BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage

Hyojun Kim and Seongjun Ahn
Software Laboratory of Samsung Electronics, Korea
{*zartoven, seongjun.ahn*}*@samsung.com*

Abstract

Flash memory has become the most important storage media in mobile devices, and is beginning to replace hard disks in desktop systems. However, its relatively poor random write performance may cause problems in the desktop environment, which has much more complicated requirements than mobile devices. While a RAM buffer has been quite successful in hard disks to mask the low efficiency of random writes, managing such a buffer to fully exploit the characteristics of flash storage has still not been resolved. In this paper, we propose a new write buffer management scheme called Block Padding Least Recently Used, which significantly improves the random write performance of flash storage. We evaluate the scheme using trace-driven simulations and experiments with a prototype implementation. It shows about 44% enhanced performance for the workload of MS Office 2003 installation.

1 Introduction

Flash memory has many attractive features such as low power consumption, small size, light weight, and shock resistance [3]. Because of these features, flash memory is widely used in portable storage devices and handheld devices. Recently, flash memory has been adopted by personal computers and servers in the form of onboard cache and solid-state disk (SSD).

Flash memory-based SSDs exhibit much better performance for random reads compared to hard disks because NAND flash memory does not have a seek delay. In a hard disk, the seek delay can be up to several milliseconds. For sequential read and write requests, an SSD has a similar or better performance than a hard disk [4]. However, SSDs exhibit worse performance for random writes due to the unique physical characteristics of NAND flash memory. The memory must be erased before it can be written. The unit of erase operation is relatively large, typically a block composed of multiple pages, where a page is the access unit. To mask this mismatch between write and erase operations, SSDs use additional software, called the flash translation layer (FTL) [6, 14] whose function is to map the storage interface logical blocks to physical pages within the device. The SSD random write performance is highly dependent on the effectiveness of the FTL algorithm.

Different types of FTL algorithms exist. Mobile phones use relatively complicated algorithms, but simpler methods are used for flash memory cards and USB mass storage disks. In SSDs, the controller has restricted computing power and working random access memory (RAM) to manage a large quantity of NAND flash memory, up to tens of gigabytes. Therefore, the SSD FTL must attain the cost efficiency of flash memory cards rather than mobile phones.

To obtain better performance with restricted resources, some FTLs exploit locality in write requests. A small portion of the flash memory is set aside for use as a write buffer to compensate for the physical characteristics of NAND flash memory. With high access locality, the small write buffer can be effective. However, FTLs show poor performance for random writes with no locality. The poor performance of SSDs for random writes can significantly impact desktop and server systems, which may have more complicated write patterns; in these systems, multiple threads commonly request I/O jobs concurrently, resulting in complex write access patterns. Therefore, the random write performance is of increasing importance in SSD design.

Several different approaches exist to enhancing random write performance. We selected a method of using a RAM buffer inside the SSD because it is very realistic and can be easily applied to current SSD products regardless of their FTL algorithms. The issue, however, is how to use the RAM buffer properly. In the case of a hard disk, the elevator algorithm is used to minimize the head movements.

In this paper, we present a new write buffer management scheme called Block Padding Least Recently Used (BPLRU) to enhance the random write performance of flash storage. BPLRU considers the common FTL characteristics and attempts to establish a desirable write pattern with RAM buffering. More specifically, BPLRU uses three key techniques, block-level LRU, page padding, and LRU compensation. Block-level LRU updates the LRU list considering the size of the erasable block to minimize the number of merge operations in the FTL. Page padding changes the fragmented write patterns to sequential ones to reduce the buffer flushing cost. LRU compensation adjusts the LRU list to use RAM for random writes more effectively. Using write traces from several typical tasks and three different file systems, we show that BPLRU is much more effective than previously proposed schemes, both in simulation and in experiments with a real prototype.

The rest of the paper is organized as follows. Section 2 presents the background and related work. Section 3 explains the proposed buffer cache management schemes. Section 4 evaluates the BPLRU scheme, and Section 5 contains our conclusions.

2 Background and Related Work

2.1 Flash Memory

There are two types of flash memories: NOR and NAND flash memories [18]. NOR flash memory was developed to replace programmable read-only memory (PROM) and erasable PROM (EPROM), which were used for code storage; so it was designed for efficient random access. It has separate address and data buses like EPROM and static random access memory (SRAM). NAND-type flash memory was developed more recently for data storage, and so was designed to have a denser architecture and a simpler interface than NOR flash memory. NAND flash memory is widely used.

Flash memories have a common physical restriction; they must be erased before writing. In flash memory, the existence of an electric charge represents 1 or 0, and the charges can be moved to or from a transistor by an erase or write operation. Generally, the erase operation, which makes a storage cell represent 1, takes longer than the write operation, so its operation unit was designed to be bigger than the write operation for better performance. Thus, flash memory can be written or read a single page at a time, but it can be erased only block by block. A block consists of a certain number of pages. The size of a page ranges from a word to 4 KB depending on the type of device. In NAND flash memory, a page is similar to a hard disk sector and is usually 2 KB. For NOR type flash memory, the size of a page is just one word.

Flash memory also suffers from a limitation in the number of erase operations possible for each block. The insulation layer that prevents electric charges from dispersing may be damaged after a certain number of erase operations. In single level cell (SLC) NAND flash memory, the expected number of erasures per a block is 100,000 and this is reduced to 10,000 in multilevel cell (MLC) NAND flash memory. If some blocks that contain critical information are worn out, the whole memory becomes useless even though many serviceable blocks still exist. Therefore, many flash memory-based devices use wear-leveling techniques to ensure that blocks wear out evenly.

2.2 Flash Translation Layer

The FTL overcomes the physical restriction of flash memory by remapping the logical blocks exported by a storage interface to physical locations within individual pages [6]. It emulates a hard disk, and provides logical sector updates[1](Figure 1). The early FTLs used a log-structured architecture [23] in which logical sectors were appended to the end of a large log, and obsolete sectors were removed through a garbage collection process. This architecture is well suited for flash memory because it cannot be overwritten. We call this method *page mapping FTL* because a logical sector (or page) can be written to any physical page [14]. The FTL writes the requested sector to a suitable empty page, and it maintains the mapping information between the logical sector and the physical page separately in both flash and main memories because the information is necessary to read the sector later.

Page-mapping FTL sufficed for small flash memory sizes; its mapping information size was also small. However, with the development of NAND flash memory and its exponential size increase, the page-mapping method became ineffective. It requires a great deal of memory for its mapping information. In some cases, the mapping information must be reconstructed by scanning the whole flash memory at start-up, and this may result in long mount time. Therefore, a new memory-efficient algorithm was required for very large NAND flash memories.

The *block mapping FTL* which is used for the Smart Media card [26], is not particularly efficient because a sector update may require a whole block update. An improvement on this scheme, called the *hybrid mapping FTL* [15], manages block-level mapping like *block map-*

[1]Even though the term *sector* represents physical block of data on a hard disk, it is commonly used as an access unit for the FTL because it emulates a hard disk. The size of logical sector in the FTL may be 512 B, 2 KB, or 4 KB for efficiency. We adopt the same convention in this paper.

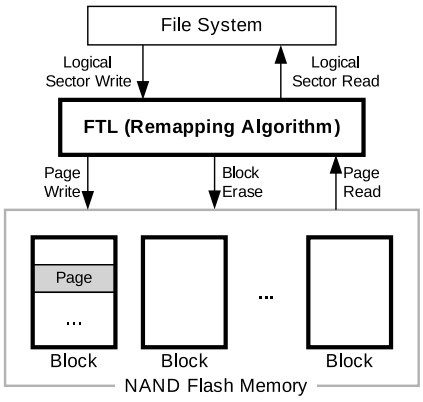


Figure 1: FTL and NAND flash memory. FTL emulates sector read / write functionalities of a hard disk to use conventional disk file systems on NAND flash memory.

ping FTL, but the sector position is not fixed inside the block. While this requires additional offset-level mapping information, the memory requirement is much less than in page mapping FTL.

Other FTLs were designed to exploit the locality of the write requests. If write requests are concentrated in a certain address range, some reserved blocks not mapped to any externally visible logical sectors can be used temporarily for those frequently updated logical sectors. When we consider the usage pattern of the flash storage, this is quite reasonable. Because the number of reserved blocks is limited, more flexible and efficient mapping algorithms can be applied to the reserved blocks while most data blocks use simple block mapping.

The *replacement block algorithm* [2] assigns multiple physical blocks to one logical block. It only requires block-level mapping information, which represents a physical block mapped to a particular logical block, and its creation order when multiple physical blocks exist. The *log-block FTL algorithm* [17] combines a coarse-grained mapping policy of a block mapping method with a fine-grained mapping policy of page mapping inside a block. Compared to the replacement block algorithm, it requires more mapping tables for log blocks, but can use reserved blocks more effectively. The log-block FTL algorithm is one of the most popular algorithms today because it combines competitive performance with rather low cost in terms of RAM usage and CPU power.

2.3 Log-block FTL

In the log-block FTL algorithm, sectors are always written to log blocks that use a fine-grained mapping policy allowing a sector to be in any position in a block. This is very efficient for concentrated updates. For example, if Sector 0 is repeatedly written four times when a block consists of four pages, all pages in the log block can be used only for sector 0. When a log block becomes full, it merges with the old data block to make a new data block. The valid sectors in the log block and in the old data block are copied to a free block, and the free block becomes the new data block. Then, the log block and the old data block become free blocks.

Sometimes, a log block can just replace the old data block. If sectors are written to a log block from its first sector to the last sector sequentially, the log block gains the same status as the data block. In this case, the log block will simply replace the old data block, and the old data block will become a free block. This replacement is called the *switch merge* in log-block FTL.

Switch merge provides the ideal performance for NAND flash memory. It requires single-block erasure and N page writes, where N is the number of pages per block. To distinguish it from a *switch merge*, we refer to a normal merge as a *full merge* in this paper. The *full merge* requires two block erasures, N page reads, and N page writes.

2.4 Flash Aware Caches

Clean first LRU (CFLRU) [21, 22] is a buffer cache management algorithm for flash storage. It was proposed to exploit the asymmetric performance of flash memory read and write operations. It attempts to choose a clean page as a victim rather than dirty pages because writing cost is much more expensive. CFLRU was found to be able to reduce the average replacement cost by 26% in the buffer cache compared to the LRU algorithm. CFLRU is important because it reduces the number of writes by trading off the number of reads. However, this is irrelevant when only write requests are involved. Thus, it is not useful for enhancing random write performance.

The *flash aware buffer policy* (FAB) [10] is another buffer cache management policy used for flash memory. In FAB, the buffers that belong to the same erasable block of flash memory are grouped together, and the groups are maintained in LRU order: a group is moved to the beginning of the list when a buffer in the group is read or updated, or a new buffer is added to the group. When all buffers are full, a group that has the largest number of buffers is selected as victim. If more than one group has the same largest number of buffers, the least recently used of them is selected as a victim. All the dirty buffers in the victim group are flushed, and all the clean buffers in it are discarded. The main use of FAB is in portable media player applications in which the majority of write requests are sequential. FAB is very effective compared to LRU. Note that BPLRU targets random write patterns.

Jiang et al. proposed DULO [8], an effective buffer

cache management scheme to exploit both temporal and spatial locality. Even though it is not designed for flash storage, the basic concept of DULO is very similar to our proposed method. In DULO, the characteristics of a hard disk are exploited so that sequential access is more efficient than random access. Similarly, flash storage shows optimized write performance for sequential writes in a block boundary because most FTLs use the spatial locality of the block level. Therefore, we use a RAM buffer to create the desirable write pattern while the existing buffer caches are used to try to reduce the number of write requests.

Recently developed SSDs for desktop or server applications face quite different access patterns compared to mobile applications. In server systems, multiple processes may request disk access concurrently, which can be regarded as essentially as a random pattern because many processes exist.

An SSD has very different performance characteristics compared to a hard disk. For random read requests, the SSD is much better than a hard disk because it does not have any mechanical parts. This is one of the most important advantages of SSDs, offset, however, by the poor random write performance. Due to the characteristics of NAND flash memory and the FTL algorithm inside an SSD, the performance is now much worse than a hard disk and the life span of the SSD can be reduced by random write patterns.

Two different approaches are possible. First, the FTL may use a more flexible mapping algorithm such as page mapping, hybrid mapping, or the super-block FTL algorithm [13]. However, flexible mapping algorithms generally require more RAM, CPU power, and a long start-up time, so they may not be feasible for SSDs. The second approach involves embedding the RAM buffer in the SSDs just like in a hard disk. We propose a method for proper management of this write buffer.

3 BPLRU

We devised BPLRU as a buffer management scheme to be applied to the write buffer inside SSDs. BPLRU allocates and manages buffer memory only for write requests. For reads, it simply redirects the requests to the FTL. We chose to use all of available RAM inside an SSD as write buffers because most desktop and server computers have a much larger host cache that can absorb repeated read requests to the same block more effectively than the limited RAM on the SSD device.

Figure 2 shows the general system configuration considered in this paper. The host includes a CPU, a file system, and a device-level buffer cache on the host side. The SSD device includes the RAM for buffering writes, the FTL, and the flash memory itself. For the host buffer cache policy, CFLRU can be applied to reduce the number of write requests by trading off the number of reads. We do not assume any special host side cache policy in this paper.

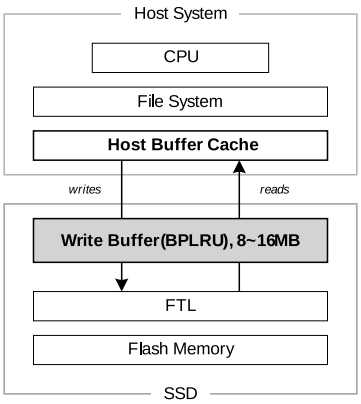


Figure 2: System configuration. BPLRU, the proposed buffer management scheme, is applied to RAM buffer inside SSDs.

BPLRU combines three key techniques, which are described in separate subsections: block-level LRU management, page padding, and LRU compensation.

3.1 Block-level LRU

BPLRU manages an LRU list in units of blocks. All RAM buffers are grouped in blocks that have the same size as the erasable block size in the NAND flash memory. When a logical sector cached in the RAM buffer is accessed, all sectors in the same block range are placed at the head of the LRU list. To make free space in the buffer, BPLRU chooses the least recent block instead of a sector, and flushes all sectors in the victim block. This block-level flushing minimizes the log block attaching cost in log-block FTL [17].

Figure 3 shows an example of the BPLRU list. In the figure, eight sectors are in the write buffer, and each block contains four sectors. When sector 15 is written again, the whole block is moved to the head of the LRU list. Thus sector 12 is at the front of the LRU list even though it has not been recently accessed. When free space is required, the least recently used block is selected as a victim block, and all sectors in the victim block are flushed from the cache at once. In the example, block 1 is selected as the victim block, and sectors 5 and 6 are flushed.

Table 1 shows a more detailed example. It assumes that only two log blocks are allowed, and eight sectors can reside in the write buffer. In this example, 14 highly scattered sectors are written in this order: 0, 4, 8, 12, 16, 1, 5, 9, 13, 17, 2, 6, 10, and 14. Because no dupli-

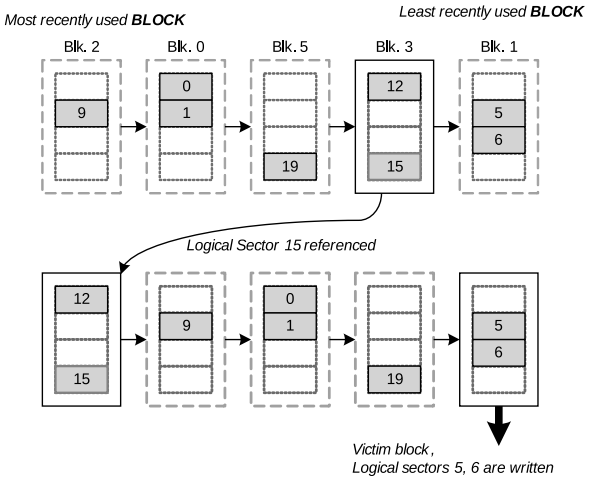


Figure 3: An example of Block-level LRU. When sector 15 is accessed, sector 12 is also moved to the front of the list because sector 12 is in the same block with sector 15.

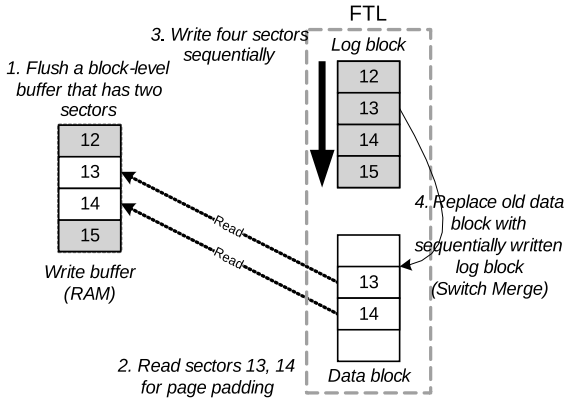


Figure 4: An example of page padding. When Block-level LRU chooses a victim block having sector 12 and sector 15, sector 13 and sector 14 are read from the storage, and four sectors are written back sequentially.

cated access exists in the sequence, the LRU cache does not influence the write pattern, and these writes evoke 12 merges in the FTL. If we apply block-level LRU for exactly the same write pattern, we can reduce the merge counts to seven.

If write requests are random, the cache hit ratio will be lower. Then, the LRU cache will act just like the FIFO queue and not influence the performance. BPLRU, however, can improve write performance even if the cache is not hit at all. If we calculate the cache hit rate for the block level, it may be greater than zero even though the sector level cache hit rate is zero. BPLRU minimizes merge cost by reordering the write sequences in the erasable block unit of NAND flash memory.

3.2 Page Padding

Our previous research [16] proposed the page padding technique to optimize the log-block FTL algorithm. In the method, we pad the log block with some clean pages from the data block to reduce the number of *full merges*. Here we apply the same concept for our buffer management scheme.

BPLRU uses a page padding technique for a victim block to minimize the buffer flushing cost. In log-block FTL, all sector writes must be performed to log blocks, and the log blocks are merged with data blocks later. When a log block is written sequentially from the first sector to the last sector, it can simply replace the associated data block with a *switch merge* operation. If a victim block flushed by BPLRU is full, then a *switch merge* will occur in the log-block FTL. Otherwise, a relatively expensive *full merge* will occur instead of a *switch merge*. Therefore, BPLRU reads some pages that are not in a victim block, and writes all sectors in the block range sequentially. Page padding may seem to perform unnecessary reads and writes, but it is more effective because it can change an expensive *full merge* to an efficient *switch merge*.

Figure 4 shows an example of page padding. In the example, the victim block has only two sectors (12 and 15), and BPLRU reads sectors 13 and 14 for page padding. Then, four sectors from sectors 12-16 are written sequentially. In log-block FTL, a log block is allocated for the writes, and the log block replaces the existing data block because the log block is written sequentially for all sectors in the block, i.e., a *switch merge* occurs. We show the effectiveness of page padding separately in Section 4.

3.3 LRU Compensation

Because LRU policy is not as effective for sequential writes, some enhanced replacement algorithms such as low inter-reference recency set (LIRS) [9] and adaptive replacement cache (ARC) [19] have been proposed.

To compensate for a sequential write pattern, we used a simple technique in BPLRU. If we know that the most recently accessed block was written sequentially, we estimate that that block has the least possibility of being rewritten in the near future, and we move that block to the tail of the LRU list. This scheme is also important when page padding is used.

Figure 5 shows an example of LRU compensation. BPLRU recognizes that block 2 is written fully sequentially, and moves it to the tail of the LRU list. When more space is needed later, the block will be chosen as a victim block and flushed. We show the effectiveness of LRU

Sector Writes	LRU		Block-level LRU	
	Cache status (8 sectors)	Log block (2 blocks)	Cache status (8 sectors)	Log block (2 blocks)
0, 4, 8, 12, 16	**16, 12, 8, 4, 0**		**[16], [12], [8], [4], [0]**	
1, 5, 9	**9, 5, 1**, 16, 12, 8, 4, 0		**[8, 9], [4, 5], [0, 1]**, [12], [16]	
13	**13**, 9, 5, 1, 16, 12, 8, 4 → 0	[0]	**[12, 13]**, [8, 9], [4, 5], [0, 1] → 16	[16]
17	**17**, 13, 9, 5, 1, 16, 12, 8 → 4	[4], [0]	**[17]**, [12, 13], [8, 9], [4, 5], → 0, 1	[0, 1], [16]
2	**2**, 17, 13, 9, 5, 1, 16, 12 → 8	[8], [4] → M[0]	**[2]**, [17], [12, 13], [8, 9], [4, 5]	
6	**6**, 2, 17, 13, 9, 5, 1, 16 → 12	[12], [8] → M[4]	**[6]**, [2], [17], [12, 13], [8, 9] → 4, 5	[4,5], [0, 1] → M[16]
10	**10**, 6, 2, 17, 13, 9, 5, 1 → 16	[16], [12] → M[8]	[8, 9, **10]**, [6], [2], [17], [12, 13]	
14	**14**, 10, 6, 2, 17, 13, 9, 5 → 1	[1] , [16]→ M[12]	**[14]**, [8, 9, 10], [6], [2], [17] → 12, 13	[12, 13], [4, 5]→ M[0, 1]
Buffer flushing by LRU order	14, 10, 6, 2, 17, 13, 9 → 5	[5] , [1]→ M[16]	[14], [8, 9, 10], [6], [2] → 17	[17], [12, 13]→ M[4, 5]
	14, 10, 6, 2, 17, 13 → 9	[9], [5] → M[1]	[14], [8, 9, 10], [6] → 2	[2], [17]→ M[12, 13]
	14, 10, 6, 2, 17 → 13	[13], [9]→ M[5]	[14], [8, 9, 10] → 6	[6], [2]→ M[17]
	14, 10, 6, 2 → 17	[17], [13]→ M[9]	[14] → 8, 9, 10	[8, 9, 10], [6]→ M[2]
	14, 10, 6 → 2	[2], [17]→ M[13]	→ 14	[14], [8, 9, 10]→ M[6]
	14, 10 → 6	[6], [2]→ M[17]		
	14 → 10	[10], [6]→ M[2]		
	→ 14	[14], [10]→ M[6]		
	Total merge count	**12**		**7**

Note: [], →, M mean a block boundary, flushing, merge operation in FTL, respectively

Table 1: Comparison of LRU and Block-level LRU. All sectors in the example write sequence are written only once. Thus, LRU cache acts just like FIFO queue, and does not influence write performance. Meanwhile, Block-level LRU can reduce the number of merges in FTL by reordering the write sequences in the erasable block unit of NAND flash memory.

compensation in Section 4 with a sequential workload.

3.4 Implementation Details

Since BPLRU is for the RAM buffer inside an SSD, we must design BPLRU to use as little CPU power as possible. The most important part of LRU implementation is to find the associated buffer entry quickly because searching is required for every read and write request.

For this purpose, we used a two-level indexing technique. Figure 6 illustrates the data structure of BPLRU. There are two kinds of nodes, a sector node and a block header node, but we designed our system to share only one data structure to simplify memory allocation and free processes. Free nodes are managed by one free node list, and a free node can be used for a block header node or a sector node.

The members of the node structure are defined as follows: Two link points for LRU or a free node list (`nPrev`, `nNext`), block number (`nLbn`), number of sectors in a block (`nNumOfSct`), and sector buffer (`aBuffer`). When a node is used as a sector node, `aBuffer[]` contains the contents of the writing sector, while it is functioning as a secondary index table that points to its child sector nodes when the node is used as a block header node. That is, every block header node has its second-level index table.

With this two-level indexing technique, we can find the target sector node using just two index references. The memory for the block header nodes should be regarded as the cost for the fast searching. To find a sector node with a sector number, first we calculate the block number by dividing the sector number by the number of sectors per block, N. Then, the first-level block index table is referenced with the calculated block number. If no associated block header exists, the sector is not in the write buffer. If a block header node exists, then we check the secondary index table in the block header node with the residue of dividing the sector number by N.

4 Evaluation

To evaluate BPLRU, we used both simulation and experiments on a real hardware prototype to compare four cases: no RAM buffer, LRU policy, FAB policy, and the BPLRU method.

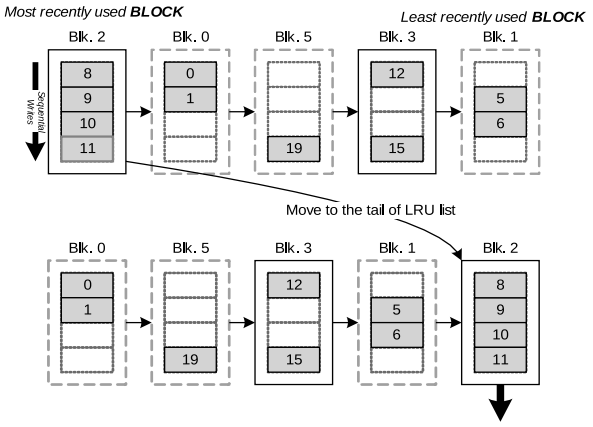


Figure 5: An example of LRU compensation. When sector 11 is written, BPLRU recognizes that block 2 is written fully sequentially, and moves it to the tail of the LRU list.

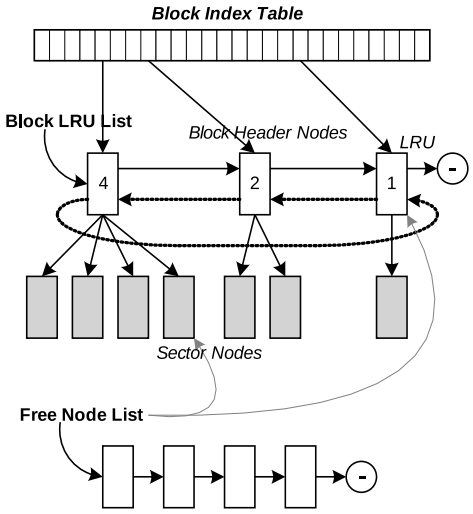


Figure 6: Data structure of BPLRU. For fast buffer lookup, two-level indexing is used. The first-level table is indexed by block numbers. Each entry in the table points to the block header if there are some buffers belonging to the block. Otherwise it has a null value. Each block header has second-level index table for all sectors in the block.

4.1 Collecting Write Traces

For our experiments, we collected write traces from three different file systems: NTFS, FAT16, and EXT3. Since we were interested in the random write performance, we excluded read requests from the traces. We attached a new hard disk to the test computer, and used a 1-GB partition for collecting write traces; our prototype hardware had 1-GB capacity limitation.

We used *Blktrace* [1] on Linux and *Diskmon* [24] on Windows XP to collect the traces. For Windows Vista, we used *TraceView* [20] in the Windows Driver Development Kit to collect write traces including buffer flush command.

We used nine different work tasks for our experiments.

W1: MS Office 2003 Installation. We captured the write traces while installing MS Office 2003 with the full options on Windows XP using NTFS. However, because we used a separate partition for the installation, some writes for Windows DLL files were missed in our traces.

W2: Temporary Internet Files. We changed the position of the temporary Internet files folder to a 1-GB partition formatted with NTFS. We then collected traces while surfing the Web with Internet Explorer 6.0 until we filled the folder with 100 MB of data spread across several small files.

W3: PCMark05. *PCMark05* [5] is one of the most widely used benchmark programs for Windows. It consists of several tests, and subsets of the test can be performed separately. We performed the HDD test and traced the write requests on NTFS in Windows XP.

W4: Iometer. We used *Iometer* [7] on NTFS to produce pure uniformly distributed random write accesses. *Iometer* creates a large file with the size of the complete partition, and then overwrites sectors randomly.

W5: Copying MP3 Files. In this task, we copied 90 MP3 files to the target partition and captured the write traces. This task was chosen to examine the effect of the write buffer when writing large files. Even though the goal of BPLRU is to enhance random write performance, it should not be harmful for sequential write requests because sequential patterns are also very important. The average file size was 4.8 MB. We used the FAT16 file system since it is the most popular in mobile multimedia devices such as MP3 players and personal media players.

W6: P2P File Download. We used *emule* [11] to capture write traces of a peer-to-peer file download program. Since emule permits configuring the positions of temporary and download folders, we changed the temporary folder position to the target partition. We then downloaded a 634-MB movie file and captured the write traces. We used the FAT16 file system, and this pattern can be considered as the worst case. Small parts of a file are written almost randomly to the storage because the peer-to-peer program downloads different parts concurrently from numerous peers.

Operation	Time(us)
256-KB Block Erase	1500
2KB-Page Read	50
2KB-Page Write	800
2KB-Data Transfer	50

Table 2: Timing parameters for simulation. These are typical timing values that are shown in the datasheet of MLC NAND flash memory.

W7: Untar Linux Sources. We downloaded *linux-2.6.21.tar.gz* from the Internet and extracted the source files to the target partition, which was formatted with the EXT3 file system. After the extraction, we had 1,285 folders and 21,594 files, and the average file size was 10.7 KB.

W8: Linux Kernel Compile. With the sources of *linux-2.6.21*, we compiled the Linux kernel with the default configuration and collected write traces.

W9: Postmark. We chose *Postmark* because it is widely used for evaluating the performance of I/O subsystems. The benchmark imitates the behavior of an Internet mail server and consists of a series of transactions. In each transaction, one of four operations (file creation, deletion, read, or write) is executed at random. We collected the write traces from all three file systems. Because *Postmark* can be compiled and run under Windows as well as Linux, we used it to compare the performance of BPLRU among the different file systems.

4.2 Simulation

4.2.1 Environment

For our simulation, we assumed the log-block FTL algorithm and a 1-GB MLC NAND flash memory with 128 2-KB pages in each block. The log-block FTL was configured to use seven log blocks. We ignored the map block update cost in the FTL implementation because it was trivial. We simulated the write throughput and the required number of block erase operations while varying the RAM buffer size from 1 to 16 MB.

We used the parameters of Table 2 for the performance calculation. Note that these values are typical of those provided by MLC NAND flash memory data sheets [25]. Some differences may exist in real timing values.

4.2.2 Performance and Erase Count

The simulation results for W1 through W8 are shown in Figure 7 through Figure 14, respectively.

For the *MS Office Installation* task (W1), BPLRU exhibited a 43% faster throughput and 41% lower erase count than FAB for a 16-MB buffer.

For the W2, the BPLRU performance was slightly worse than FAB for buffers less than 8 MB. However, the performance improved for buffer sizes larger than 8 MB, and the number of erase operations for BPLRU was always less than for FAB. The result of *PCMark05* (W3) test (Figure 9) was very similar to the *Temporary Internet Files* (W2) test.

The result of the *Iometer* test (W4) was different from all the others. FAB showed better write performance than BPLRU, and the gap increased for bigger buffer cache sizes. This is because of the difference in the victim selection policy between FAB and BPLRU. BPLRU uses the LRU policy, but no locality exists in the write pattern of Iometer because it generates pure random patterns. Therefore, considering the reference sequence is a poorer approach than simply choosing a victim block with more utilized sectors like FAB. However, due to page-padding, BPLRU exhibits lower erase counts.

Figure 11 shows the effect of BPLRU for a sequential write pattern (W5). While the result of the peer-to-peer task (W6)(figure 12) illustrates the generally poor performance of flash storage for random writes, it does show that BPLRU can improve the performance of random write requests significantly. We can see that FAB requires more RAM than BPLRU to get better performance.

The two results with the EXT3 file system (Figures 13 and 14) demonstrate the benefits of BPLRU. It provides about 39% better throughput than FAB in the Linux source untar case, and 23% in the Linux kernel compile task with a 16-MB buffer.

4.2.3 File System Comparison

To show the effects of BPLRU on different file systems, we used the Postmark benchmark for all three. We can see in Figure 15 that BPLRU exhibits fairly good throughput for all three file systems.

4.2.4 Buffer Flushing Effect

In our simulation, we assumed that no SCSI or SATA buffer flush command was ever sent to the device. However, in practice, file systems use it to ensure data integrity. Naturally, the inclusion of the flush command will reduce the effect of write buffering and degrade BPLRU performance.

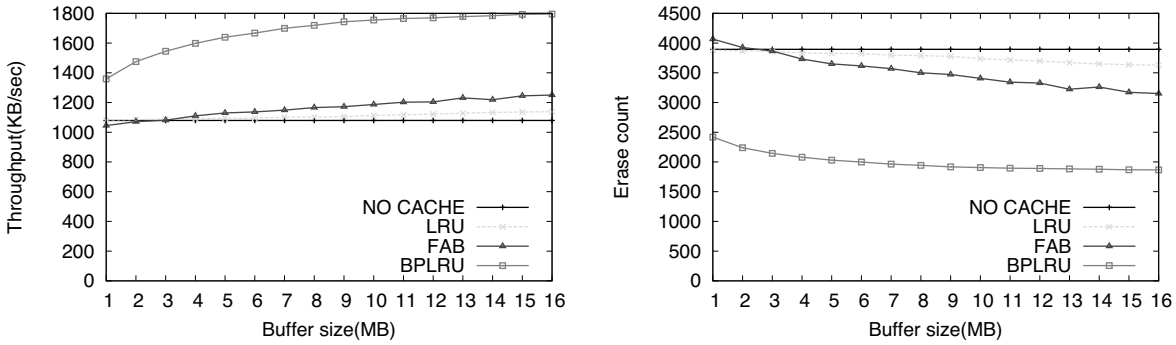


Figure 7: W1 MS Office 2003 installation (NTFS).

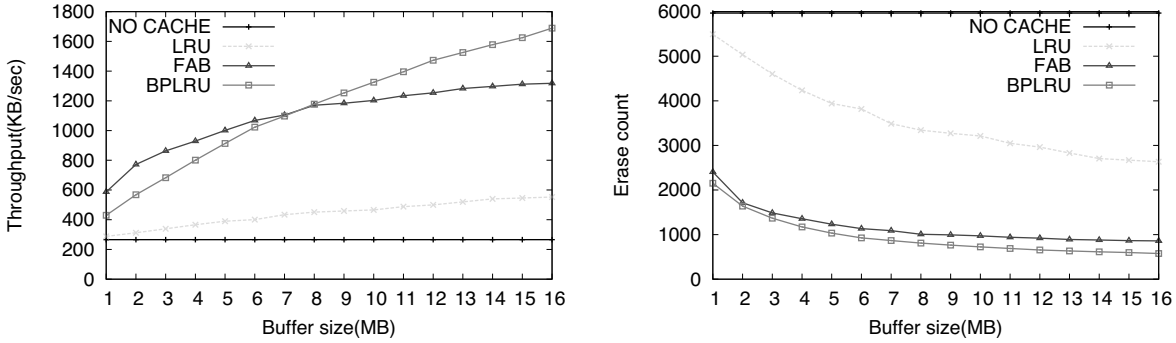


Figure 8: W2 Temporary internet files of Internet Explorer (NTFS).

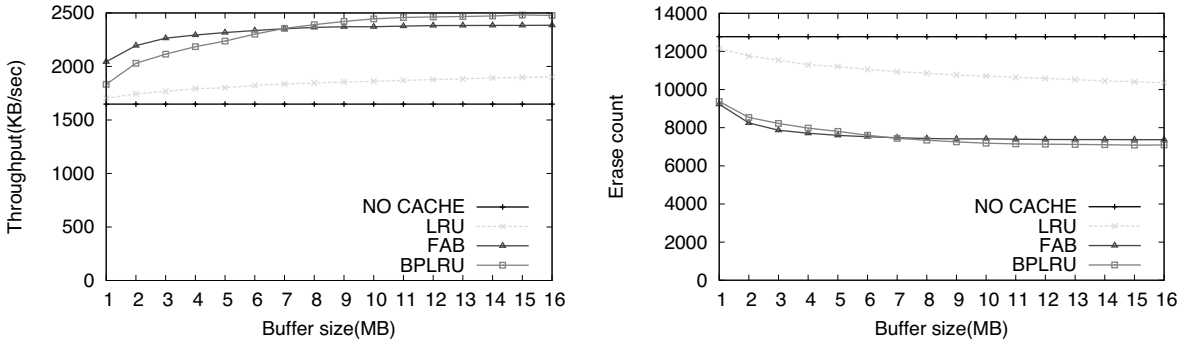


Figure 9: W3 HDD test of PCMark05 (NTFS).

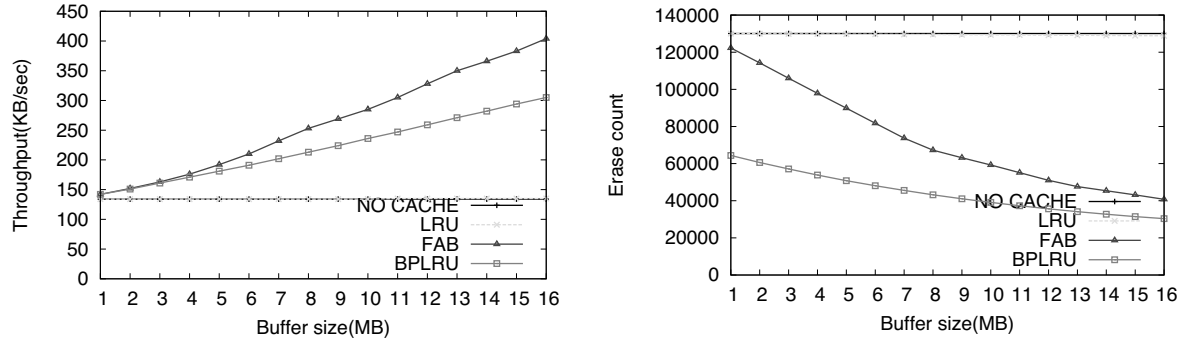


Figure 10: W4 Random writes by Iometer (NTFS).

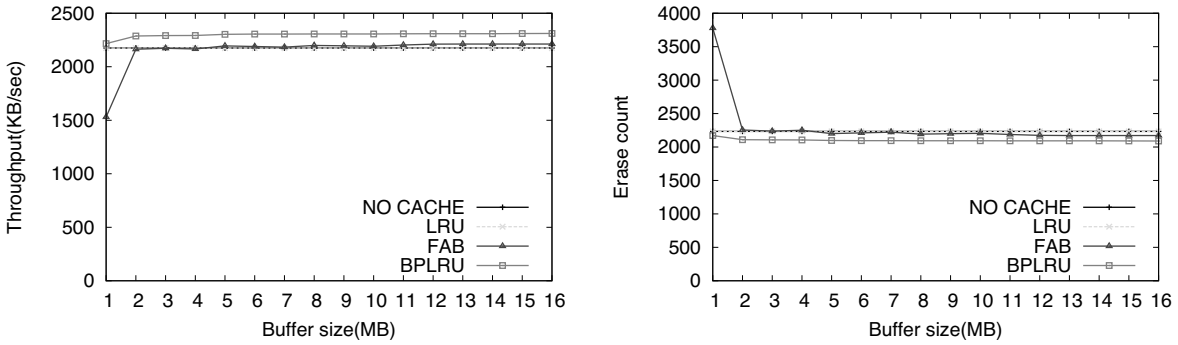


Figure 11: W5 Copy MP3 files (FAT16).

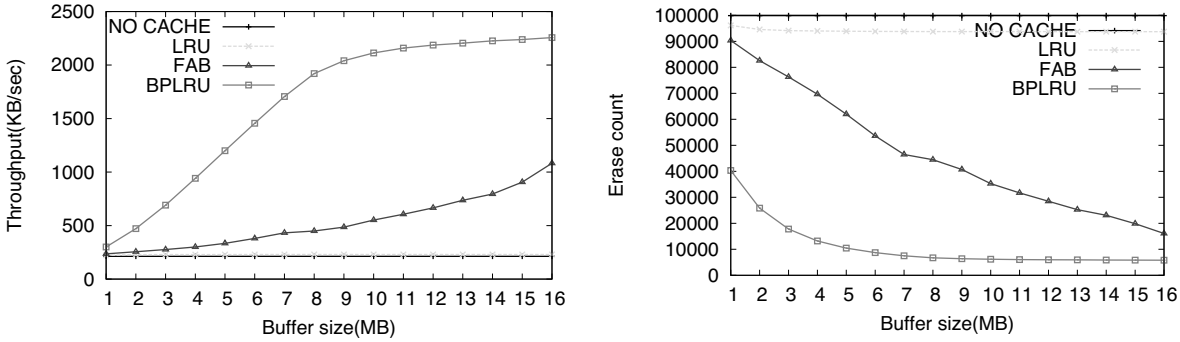


Figure 12: W6 634-MB file download by P2P program, emule (FAT16).

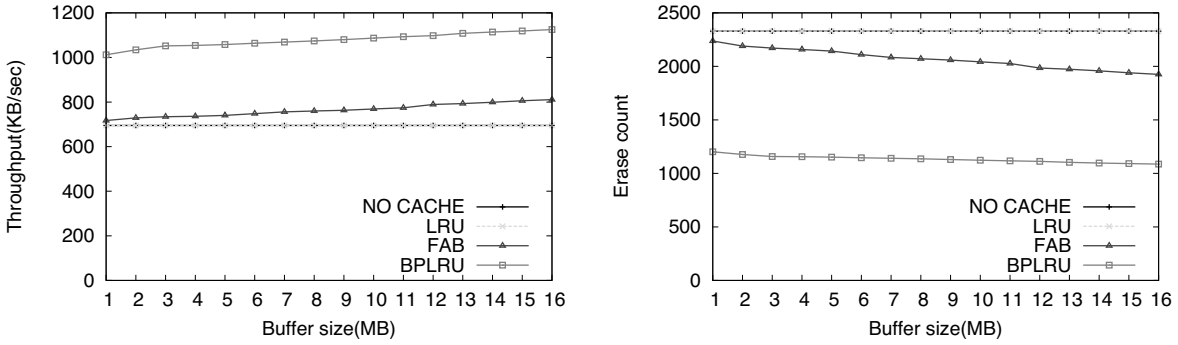


Figure 13: W7 Untar linux source files from linux-2.6.21.tar.gz (EXT3).

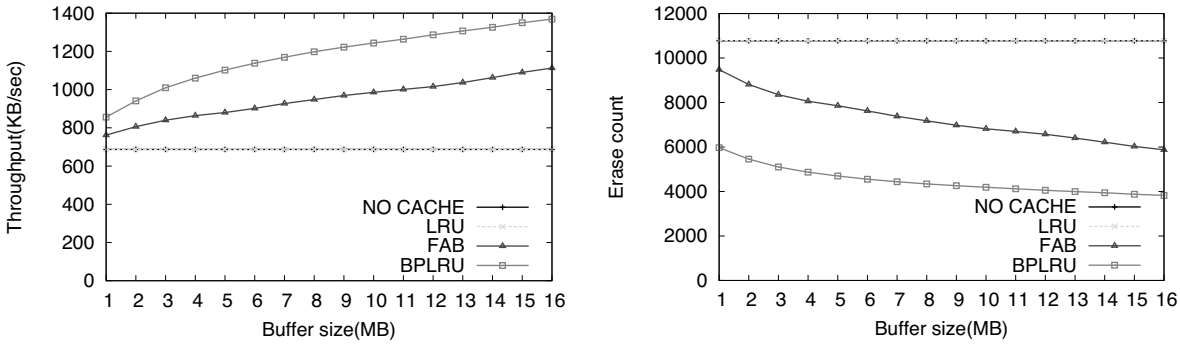
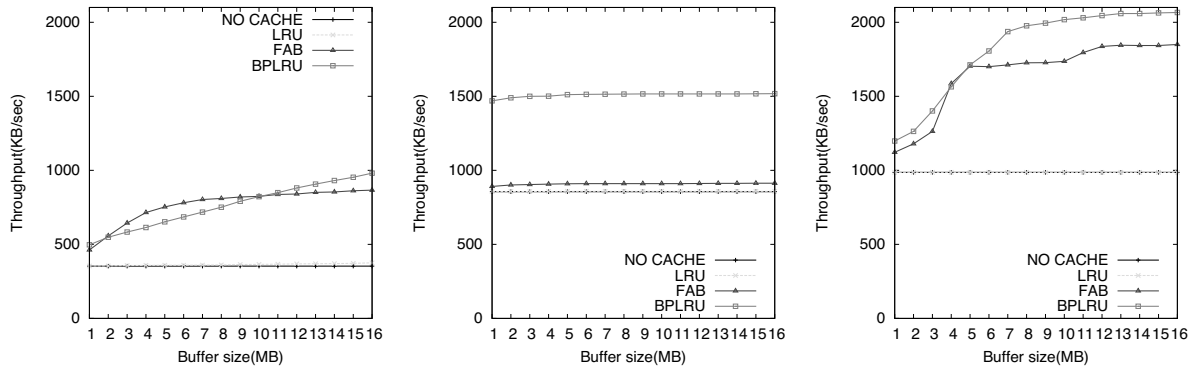


Figure 14: W8 Kernel compile with linux-2.6.21 sources (EXT3)



(a) NTFS (b) FAT16 (c) EXT3

Figure 15: Results of Postmark (W9) on three different file systems. BPLRU shows fairly good performance compared with LRU and FAB. Especially BPLRU shows the biggest performance gain for FAT16.

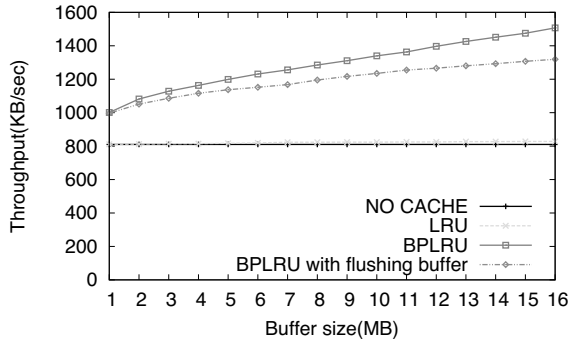


Figure 16: Buffer flushing effect. The throughput of BPLRU is reduced by about 23

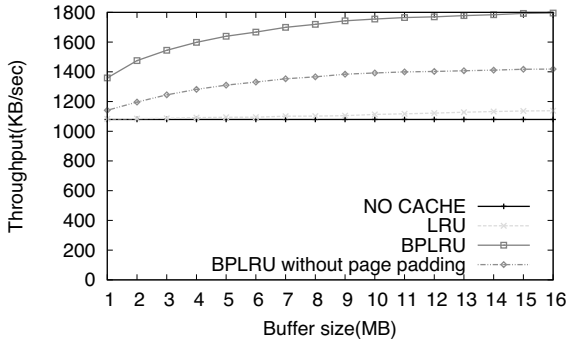


Figure 17: Page padding effect. The throughput for MS Office 2003 installation is reduced by about 21

To collect write traces including buffer flush command, we had to use Windows Vista since the *TraceView* utility does not trace this command for Windows XP. However, we discovered that for some unknown reason, Windows does not send the buffer flush command to a secondary hard disk.

Because of this, we traced the flush command on the primary C: drive to determine approximately how much it decreases the performance. As the simulation results in Figure 16 show, buffer flushing with a 16-MB buffer reduces the throughput by approximately 23%.

4.2.5 Detailed BPLRU Analysis

We also tested the benefits of page padding and LRU compensation on a subset of the experiments previously described. Figure 17 shows the effect of page padding with the simulation results for the write traces from the MS Office 2003 installation, and Figure 18 shows the effect of LRU compensation for the traces from copying MP3 files on the FAT file system. Page padding and LRU compensation enhance the throughput by about 26% (16 MB RAM) and 10% (1 MB RAM), respectively.

4.3 Real Hardware Prototype

We also implemented and tested three algorithms on a real prototype based on a target board with an ARM940T processor (Figure 19). It has 32 MB of SDRAM, a USB 2.0 interface, two NAND sockets, and an error correction code hardware engine for MLC and SLC NAND flash memory. We used a 200-MHz clock and a 1-GB MLC NAND flash memory whose page size was 2 KB. Since 128 pages were in each block, the block size was 256 KB.

We used our prototype to implement a USB mass storage (UMS) disk. We used the log-block FTL algorithm because of its wide commercial use. We configured it for seven log blocks and used an 8-MB RAM cache for three algorithms: LRU, FAB, and BPLRU. We used a 2.4-GHz Pentium 4 PC with Windows XP Professional, 512 MB

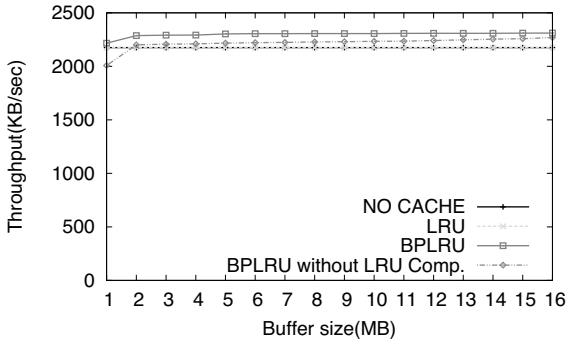


Figure 18: LRU compensation effect. The throughput of BPLRU for copying MP3 files on FAT16 is reduced by approximately 9

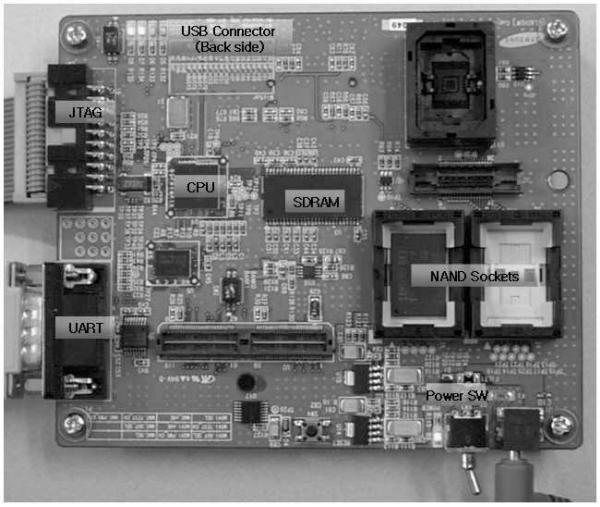


Figure 19: Prototype flash storage board. We implemented USB mass storage device with 1-GB MLC NAND flash memory and 8-MB RAM buffer.

of RAM, and a USB 2.0 interface as a host machine.

We designed a simple application to replay write traces to the UMS disk. After attaching our prototype UMS disk to the host computer, the application replayed the write traces and measured the time automatically. When all traces were written to the UMS disk, our application sent a special signal to the UMS disk to flush its RAM buffer to FTL and report the total elapsed time in seconds including the RAM flushing time. During the test, the trace replay application wrote sectors by directly accessing the Windows device driver, bypassing the Windows file system and buffer cache. Therefore the experimental results were not affected by the file system and the buffer cache of the host.

Figure 20 compares the simulation and the prototype experimental results for six tasks. Some slight differences are evident. For the MP3 copying task, the prototype results show that using the RAM buffer is worse than not using it. Also, the performance of BPLRU in the simulation is slightly better than in our prototype system. The reason for this is that we did not include the RAM copy cost in our simulation, but simply derived the performance from the operation counts of page reads, page writes, and block erasures. In addition, the operation time of NAND flash memory was not constant in our real-target experiments. For example, we used 1.5 ms for the block erase time because this value is given by data sheets as a typical time. However, the actual time may depend on the status of the block on the actual board. Even so, the overall trends were very similar between the prototype experiment and the simulation results.

5 Conclusions

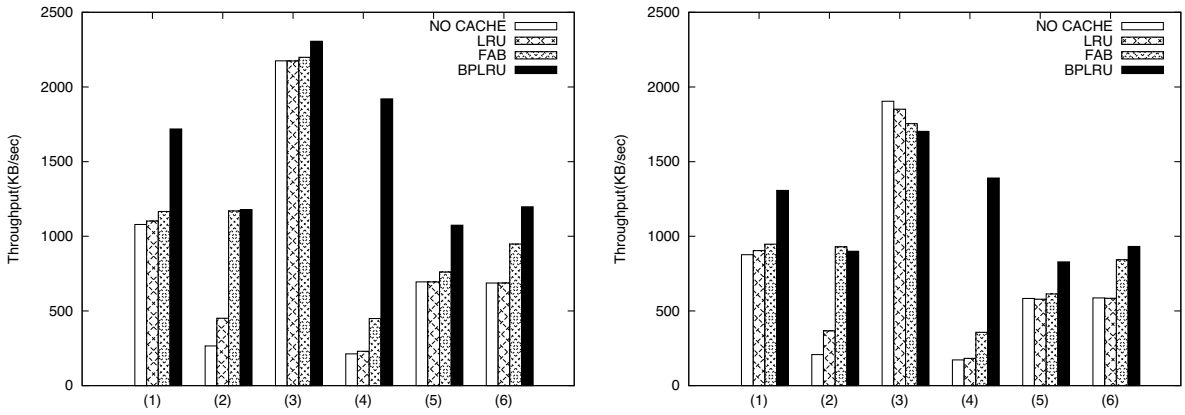
We demonstrated that a certain amount of write buffer in a flash storage system can greatly enhance random write performance. Even though our proposed BPLRU buffer management scheme is more effective than two previous methods, LRU and FAB, two important issues still remain. First, when a RAM buffer is used in flash storage, the integrity of the file system may be damaged by sudden power failures. Second, frequent buffer flush commands from the host computer will decrease the benefit of the write buffer using BPLRU.

Dealing with power failures is an important operational consideration for desktop and server systems. We are considering different hardware approaches to address this issue as an extension of our current work. A small battery or capacitor may delay shutdown until the RAM content is safely saved to an area of flash memory reserved for the purpose; an 8-MB buffer could be copied to that space in less than 1 s. Also, we may use nonvolatile magnetoresistive RAM or ferroelectric RAM instead of volatile RAM.

Our study focused on a write buffer in the storage device. However, the BPLRU concept could be extended to the host side buffer cache policy. We may need to consider read requests with a much bigger RAM capacity. At that point, an asymmetrically weighted buffer management policy will be required for read and write buffers, such as CFLRU or LIRS-WSR [12]. This is also a subject for future research.

6 Acknowledgments

We thank Dongjun Shin, Jeongeun Kim, Sun-Mi Yoo, Junghwan Kim, and Kyoungil Bahng for their frequent assistance and valuable advice. We also like to express our deep appreciation to our shepherd, Jiri Schindler, for his helpful comments on how to improve the paper.



(a) Simulation Results

(b) Prototype Experimental Results

Figure 20: Results comparison between simulation and prototype results for six workloads. (1) MS Office Installation (NTFS), (2) Temporary Internet Files (NTFS), (3) Copying MP3 Files (FAT16), (4) P2P File Download (FAT16), (5) Untar Linux Sources (EXT3), and (6) Linux Kernel Compile (EXT3)

References

[1] Jens Axboe. Block IO Tracing. `http://www.kernel.org/git/?p=linux/kernel/git/axboe/blktrace.git;a=blob;f=README`.

[2] Amir Ban. Flash File System, 1993. United States Patent, No 5,404,485.

[3] Fred Douglis, Ramon Caceres, M. Frans Kaashoek, Kai Li, Brian Marsh, and Joshua A. Tauber. Storage alternatives for mobile computers. In *Operating Systems Design and Implementation*, pages 25–37, 1994.

[4] Douglas Dumitru. Understanding Flash SSD Performance. Draft, `http://www.storagesearch.com/easyco-flashperformance-art.pdf`, 2007.

[5] Futuremark Corporation. PCMARK'05. `http://www.futuremark.com/products/pcmark05/`.

[6] Intel Corporation. Understanding the Flash Translation Layer (FTL) Specification. White Paper, `http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf`, 1998.

[7] Iometer Project, iometer-[user—devel]@lists.sourceforge.net. Iometer Users Guide. `http://www.iometer.org`.

[8] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *FAST'05: Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[9] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 31–42, New York, NY, USA, 2002. ACM.

[10] Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee. FAB: flash-aware buffer management policy for portable media players. *Consumer Electronics, IEEE Transactions on*, 52(2):485–493, 2006.

[11] John, Hendrik Breitkreuz, Monk, and Bjoern. eMule. `http://sourceforge.net/projects/emule`.

[12] Hoyoung Jung, Kyunghoon Yoon, Hyoki Shim, Sungmin Park, Sooyong Kang, and Jaehyuk Cha. LIRS-WSR: Integration of LIRS and writes sequence reordering for flash memory. *Lecture Notes in Computer Science*, 4705:224–237, 2007.

[13] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A superblock-based flash translation layer for NAND flash memory. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, pages 161–170, New York, NY, USA, 2006. ACM.

[14] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[15] Bumsoo Kim and Guiyoung Lee. Method of driving remapping in flash memory and flash memory architecture suitable therefore, 2002. United States Patent, No 6,381,176.

[16] Hyojun Kim, Jin-Hyuk Kim, ShinHo Choi, HyunRyong Jung, and JaeGyu Jung. A page padding method for fragmented flash storage. *Lecture Notes in Computer Science*, 4705:164–177, 2007.

[17] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[18] M-Systems. Two Technologies Compared: NOR vs. NAND. White Paper, `http://www.dataio.com/pdf/NAND/MSystems/MSystems_NOR_vs_NAND.pdf`, 2003.

[19] Nimrod Megiddo and Dharmendra S. Modha. ARC: a self-tuning, low overhead replacement cache. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[20] Microsoft. Windows Driver Kit: Driver Development Tools, TraceView. `http://msdn2.microsoft.com/en-us/library/ms797981.aspx`.

[21] Chanik Park, Jeong-Uk Kang, Seon-Yeong Park, and Jin-Soo Kim. Energy-aware demand paging on NAND flash-based embedded storages. In *ISLPED '04: Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, pages 338–343, New York, NY, USA, 2004. ACM.

[22] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 234–241, New York, NY, USA, 2006. ACM.

[23] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[24] Mark Russinovich. DiskMon for Windows v2.01. `http://www.microsoft.com/technet/sysinternals/utilities/diskmon.mspx`, 2006.

[25] Samsung Electronics. K9XXG08UXM 1G x8 Bit / 2G x 8 Bit NAND Flash Memory. `http://www.samsung.com/global/business/semiconductor/products/flash/Products_NANDFlash.html`, 2005.

[26] SSFDC Forum. SmartMedia Specification. `http://www.ssfdc.or.jp`.