FAB: Flash-Aware Buffer Management Policy for Portable Media Players

Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee

Abstract — *This paper presents a novel buffer management scheme for portable media players equipped with flash memory. Though flash memory has various advantages over magnetic disks such as small and lightweight form factor, solid-state reliability, low power consumption, and shock resistance, its physical characteristics imposes several limitations. Most notably, it takes relatively long time to write data in flash memory and the data cannot be overwritten before being erased first. Since an erase operation is performed as a unit of larger block, the employed strategy for mapping logical blocks onto physical pages affects real performance of flash memory. This article suggests a flash-aware buffer management scheme that reduces the number of erase operations by selecting a victim based on its page utilization rather than based on the traditional LRU policy. Our scheme effectively minimizes the number of write and erase operations in flash memory, reducing the total execution time by 17% compared to the LRU policy*[1].

Index Terms — flash memory, buffer, replacement

I. INTRODUCTION

Portable Media Player (PMP) is a device that stores or plays multimedia contents. We use the term rather loosely in this paper, encompassing a wide range of consumer electronics devices which share the similar characteristics, such as MP3 players, portable DVD players, digital cameras, PDAs, and even cellular phones. Generally, it consists of a processing unit, main memory, and storage for storing media data with its capacity ranging from several Mbytes to Gbytes. The medium for this storage has been hard disks, but due to its fragility and high power consumption they have been replaced with flash memory.

Though flash memory is still more expensive than magnetic disk yet, it has distinctive advantages such as small and lightweight form factor, solid-state reliability, low power consumption, and shock resistance [1]. These characteristics are especially well suited to the requirements of mobile PMPs. Therefore, many vendors adopt flash memory for the non-volatile storage and this trend will remain for the foreseeable future.

[1] This research was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031)

Heeseung Jo, Jeong-Uk Kang, Seon-Yeong Park, Jin-Soo Kim, and Joonwon Lee are with the Computer Science Division, Korea Advanced Institute of Science and Technology, Daejeon, Korea (e-mail: heesn@camars.kaist.ac.kr, ux@calab.kaist.ac.kr, parksy@camars.kaist.ac.kr, jinsoo@kaist.ac.kr, and joon@cs.kaist.ac.kr).

Contributed Paper
Manuscript received April 15, 2006

Modern operating system (OS) supports a buffer mechanism to enhance the performance that is otherwise limited by slow operations of the secondary storage. Even for embedded OS that does not support the buffer mechanism, a proprietary buffering scheme is usually employed for the device. Because it is relatively slow to read from the storage, OS copies the data from storage to the buffer area and serves the next read operations from the faster main memory.

The buffer mechanism is also helpful for PMPs. For example, an MP3 player has a 32Mbytes DRAM buffer [2] with several Gbytes of flash memory for the media storage. It reads or writes several whole MP3 files to the buffer at a stretch, because DRAM buffer accesses are much faster than flash memory especially in write operation. In case that storage medium is hard disk, it can also minimize the energy consumption by turning off hard disk. For digital cameras, buffering a picture in RAM prior to actual writing on flash memory significantly reduces response time that is one of the most important performance metric for digital cameras.

In this paper, we propose a novel Flash-Aware Buffer (FAB) management scheme for PMPs. FAB suppose a PMP which has a DRAM buffer and a NAND flash storage. Since FAB is well aware of the characteristics of NAND flash memory and the mechanism of flash controller, it assists the flash controller to increase the performance of flash memory. The basic idea of FAB is to select a victim carefully when some of data in the DRAM buffer need to be evicted to the NAND flash memory. Our FAB scheme effectively minimizes the number of write and erase operations to the flash memory, reducing the total execution time by 17% compared to the traditional scheme which relies on the LRU buffer management policy.

The rest of the paper is organized as follows. Section 2 presents the background and related work to understand FAB. Section 3 describes the overview of FAB scheme and Section 4 presents the implementation details and the data structures of FAB. Section 5 describes the experimental results. Finally we conclude in section 6.

II. BACKGROUND AND RELATED WORK

This section introduces the hardware characteristics and the internal mechanism of flash memory along with a software layer that provides the block device interface of flash memory, which motivate us to devise the proposed buffering scheme.

A. Flash Memory

Flash memory is divided into two types, i.e., NOR and NAND. NOR flash memory supports the byte unit I/O and

shows shorter read time and longer write time compared to NAND flash memory [3]. It is mainly used as storage for program codes since it can be accessed by a byte unit. For such a trait, the BIOS of a computer system is usually stored on NOR flash memory. On the other side, NAND flash memory supports the page unit I/O with slower read time and faster write time. NAND flash memory is mainly used for data storage and it is regarded as a replacement of hard disk [4].

There are three basic operations in NAND flash memory: read, write, and erase. The read operation fetches data from a target page, while the write operation writes data to a target page. The erase operation resets all values of a target block to 1. In flash memory, once a page is written, it should be erased before it is written again, and this limitation is usually called *erase-before-write*. The read and write operations are performed by a page unit (512 Bytes or 2 Kbytes) and the erase operation is performed by a block unit (16 Kbytes or 128 Kbytes). A block consists of a fixed number of pages and a page holds a fixed number of sectors. Sector is the basic unit that can be seen by OS or other software.

Although flash memory has various advantages over hard disk, it also has some limitations as storage. Among them, the followings are the important concerns in designing our buffering scheme.

- No in-place-update: The previous data should be erased first in order to overwrite another data in the same physical area. The worse problem is that the erase operation cannot be performed on the particular data selectively, but on the whole block containing the original data. Apparently, it is not efficient to perform costly erase operation on every data write and more sophisticated handling of write operation is required.
- Asymmetric operation latencies: For NAND flash memory, read time is faster about 8 times than write time. Because a write operation sometimes involves an erase operation, it may entail non-deterministic long delay. For this reason, it is important to reduce the number of write operations.
- Uneven wear-out: Each block can be erased only for a limited number of times, usually several hundreds of thousands times. Once the number is reached, the block cannot be used any more. Therefore it is necessary to distribute erase operations evenly over the whole blocks.

B. Flash Translation Layer

Since most operating systems expect a block device interface for the secondary storage even though flash memory does not support it, a thin software layer called FTL (Flash Translation Layer) is usually employed between OS and flash memory. The main role of FTL is to emulate the functionality of block device with flash memory, hiding the latency of erase operation as much as possible. FTL achieves this by redirecting each write request from OS to an empty location in flash memory that has been erased in advance, and by maintaining an internal mapping table to record the mapping information from the logical sector number to the physical location.

The mapping schemes of FTL are classified either into a page-level mapping scheme [6] or into a block-level mapping scheme [7]. Page-level mapping is a fine-grained translation from a logical sector number to a physical sector number. For this scheme, the size of the mapping table is relatively large, and thus it needs large SRAM. Block-level mapping schemes translate a logical sector number to a physical block number and offset. Since the mapping table for this scheme is to find a block number rather than a sector number, its size is smaller. The offset helps to find the wanted page within a block. In block-level mapping schemes, a set of consecutive sectors usually stored in a single block. Once a new page cannot be accommodated in the assigned block, the block is erased after copying valid pages into a clean block.

Many FTL schemes are proposed to reduce the number of block erase and valid page copying in block-level mapping. Kim et al. suggest an FTL scheme that uses some fixed number of blocks as logging blocks [5]. If there is a write request, this scheme writes the data on a log block and maps the physical location of the page to the page in the log block. In this way, it logs the changes of the data until the log block becomes full. When all the log blocks are used, then some log blocks are merged into a new block to prepare clean log blocks. This scheme may need a large number of log blocks since each block needs the corresponding log block when one of its pages is updated. When free log blocks are not sufficient, the merge operation called *garbage collection* will happen too frequently. Also, the utilization of the log block is low since even a single page update of a block necessitates a whole log block.

The Fully Associative Sector Translation (FAST) FTL scheme is suggested to overcome the limitations of the log block scheme [8]. In the FAST scheme, the overwrite requests of a block is spread on all of the log blocks. It means the merge operation is delayed much longer and the total number of erase operations becomes smaller than that of the previous log block scheme.

Both of the log block scheme and the FAST scheme support the *switch merge*. If a log block holds all pages of an old data block, it switches the log block with the old data block. The switch merge is an optimal case of garbage collection because there is no overhead incurred for copying valid pages.

C. Related Work

Many efforts have been made to address the aforementioned limitations of flash memory. Kawaguchi et al. [9] proposed a translation layer for flash file systems based on an analysis of cost-benefit for garbage collection. Chiang et al. [10] investigated a Dynamic dAta Clustering (DAC) method to cluster data during data update to reduce the number of block erase. On the other hand, Wu et al. [11] proposed a large non-volatile main memory storage system with write buffering in battery-backed SDRAM to hide write latency. Lee et al. [12] examined a NAND-type flash memory package with a smart buffer cache to raise the hit ratio. Recently, Park et al. [13]

proposed a replacement algorithm called Clean First LRU (CFLRU) to minimize the number of write requests from the virtual memory system. Our work is primarily differs with these previous work in that we focus on exploiting the buffer management policy by considering the physical characteristics of flash memory without any modification to FTL or underlying hardware.

III. FLASH-AWARE BUFFER MANAGEMENT

A. System Overview

Fig. 1 shows the overall block diagram of a typical PMP system. All requests issued from CPU pass through the DRAM buffer which serves both read and write requests. If the requested data is not in the buffer, it should be serviced by the NAND flash storage. We assume that the NAND flash storage has its own storage controller for running FTL. Since other parts in the diagram are irrelevant to our discussion, their details are elided for brevity.

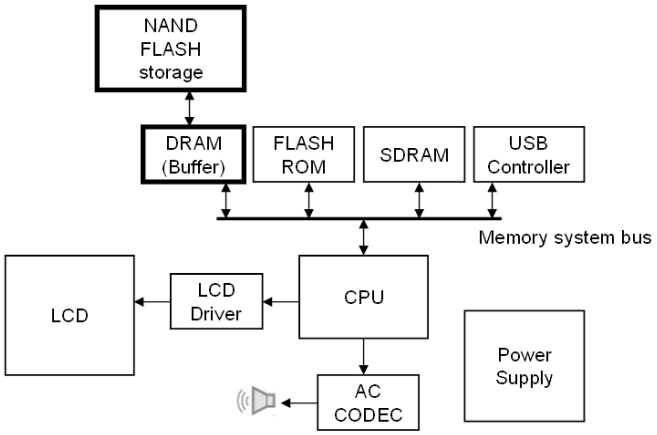


Fig. 1. The overall block diagram of PMP system

B. Flash-Aware Buffer Management Policy

The buffer is fully occupied with cached media data pretty soon after the boot time. To secure a buffer entry for a new data, a victim entry should be selected to be evicted from the buffer. The most common policy for selecting a victim is Least Recently Used (LRU) [14]. LRU is based on the theory that if a memory location is accessed recently, the location will be accessed again in the near future. Though LRU shows satisfactory performance for most memory systems, the characteristics of NAND flash memory prohibits it from being the best solution. For example, a typical file access pattern for PMP consists of a long sequential access for media data and several short accesses for metadata of the file. In this case, LRU policy can not hold the data for short accesses in the buffer because the long sequential access pushes them away from the buffer.

Our FAB policy is designed to achieve the following purposes. First, it should minimize the number of write and erase operations in flash memory since they are most prominent sources for the latencies. Second, the mechanism should help the storage controller to utilize the switch merge since it reduces the number of valid page copying during the merge operation. The third one is to maximize overwrite of hot pages since repetitive writes on hot pages cause a large number of log blocks on the flash storage resulting in frequent garbage collections. The last one is to minimize the search time to find the requested data in the buffer.

If the buffer is full, FAB selects a block as a victim and then flushes all of the pages that belong to the selected block. The victim block is a block that has the maximum number of pages in the buffer. FAB also employs a block-level LRU policy to avoid the problem of page-level LRU policy described in the previous example. At the same time, when a write of long contiguous data fills up the buffer, FAB selects these pages as victims to maximize the chance of switch merge in flash memory. This also increases the number of buffer hit of hot pages because the pages of short writes can be stayed longer in the buffer.

If FAB receives a write request, it searches the buffer first. On a hit, the data is overwritten in the buffer. Otherwise, a new slot is allocated in the buffer and the requested sector is written to the slot. Since all write requests are buffered, the actual data write on flash memory is occurred only when a block is replaced

```
FAB_Write (block, page, data)
{
    if ((bufloc = Search_Buffer (block, page)) != null) {
        Write_Page (bufloc, data);
    }
    else {
        if (Buffer_Full ()) {
            victim  = Select_Victim_Block ();
            Flush_Victim_Block (victim);
        }
        bufloc = Allocate_New_Page ();
        Write_Page (bufloc, data);
    }
    Rearrange_Blocklist_For_LRU (block);

}
```

Fig. 2. Handling a write request in FAB

In the case that the buffer is full, a victim page to be replaced should be selected, and the victim page is written back to the flash storage if the page is dirty. As mentioned before, our replacement scheme replaces a whole block rather than an individual page. A victim block is the one who has most pages in the buffer, and all the pages that belong to this victim block are evicted from the buffer. When several blocks tie, recency is considered as in the LRU policy. This policy reduces the number of page copies in the flash storage when blocks are merged. Since most PMP data are contiguous multimedia, most flash memory blocks are either full of valid pages or empty. However, in those blocks used by file

metadata, it is very probable that only several pages are updated frequently and some valid and invalid pages are mixed together. By choosing a full block as a victim, our policy favors metadata over media data, which is a very desirable feature for multimedia buffering. Fig. 2 shows how the write operation is handled in our scheme.

For a read request, FAB works similarly. If it is a hit in the buffer, FAB immediately returns the data to CPU. If FAB cannot find the sector in the buffer, it reads the data from flash storage to the buffer and returns the data. When there is not enough space for the requested data, FAB selects a victim block and flushes pages that belong to the block into flash storage. Fig. 3 shows an algorithm for this read operation.

```
FAB_Read (block, page, data)
{
    if ((bufloc = Search_Buffer (block, page)) != null) {
        Rearrange_Blocklist_For_LRU (block);
        Read_Page (bufloc, data);
    }
    else {
        if (Buffer_Full ()) {
            victim  = Select_Victim_Block ();
            Flush_Victim_Block (victim);
        }
        bufloc = Allocate_New_Page ();
        Read_From_Flash (bufloc, block, page);
        Read_Page (bufloc, data);
    }
    Rearrange_Blocklist_For_LRU (block);
}
```

Fig. 3. Handling a read request in FAB

Our FAB policy has three major advantages as follows.

- Reduced write and erase operations: Repetitive writes on the same page induce tremendous overhead for the flash storage since each version of write may generate a separate log block resulting in many sparse blocks necessitating frequent garbage collections. Our buffering scheme eliminates such a pathological case. Also, even though pages that belong to the same block are updated individually at different times, FAB allows flushing those pages at once when the block is replaced. This increases the chance of switch merge in flash storage, which otherwise would not be possible.
- Response time: Like other caching schemes, our buffering scheme reduces response times experienced by PMP users. Even when it is a miss, writing multimedia data on the buffer rather than on flash memory will reduce response time significantly. A smart data structure explained below also reduces the time taken for searching a page in the buffer.
- Hot page filtering: Hot pages are those that are written repetitively within a short time range. When such pages are scattered, they generate enormous number of log blocks on the flash storage. Clustering them onto a few blocks reduces garbage collection cost, but it needs bookkeeping to discern hot pages from cold pages. With our buffering scheme, hot pages are serviced in the buffer, and thus they do not impose any stress on the flash storage.

IV. The Implementation Details of FAB

A. Data Structures for FAB

To realize the buffering scheme explained in the previous section several data structures along with proper algorithms need to be devised. In our implementation, they are implemented with two issues in mind; time efficiency and space efficiency. The former issue is mainly related to the algorithm for searching the buffer while the later is about the memory space needed for storing tables and linking pointers.

FAB frequently searches the buffer because all the storage requests from CPU pass through the buffer. For each request, FAB has to decide whether the requested sector is in the buffer or not. To minimize the time to search the buffer, we contrived a data structure as shown in Fig. 4. The block node list is a linked list of blocks sorted by their recency. The page node list is a list of pages that belong to the corresponding block.

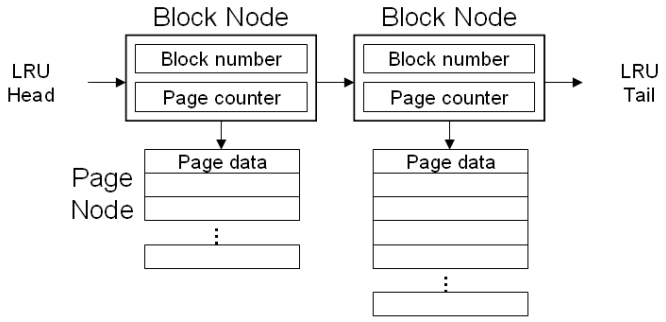


Fig. 4. The main data structures of FAB. Horizontally, the block nodes are linked as a linked list. Vertically, the page nodes are listed for each block node.

A block node has a block number, a page counter, a pointer for the next block node, and a pointer for its page node list. The block number field identifies a unique block in flash memory. The block node list pointer is for the horizontal linked list where all the block nodes are sorted by recency that is to be used for the block-level LRU policy. The page counter denotes the number of pages allocated in this block, and it is the primary metric to decide when selecting a victim. There is a pointer for each block node to point to the list of page nodes that belong to the block. Page node list has the page data associated with the page number. The size of a block node is 16 Bytes and the block node list is the only additional memory overhead of FAB. A data structure similar to the page node list is also required for the traditional page-level LRU policy.

B. Search Operation

A request from CPU consists of a sector number and the type of operation. The block that contains the wanted page(sector) can be identified by dividing the sector number by the block size. After the block number is identified, FAB searches the corresponding block node in the block node list.

Although it is implemented with a sequential search algorithm which has an *O(n)* complexity, the actual cost is not so high since the number of block nodes that can be in the linked list is small and limited by the buffer size. More complicated data structures such as hash tables and balanced trees could be adopted for faster search, but they make it difficult to implement the LRU mechanism without building an additional LRU list. Our experimental results in Section 5 confirm that this simple linear search does not cause any noticeable overhead.

If the wanted is not found in the block node list, FAB immediately terminates the search since it is a miss in the buffer. If the block is found, it continues to search the page node list. The page node list is structured as a search tree for efficient search operation. After finding the associated page node, FAB rearranges the LRU list by moving the block node to the head of the block node list.

C. Insert Operation

When the wanted page is not found in the buffer, a new page is allocated and attached to the page list of the corresponding block node. The page counter of this node is incremented by one. If the block node does not exist, a new block node should be added at the head of the block node list, and its page counter is set to one.

If the miss comes from a read request, the wanted page is retrieved from the flash storage and stored in the newly allocated page in the buffer. If it is from a write request, the data is written only in the new page, leaving the old page in the flash storage untouched. Since further requests to the page will be serviced in the buffer, the old copy of the page in the flash storage does not cause any problem.

D. Replace Operation

When the buffer is full and cannot accommodate a new page data, the buffer should evict a number of pages from the buffer. If the evicted page is dirty, i.e., modified while being in the buffer, it is written back to flash memory while unmodified pages are claimed immediately. Being a block-level LRU scheme, FAB selects a victim block, and flushes all the pages that belong to the block. FAB selects a victim block by the following rules in order.

1. The block which has the largest number of pages is selected. If a block has all of pages in the buffer, FAB will pick out the block first and it is an optimal selection.
2. If multiple blocks are selected by the above rule because all of them have the same number of pages in the buffer, then the block at the tail of the list is selected since it has not been accessed for the longest time.

To find a victim block, FAB starts searching from the tail of the block node list and if a block which is full of pages is found, it becomes the victim. Considering the characteristics of the files used for PMP devices, the block node list is supposed to contain many such block nodes. Fig. 5 shows a victim block selection algorithm.

```
#define BlockPerPageNum 64
Select_Victim_Block ()
{
   VictimBlock = -1;
   MaxPageNum = 0;
   For each BlockNode from BlockNodeListTail to
   BlockNodeListHead
   {
      if (BlockNode.PageCount == BlockPerPageNum)
          return BlockNode.BlockNum;
      if (BlockNode.PageCount > MaxPageNum)
      {
          MaxPageNum = BlockNode.PageCount;
          VictimBlock = BlockNode.BlockNum;
      }
   }
   return VictimBlock;
}
```

Fig. 5. Selecting a victim block in FAB

Our scheme replaces a whole block and most blocks are full of valid pages. Therefore, when a block is written back to flash memory, it just needs to switch the new block with old one if any, eliminating the need for copying valid pages from the old block. This switch merge is an optimal case because FTL can switch the log block with the old data block [5].

V. Experimental Results

A. Evaluation Methodology

We implemented our FAB policy on a simulator which mainly consists of NAND flash storage and a DRAM buffer. The simulator models known parameters related to current technologies as exactly as possible. The traces are extracted from disk access logs of real user activities on FAT32 file system. The workload is chosen to reflect representative PMP usage as shown in Table I.

Table I
Traces used for simulation

Trace	Description	The size of flash storage	The number of sectors written
Pic	The traces of digital camera. Picture files are about 1Mbytes.	512 MB	868,154
		1024 MB	1,803,561
		2048 MB	4,335,023
MP3	The traces of MP3 player. MP3 files are about 4-5Mbytes.	512 MB	1,146,234
		1024 MB	1,935,572
		2048 MB	5,602,743
Mov	The traces of movie player. Movie files are about 15-30Mbytes.	512 MB	1,067,905
		1024 MB	2,296,341
		2048 MB	7,196,759

The main performance metrics are the number of write and erase operations since they are the major factors limiting the performance of flash memory-based system. The sizes of flash memory experimented are 512 Mbytes, 1 Gbytes, and 2

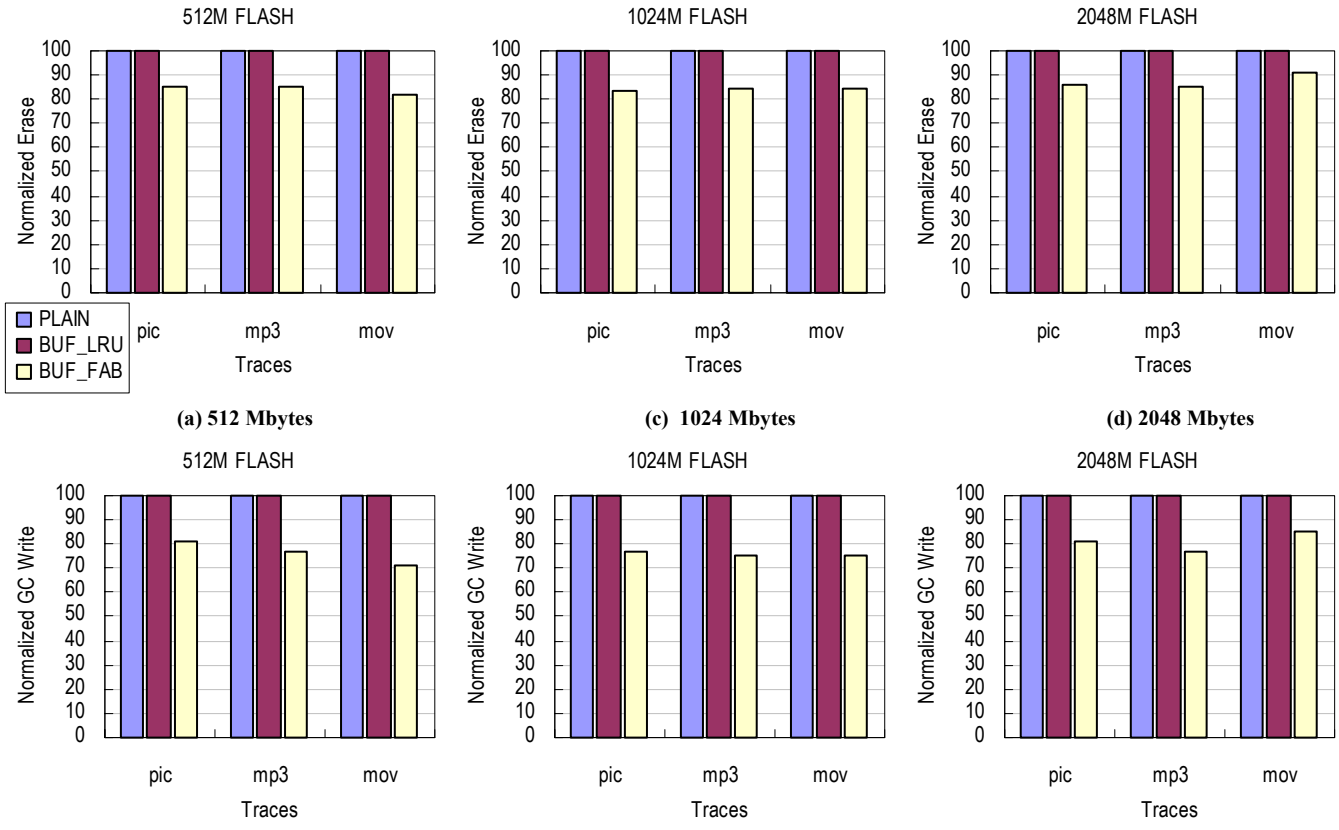


Fig. 6. The normalized number of write and erase operations of FAST, FAST+LRU buffer scheme, and FAST+FAB buffer scheme

Gbytes. The size of a sector is 512 Bytes and a page can hold 4 sectors. The number of pages that are accommodated in a block is 64 and, therefore, the size of a block is 128 Kbytes.

We adopted FAST as an FTL scheme of flash memory because its performance is known to be better than the log block scheme [8]. Though it appears that FAST would perform better for a larger number of log blocks, there exists a certain number beyond that the performance decreases since the overhead for managing log blocks increases and the benefit of additional log block gets diluted. Our experiments show that FAST performs best when the size of log blocks is 8 for our simulation conditions.

The access times for DRAM and NAND flash memory are summarized in Table II. These parameters are used to calculate the total execution time and are set to be the same as those used in the previous research for a fair comparison [8].

TABLE II
ACCESS TIMES OF DRAM AND NAND FLASH MEMORY

Operation	Access time
DRAM Buffer access time (ns/512 Bytes)	2,560
NAND Flash read time (ns/512 Bytes)	36,000,
NAND Flash write time (ns/512 Bytes)	266,000
NAND Flash erase time (ns/128 Kbytes)	2,000,000

B. Performance Evaluation

Fig. 6 a), b) are the results with a 512 Mbytes flash storage. There are 3 bars for each PMP type. The first bar (PLAIN) is for the FAST scheme without the intermediate buffer. The second bar (BUF_LRU) presents the FAST scheme with a buffer managed by a page-level LRU policy. The last bar (BUF_FAB) is for our FAB scheme. Fig. 6 a) shows normalized erase numbers for each scheme and Fig. 6 b) presents normalized write numbers needed for garbage collection. Because the total number of write count of pure data is same for all schemes, we only counted the number of write operations performed during garbage collection. Those writes are generated during garbage collection to copy valid pages to another clean block before the block is erased. The rest of graphs in Fig. 6 show the results for other sizes of flash storage, 1024 Mbytes and 2048 Mbytes. We can see that varying the size of the flash storage does not change the trend of overall performance gap.

All the results in Fig. 6 show that a pure LRU (BUF_LRU bars) scheme for the buffer management does not affect the performance of flash memory. This result was expected considering how multimedia files are accesses on PMPs. Most multimedia files are accessed sequentially and they are rarely accessed again within a short time interval. Buffering such a file is harmful rather than being a help since such large files would evict small files from the buffer even though most of them contain metadata that are

accessed repetitively.

FAB reduces the number of writes and erases by 29% and 19%, respectively, compared to the other schemes. This performance gains can be explained by the fact that evicting multimedia data first from the buffer helps retaining those metadata that are frequently accessed.

Fig. 7 shows the execution time breakdown taken for MP3 player trace with a 1024 Mbytes flash storage. GCREAD denotes the consumed flash read time during garbage collection including the time to identify whether the data is valid or invalid. GCWRITE presents the time taken to write valid pages during garbage collection. ERASE means the time to erase blocks and DATAWRITE is the time to write the requested data in flash memory.

In the case of our FAB scheme (BUF_FAB), since the number of garbage collection is smaller, the all the times for GCREAD, GCWRITE, and ERASE are smaller than others. While the overhead to search the buffer in BUF_FAB is measured only less than 0.1% of the total time, the corresponding overhead in BUF_LRU is about 33% of the total time, because the latter needs to search for a page in the longer page-level LRU list. Since the buffer access time heavily depends on the actual implementation of LRU algorithm, we have omitted the buffer access time in the results in Fig. 7 for a fair comparison. However, the fact that a significant amount of time is spent on searching the buffer in BUF_LRU suggests that the data structure and mechanism of FAB is simple, yet very efficient. Overall, BUF_FAB improves the execution time by 17% compared to BUF_LRU and PLAIN.

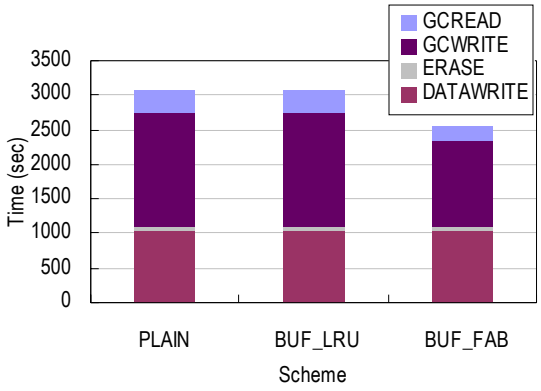


Fig. 7. The execution time breakdown for MP3 player trace with a 1024 Mbytes flash storage.

Fig. 8 presents the normalized number of buffer hits. The buffer hit means the case when the requested data is already in the buffer and serviced immediately from the buffer. As the file size manipulated by the application becomes larger from PIC to MP3 and MOV, the gaps between FAB and pure LRU get bigger and bigger. This result was expected and explained before in this section.

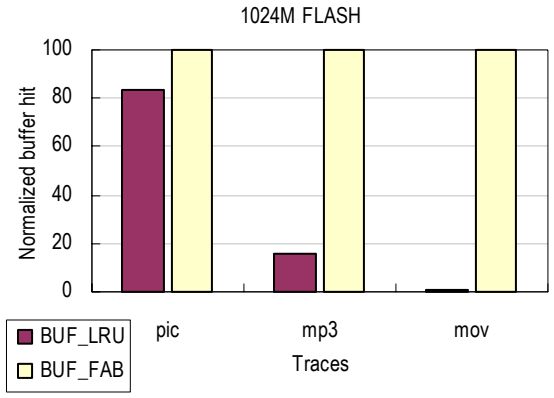


Fig. 8. The normalized number of buffer hits.

C. *The Effects of Buffer Size*

The results presented in Fig. 9 show the effects of buffer size. Because the DRAM buffer is an expensive resource for PMPs in terms of cost and power, it is necessary to make trade-off with performance. Fig. 9 shows the number of erase operations performed for the given buffer size during the simulation of digital camera trace. The buffer size is varied from 1 Mbytes to 32 Mbytes.

As the number of sectors written increases according to the flash storage size (cf. Table I), it is obvious that the larger flash storage size induces the larger number of erase operations. However, they all show a drop when the buffer size becomes 8 Mbytes, and this trend is similar for other traces as well. Actually, the optimal DRAM buffer size tends to be related to the working set size and other characteristics of the workload. However, we would like to note that, for evaluated traces and environments, the use of DRAM buffer whose capacity is less than 0.8% of the flash storage (e.g., 8 Mbytes DRAM buffer for 1 Gbytes flash storage) is quite effective in reducing the number of erase operations in flash memory.

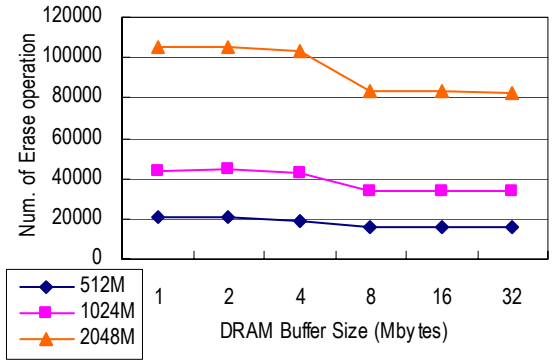


Fig. 9. The number of erase operations varying the DRAM buffer size.

D. *Memory Overhead*

We also evaluated the memory space overhead required for FAB. Table III shows the number of block nodes for various

DRAM buffer sizes and the corresponding memory size during the simulation of MP3 trace on 1 Gbytes flash storage. Since the number of block nodes changes according to the execution of the trace at run time, the number in Table III represents the largest number of block nodes stored on memory. Recall that the block node data structure which takes 16 Bytes of memory is the only memory overhead of FAB.

We can see that for small buffer size, the buffer is fragmented and occupied by many pages that belong to different blocks. This is because the buffer size is not big enough to hold all the blocks touched by the application. However, once the buffer size becomes larger than 4 Mbytes, the maximum number of block nodes is significantly reduced, although it slightly fluctuates according to the buffer size. The results for other traces were also similar. As shown in Table III, the memory overhead is very trivial compared to the general DRAM size used by PMPs.

TABLE III
MEMORY OVERHEAD FOR FAB (MP3 TRACE ON 1 GBYTES FLASH)

Buffer size (Mbytes)	The largest number of block nodes	Memory overhead (Bytes)	Memory overhead (%)
2	660	10,560	0.5%
4	116	1,856	0.04%
8	107	1,712	0.02%
16	158	2,528	0.015%

VI. CONCLUSION

Flash memory is rapidly replacing hard disks for most consumer electronics devices. Most software and governing algorithms, however, are still based on hard disks even though flash memory exhibits quite different operational behaviors. From this observation, we proposed a novel flash-aware buffer management policy which selects a victim block to be replaced from the buffer based on the ratio of valid pages in the block rather than based on its recency. Though this idea was proposed to reduce the number of page copies and erase operations, it turns out to help small metadata files to outlive long multimedia files in the buffer. Since metadata is usually accessed repetitively while multimedia file is accessed once in a while, our scheme also increases the hit ratio which is the most important metric for a buffering scheme.

Simulation studies show that our scheme reduces the number of write operations on flash memory by 29 % and the number of erase operations by 19 %. Boosted by these gains, the execution time is also reduced by 17 % compared with a general LRU buffer management policy.

We plan to implement FAB in the real platform and to evaluate it with various workloads used in PMPs.

REFERENCES

[1] F. Douglis, R. Caceres, M. Kaashoek, P. Krishnan, K. Li, B. Marsh, and J. Tauber, "Storage Alternatives for Mobile Computers," *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI-1)*, 1994, pp. 25-37.

[2] *http://www.apple.com/ipodnano*

[3] Intel Corporation, "3 Volt Synchronous Intel StrataFlash Memory," *http://www.intel.com/*

[4] M. Slocombe, "Samsung CEO: NAND Flash Will Replace Hard Drives," *http://digital-lifestyles.info/display_page.asp?section=platforms&id=2573*, Sep. 2005.

[5] J. Kim, J. M. Kim, S. H. Hoh, S. L. Min, and Y. Cho, "A Space-efficient Flash Translation Layer for CompactFlash Systems," *IEEE Trans. Consumer Electron.,* vol. 48, no. 2, pp. 366-375, May 2002.

[6] *CompactFlash Association, http://www.compactflash.org/.*

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

[8] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S.-W. Park, and H.-J. Song, "FAST: An FTL Scheme with Fully Associative Sector Translations," *UKC 2005*, August 2005

[9] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-memory based File System," *Proceedings of the USENIX Winter Technical Conference*, 1995, pp. 155-164.

[10] M. L. Chiang, P. C. H. Lee, and R. C. Chang, "Using Data Clustering to Improve Cleaning Performance for Flash Memory," *Software-Practice and Experience*, vol. 29, no. 3, pp. 267-290, 1999.

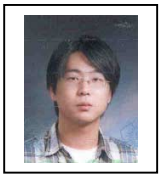
[11] M. Wu and W. Zwaenepoel, "eNVy: A Non-volatile, Main Memory Storage System," *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-6)*, 1994, pp. 86-97.

[12] J. H. Lee, G. H. Park, and S. D. Kim, "A New NAND-type Flash Memory Package with Smart Buffer System for Spatial and Temporal Localities," *Journal of Systems Architecture*, Vol. 51, No. 2, pp. 111-123, Feb. 2005.

[13] C. Park, J. Kang, S. Y. Park, and J. Kim, "Energy-Aware Demand Paging on NAND Flash-based Embedded Storages," *Proceedings of the 2004 International Symposium on Low Power Electronics and Design (ISLPED'04)*, 2004, pp. 338-343.

[14] A. Dan and D. Towsley, "An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes," *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, 1990, pp 143-152.

Heeseung Jo received the BS degree in computer science from Sogang University, Korea, in 2000, and the MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Korea, in 2006. He worked as an engineer at the mobile platform service team of KTF, Korea, from 2001 to 2004. He is a PhD candidate of KAIST. His research interests include Flash memory, file system, embedded system, fast boot, and low power.



Jeong-Uk Kang received his B.S., and M.S. degrees in Computer Science Division, Dept. of EECS from Korea Advanced Institute of Science and Technology (KAIST) in 1998, and 2000, respectively. Currently, he is enrolled in the PhD program in Computer Science Divison, KAIST. His research interests include operating systems and storage systems.

Seon-Yeong Park received the BS degree in computer engineering from Chungnam National University, Korea, in 1999, and the MS degree in computer science from Korea Advanced Institute of Science and Technology (KAIST), Korea, in 2001. She worked as a research member at the computer and software lab of Electronics and Telecommunications Research Institute (ETRI), Korea, from 2001 to 2003. She is a PhD candidate of KAIST. Her research interests include flash memory, file system, and cache management.

Jin-Soo Kim received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University, Korea, in 1991, 1993, and 1999, respectively. He was with the IBM T. J. Watson Research Center as an academic visitor from 1998 to 1999. He is currently an assistant professor of the department of electrical engineering and computer science at Korea Advanced Institute of Science and Technology (KAIST). Before joining KAIST, he was a senior member of research staff at Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002. His research interests include flash memory-based storage and operating systems.

Joonwon Lee received the B.S. degree from Seoul National University, in 1983 and the M.S. and Ph.D. degrees from the College of Computing, Georgia Institute of Technology, in 1990 and 1991, respectively. From 1991 to 1992, he was with IBM T. J. Watson Research Center where he was involved in developing scalable-shared memory multiprocessors. He is currently a faculty member at KAIST. His research interests include operating systems, computer architectures, and low power embedded software.