Fig. 15. In fact, BPLRU can be seen as an integrated scheme that consists of the LRU-C write buffer management policy and a simple write buffer-aware FTL algorithm that only performs valid page copies for missing pages in the victim cluster and switches old and new data blocks. That is why we included BPLRU for performance comparison in this section but excluded it in Section 4.2.

We can see from Fig. 15 that CLC + Optimistic FTL always outperforms BPLRU. While BPLRU could remove both the full and partial merge operations from the FTL, it cannot show good performance when the size of the victim cluster is small. When the victim cluster is small, too many pages need to be copied from the original data block to the new data block (valid page copies). Hence, it shows good performance only when the block write access locality is large enough to make page clusters in the write buffer become sufficiently large. In Optimistic FTL, pages in the small victim cluster can be appended to the corresponding log block by an *append* operation, which requires not only a relatively small number of valid page copies but also no erase operation. In fact, the effect of the block padding scheme in BPLRU is identical with Optimistic FTL when all interference intervals of victim clusters are 0, which is the worst case for Optimistic FTL.

Fig. 16 shows the overall performance comparison between the traditional schemes and the proposed scheme (CLC with $\alpha = 0.1$ + Optimistic FTL). The number of log blocks for FTL algorithms is 128. We can see that the proposed scheme outperforms in all cases. Based on the result, we can say that the proposed scheme can be an outstanding choice for an NVRAM write buffer management scheme with flash memory-based storage devices.

6 CONCLUSION

Recently, high performance nonvolatile random access memories (NVRAM) such as FeRAM, PRAM and MRAM, have emerged in the marketplace, and the capacity of the NVRAM is predicted to increase rapidly. There can be various ways to exploit NVRAM in the computer system

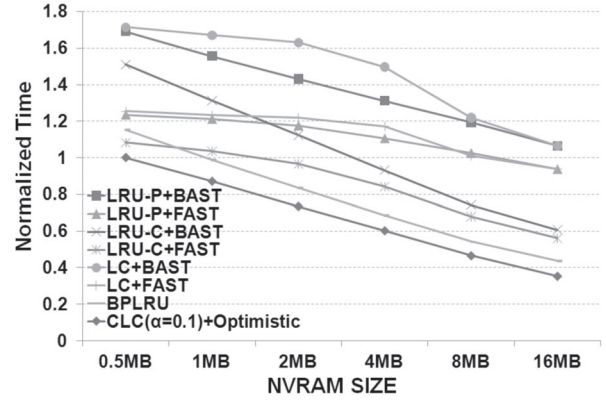


Fig. 16. Overall performance comparison between the traditional and the proposed schemes (FAT trace).

and using NVRAM as a write buffer could be one of them. In this paper, we examine various NVRAM write buffer management policies, LRU-P, LRU-C, and LC, among which the latter two policies are based on page clustering. The LRU-C policy exploits the temporal locality of block accesses to increase the hit ratio in the write buffer, and the LC policy attempts to maximize the number of simultaneously destaged pages in order to minimize the overhead of merge operations. The proposed policy, CLC, also clusters pages belonging to the same block in the flash memory so that the page cluster matches the erasure unit of flash memory. The CLC policy not only exploits the temporal locality but also maximizes the number of simultaneously destaged pages. Simulation results have shown that the CLC policy outperforms traditional page-level LRU policy (LRU-P) by a maximum of 51 percent.

Log block based FTL algorithms, such as BAST and FAST, for flash memory-based storage systems show poor performance when the write pattern is small and random. Hence, the write buffer management policy for NAND flash memory-based storage systems should be designed to function with the behavior of the FTL. Using an NVRAM write buffer for page clustering, small random writes can be transformed into sequential writes to the flash memory. The sequential writes to the flash memory decrease the necessity of sector mapping for log blocks. If we can keep the ordered sequence of pages in a log block without large overhead, we can remove the sector mapping table, which requires not only much memory space but also page search overhead. Also, it can simplify the complicated merge process by removing the *full merge* operation. In this paper, we also proposed a write buffer-aware FTL algorithm, Optimistic FTL, which includes three novel operations—*append*, *log block switch*, and *data block switch*. The Optimistic FTL algorithm not only removes the necessity of a page mapping table for the log blocks, which enables a faster page read operation, but also provides low and stable merge latencies. Simulation results showed that the Optimistic FTL outperforms traditional write buffer-unaware log block-based FTL algorithms. Also, the combination of the proposed write buffer management policy (CLC) and the Optimistic algorithm always outperformed all other combinations of traditional schemes.

ACKNOWLEDGMENTS

This work was supported by grant No. R01-2007-000-20649-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

REFERENCES

- [1] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," *Proc. Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2006.
- [2] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: Flash-Aware Buffer Management Policy for Portable Media Players," *IEEE Trans. Consumer Electronics*, vol. 52, no. 2, pp. 485-493, May 2006.
- [3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, M. Seltzer, "Non-Volatile Memory for Fast, Reliable File Systems," *Operating Systems Rev.*, vol. 26, pp. 10-22, Oct. 1992.
- [4] J.-F. Paris, T.R. Haining, and D.E. Long, "A Stack Model Based Replacement Policy for A Non-Volatile Cache," *Proc. IEEE Symp. Mass Storage Systems*, pp. 217-224, Mar. 2000.
- [5] B. Gill and D.S. Modha, "WOW: Wise Ordering for Writes Combining Spatial and Temporal Locality in Non-Volatile Caches," *Proc. USENIX Conf. File and Storage Technologies (FAST)*, Dec. 2005.
- [6] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1994.
- [7] F. Douglass, F. Kaashoek, K. Li, R. Cceres, B. Marsh, and J.A. Tauber, "Storage Alternatives for Mobile Computers," *Proc. Operating Systems Design and Implementation*, 1994.
- [8] K. Kim and G.-H. Koh, "Future Memory Technology Including Emerging New Memories," *Proc. Int'l Conf. Microelectronics*, May 2004.
- [9] "Samsung Electronics 1gx8bit/2gx16bit NAND Flash Memory," <http://www.samsung.com/Products/Semiconductor/NAND-Flash/SLCLargeBlock/16Gbit/K9WAG08U1M/K9WAG08U1M.htm>, 2009.
- [10] <http://www.techshout.com/hardware/2006/14/512mbit-pram-phase-change-ram-announced-by-samsung/>, 2009.
- [11] F.B. et al, "A Multi-Level-Cell Bipolar-Selected Phase-Change Memory," *Proc. Int'l Solid State Circuits Conf.*, Feb. 2008.
- [12] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification," 1998.
- [13] T. Shinohara, "Flash Memory Card with Block Memory Address Arrangement," US Patent no. 5,905,993, 1999.
- [14] M-Systems, "Flash-Memory Translation Layer for NAND Flash (NFTL)."
- [15] L.-P. Chang and T.-W. Kuo, "An Adaptive Stripping Architecture for Flash Memory Storage Systems of Embedded Systems," *Proc. IEEE Eighth Real-Time and Embedded Technology and Applications Symp. (RTAS)*, Sept. 2002.
- [16] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compact Flash Systems," *IEEE Trans. Consumer Electronics*, vol. 48, no. 2, pp. 366-375, May 2002.
- [17] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer Based Flash Translation Layer Using Fully Associative Sector Translation," *ACM Trans. Embedded Computing Systems*, vol. 6, no. 3, 2007.
- [18] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys*, vol. 37, no. 2 pp. 138-163, June 2005.
- [19] Aleph One Company "Yet Another Flash Filing System."
- [20] D. Woodhouse, "JFFS: The Journaling Flash File System."
- [21] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A Superblock-Based Flash Translation Layer for NAND Flash Memory," *Proc. Sixth Ann. ACM Conf. Embedded Systems Software*, Oct. 2006.
- [22] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," *Proc. Sixth USENIX Conf. File and Storage Technologies*, Feb. 2008.
- [23] Embedded Systems and Wireless Networking Lab., <http://news.lab.csie.ntu.edu.tw/~flash/index.php?SelectedItem=Traces>.



Sooyong Kang received the BS degree in mathematics and the MS and PhD degrees in computer science from Seoul National University (SNU), Korea, in 1996, 1998, and 2002, respectively. He was then a postdoctoral researcher in the School of Computer Science and Engineering, SNU. He is now with the Division of Information and Communications, Hanyang University, Seoul. His research interests include operating systems, multimedia systems, storage systems, flash memories and next-generation nonvolatile memories, and distributed computing systems.



Sungmin Park received the BS degree in computer science education and the MS degree in electronics and computer engineering from Hanyang University in 2005 and 2007, respectively. He is currently working toward the PhD degree at the School of Electronics and Computer Engineering, Hanyang University. His research interests include operating system and flash memory-based storage system.



Hoyoung Jung received the BS degree in material science and engineering and the MS degree in information and communications from Hanyang University, Korea, in 2004 and 2006, respectively. He is currently working toward the PhD degree at the School of Electronics and Computer Engineering, Hanyang University. His research interests include DBMS, flash memory-based storage system, and embedded system.



Hyoki Shim received the BS degree in urban engineering and the MS degree in information and communications from Hanyang University, Korea, in 2005 and 2008, respectively. He is currently working toward the PhD degree at the School of Electronics and Computer Engineering, Hanyang University. His research interests include database systems, operating system, and flash memory-based storage system.



Jaehyuk Cha received the BS, MS, and PhD degrees in computer science, all from Seoul National University (SNU), Korea, in 1987, 1991, and 1997, respectively. He was with the Korea Research Information Center (KRIC) from 1997 to 1998. He is now an associate professor at the Division of Information and Communications, Hanyang University. His research interests include XML, DBMS, flash memory-based storage system, multimedia content adaptation, and e-Learning.

► For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.