DOI:10.5392/IJoC.2010.6.4.022

Memory Compaction Scheme with Block-Level Buffer for Large Flash Memory

Weonil Chung
Dept. of Information Security Engineering
Hoseo University, Asan, Chungcheongnam-Do, Korea

Liangbo Li
Dept. of Computer and Information Engineering
Inha University, Inchoen, Korea

ABSTRACT

In flash memory, many previous garbage collection methods only merge blocks statically and do not consider the contents of buffer. These schemes may cause more unnecessary block erase operations and page copy operations. However, since flash memory has the limitation of maximum rate and life cycle to delete each block, an efficient garbage collection method to evenly wear out the flash memory region is needed. This paper proposes a memory compaction scheme based on block-level buffer for flash memory. The proposed scheme not only merges the data blocks and the corresponding log block, but also searches for the block-level buffer to find the corresponding buffer blocks. Consequently, unnecessary potential page copying operations and block erasure operations could be reduced, thereby improving the performance of flash memory and prolonging the lifetime of flash memory.

Keywords*: Flash Memory, Memory Compaction, Garbage Collection, Block-level buffer, FTL.*

1. INTRODUCTION

As the usage of mobile devices and mobile phones has rapidly increased, flash memory has been considered as the next-generation storage systems to replace the hard-disk because it has some advantages such as faster access speed, lightweight, low power consumption, small size and shock resistance. However, since flash memory is characterized by its erase-before-write operation, it must be erased before new data is written to a given physical location. Unfortunately, write operations are performed in unit of sector, while erase operations are executed in unit of block, usually, a block consists of many sectors. Besides, flash memory will accumulate obsolete sectors after lots of updates due to the erase-before-write characteristic. To make space for new blocks, obsolete sectors must be reclaimed. The only way to reclaim a sector is to erase an entire unit, in which this process is called garbage collection. Moreover, flash memory can be erased in limited times. Therefore, flash memory requires a well-designed garbage collection scheme to evenly wear out the flash memory region [1]-[5].

Some previous work focus on how to design a well FTL (Flash Translation Layer) and how to reduce write operations using buffer mechanism [6]-[10]. FTL is the driver that works in conjunction with an existing operating system to make linear flash memory appear to the system like a disk drive. The key role of FTL is to redirect each write request from the host file system to an empty area that has been already erased in advance. The buffer mechanism is also been used in flash memory to reduce the write requests. Since the erase-before-write characteristic of flash memory, frequent updates will make the performance of flash memory decrease. Write buffer cache could gather these frequent updates and make them to one write request. Therefore, buffer mechanism reduces the write operations to flash memory.

Among the previous works [11]-[14], the garbage collection is done only considered the contents of flash memory. Merging the data block and log block is to copy valid pages in them to a new block. However, when the contents in buffer are flushed to flash memory, the pages in new block produced by garbage collection are possible to become invalid. Under this situation, flash memory needs more pages copy operations and block erase operations in next garbage collection process. Actually, it can be avoided by making a well garbage collection scheme.

In this paper, we propose a novel garbage collection method called block-level buffer garbage collection. When FTL needs to merge blocks during the garbage collection process, it will refer to the contents of buffer. By examining the contents of buffer, FTL copies the best corresponding buffer blocks to flash memory or selects another appropriate block as victim block to improve the performance of flash-based storage

* *Corresponding author, E-mail: wnchung@hoseo.edu*
Manuscript received Sep. 17, 2010 ; accepted Dec. 20, 2010

system. In shortly, the proposed method is divided into two parts: how to merge blocks based on block-level buffer and how to select a log block as victim block. We focus on these two issues in this paper. First, our novel garbage collection method merges block with three kinds of blocks: log blocks, data blocks and buffer blocks. Second, our victim block selection method selects a log block as victim block through searching the contents of buffer. The proposed method can reduce unnecessary potential page copying numbers and unnecessary potential block erasure numbers.

2. RELATED WORKS

2.1 Flash Memory

Flash memory is a type of nonvolatile, electrically erasable programmable read-only memory. In this paper, our proposed method mainly focuses on NAND flash memory.

NAND flash memory consists of blocks, each of which consists of pages [7]. There are three basic operations for a NAND flash memory: read, write and erase. Read and write operations are performed on a page basis, while an erase operation is executed on a block basis. There are a few drawbacks as follows: 1) No in-place-update: The memory must be erased before new data can be written. The worse problem is that the erase operation is performed block by block, while the write operation is performed page by page. 2) Asymmetric operation costs: For flash memory, read operations are faster than write operations. In addition, as a write operation may accompany an erase operation, the write operational latency becomes even longer. 3) Uneven wear-out: The number of erasures on each block is limited, to 100,000 or 1,000,000 times. Once the number is reached, the block cannot be used any more.

Therefore, the number of write and erase operations should be minimized not only to improve the overall performance but also to maximize the lifetime of NAND flash memory.

2.2 Flash Translation Layer (FTL)

FTL is a translation layer between the native file system and flash memory [10]. The main role of FTL is to emulate the functionality of block device with flash memory. It emulates a hard disk, and provides logical sector updates. FTL achieves this by redirecting each write request form file system 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 logical sector number to physical location. Besides the address translation from logical sectors to physical sectors, FTL carries out several other important functionalities, such as guaranteeing data consistency and reclaiming the discarded data blocks for reuse.

The mapping between the logical address and the physical address can be managed at sector, block, or hybrid level. Therefore, the mapping scheme is categorized with sector-level mapping, block-level mapping, and hybrid mapping [6],[8],[9]. When free log blocks are not sufficient, the merge operation called garbage collection will happen. Since any log block is associated with data blocks, merge operation is to copy valid data from log block and data blocks to new free block. While executing the merge operation, multiple page copy operations and erase operations are invoked. Therefore, merge operations seriously degrade the performance of flash memory because of extra operations.

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.

2.3 Flash Buffer Cache

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 evicting them at the same time, 2) evict 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 expensive, and 3) detect sequential page writes and evict those sequential pages preferentially and simultaneously [2], [3], [15].

The buffer can manage data in page-level management buffer and block-level management buffer. In page-level scheme, pages are evicted to flash memory page by page. When buffer pages are managed in a block-level scheme, pages are clustered by corresponding block number in the flash memory. 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 shows better overall performance than the page-level buffer management policy [12], [16]-[18].

2.4 Cleaning Policies

Cleaning policies determine when to clean, which blocks to clean, and where to write data in order to minimize cleaning cost. The cleaning cost includes erasure cost and the migration cost for copying valid data into other blocks. In this paper, we measure the quality of cleaning policy by block erasure numbers and page copying numbers with previous works such as *greedy* policy, *cost-benefit* policy [11], and *cost age times* policy [13]. *Greedy* policy considers only cleaning cost.

Greedy policy always selects blocks with the largest amount of garbage for cleaning, hoping to reclaim as much space as possible with the least cleaning work. *Cost-benefit* policy [14] chooses to clean blocks that maximize the formula: $(benefit/cost) = (age*(1-u))/2u$, where u is the block utilization and $(1-u)$ is the amount of free space reclaimed. The *age* is the time since the most recent modification (i.e., the last block invalidation) and is used as an estimate of how long the space is likely to stay free. The cost of cleaning a block is $2u$ that one u is to read valid pages and the other u is to write them back. *Cost Age Times (CAT)* policy [16] chooses to clean blocks that minimize the formula: $(CleaingCost)*Age^{-1}*(number\ of\ Cleaning)$. The *Cleaning Cost* is defined as the cleaning cost of every useful write to flash memory as $u/(1-u)$, where u is the percentage of valid data in a block. *Every (1-u)* write incurs the cleaning cost of writing out u valid data. The *Age* is defined as the elapsed time since the block was created. The *Number of*

Cleaning is defined as the number of times a block has been erased.

In comparison, *Cost-benefit* policy considers cleaning cost and age of data. And *cost age times* policy considers cleaning cost, age of the data, and number of cleaning. But all the above cleaning policies focus on selecting a victim block statically in flash memory. The proposed method is different from them, since we not only consider the static state of flash memory, but also consider the dynamic state of buffer cache.

3. BLOCK-LEVEL BUFFER AWARE MERGE

3.1 Buffer-aware Block Merge

In order to prevent potential unnecessary page migrations, the proposed method refers to the contents of buffer during the block merging process. There is trade-off between merge performance and the buffer hit ratio. First, move up-to-date blocks in the buffer into new allocated data blocks can improve the merge performance. Second, if the corresponding blocks will not be evicted to flash memory in the near future, they will be updated frequently. It causes that moving these blocks into flash memory will make the buffer hit ratio decreased. In this case, it is beneficial to choose another log block as victim log block. Therefore, the issue can be viewed as how to merge blocks and how to select victim log block.

As mentioned before, the cleaning policy determined when to clean, which blocks to clean and where data to write. In this paper, we do not discuss the issue of when to clean, since the cleaning can start when the number of free blocks becomes lower than a threshold. However, we will set a threshold to test the proposed method in our experiments. If no more empty sectors exist in the log blocks, the proposed garbage collection approach chooses one of the log blocks as victim and merges the victim block with its corresponding data blocks and its corresponding buffer blocks.

The merge operation proceeds as follows: First, given a log block as victim block, find the corresponding data blocks and allocate the same amount of free blocks. Before processing the merge operation, it is necessary to determine the log block that will serve as the merge target. Three existing cleaning policies which selected a victim log block according to different features are described. In this paper, we will present a new log block selection method which is distinct from the existing cleaning policies. In hybrid mapping scheme of FTL, there is a block-level mapping table addition to a page-level mapping table, in which the page-level mapping table contains the separate page mapping information between log blocks and the corresponding data blocks. Therefore, we can obtain the corresponding data blocks when given a log block through searching the page-level mapping table. Second, search the buffer to find whether the blocks which have the same number to the corresponding data blocks exist or not. In block-level buffer, data are organized in unit of block. Each block has a unique number called logical address which is identical to the logical number of data blocks in flash memory. Therefore, it is easy to find the corresponding buffer blocks when we obtained the corresponding data blocks. Third, flush the sectors in buffer blocks to free blocks, and copy the most up-to-data version from the log blocks to free blocks, then fills each empty sector in the free block with its corresponding sector in the data blocks. Fourth, erase the log block and corresponding data blocks.

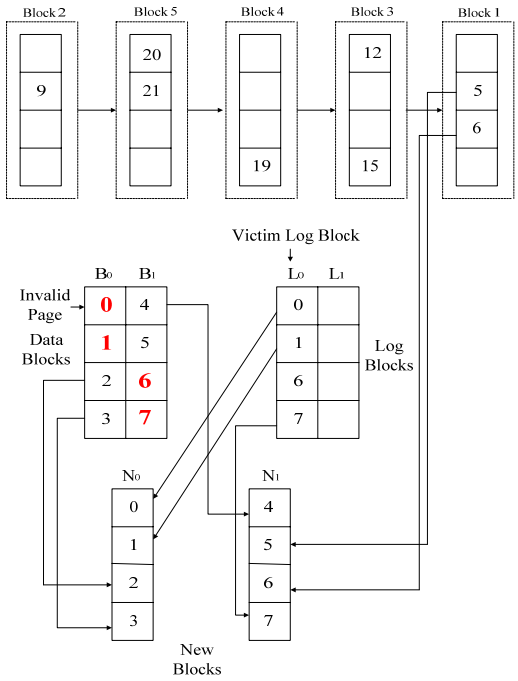

Fig. 1. Merge blocks refer to contents of buffer

Fig. 1 is the typical process of our proposed block-level buffer aware merge method. In this fig., B0 and B1 are data blocks, L0 and L1 are log blocks. To log block L0, data blocks B0 and B1 are its corresponding data blocks, block 1 in the buffer is its corresponding buffer block.

When log block L0 is selected as victim block, we find that B0 and B1 are its corresponding data blocks and there also exists corresponding buffer block 1 in the buffer, and then we allocate two new blocks N0 and N1 from free blocks. Next, sector 5 and 6 in buffer are first flushed into new blocks, then copy sector 0, 1 and 7 in log block to new blocks, finally fill the empty sectors using sector 2, 3 and 4 in data blocks. After merge all the blocks, erase log block L0 and data blocks B0 and B1.

3.2 Victim Block Selection

In step two of searching the buffer in the proposed approach, there may cause three situations according whether the blocks which have the same number to corresponding data blocks in buffer is found or not.

In case of not found, the garbage collection only needs to merge the log block and its corresponding data blocks. This process is done like the buffer-unaware merge operation. Fig. 2 shows the relationship between buffer and the flash memory.

In case of found and the found blocks belong to the least recently used blocks, if the blocks locate at the near end of the buffer list, this means these blocks will be flushed to flash memory in the shortly future. It is appropriate to move these sectors in the buffer into the new free blocks. Our proposed

method is suitable for this situation. Fig. 3 shows the relationship between buffer and the flash memory.

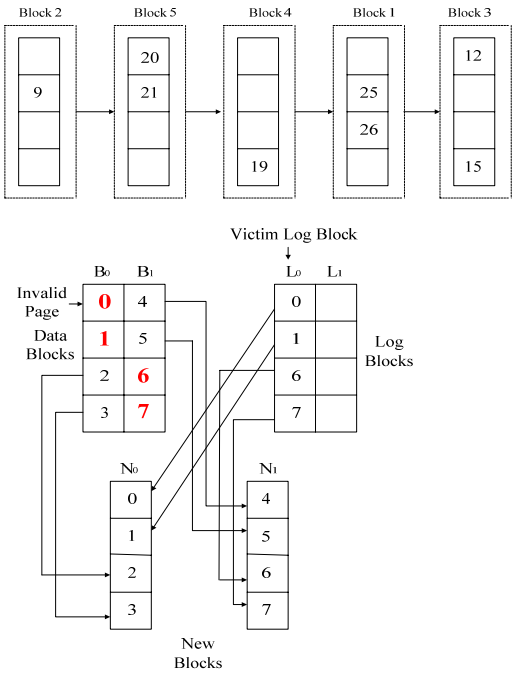

Fig. 2. No corresponding blocks in buffer

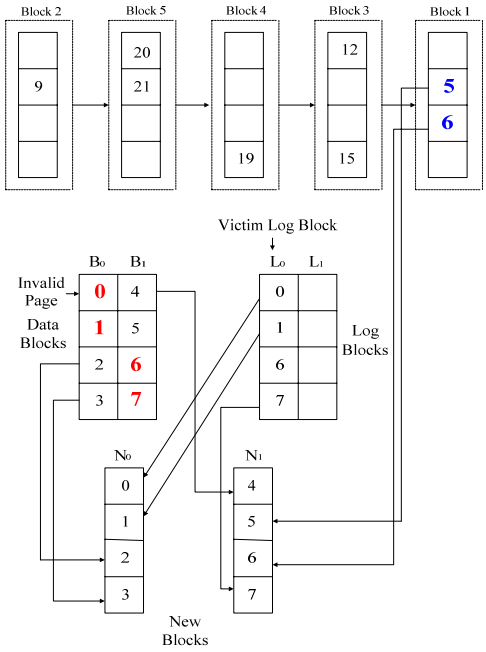

Fig. 3. Corresponding blocks locate at the near end of the buffer

In case of found but the found blocks are hot blocks, the blocks locate at the near start of the buffer list, which means these sectors in the found blocks will be updated frequently. If we choose these blocks to flush to flash memory, the buffer hit ratio will decrease. Therefore, it is beneficial to find another log block as victim log block. Fig. 4 shows the relationship between buffer and the flash memory.

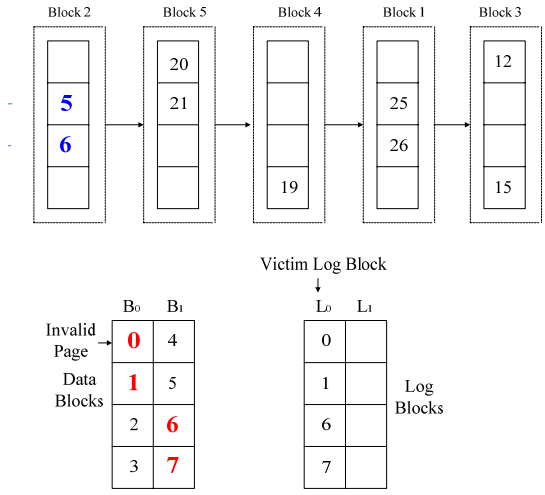

Fig. 4. Corresponding blocks locate at the near start of the buffer

Through our analysis, whether or not evicting a corresponding buffer block to flash memory depends on the block position where blocks reside in the buffer. Hot blocks are these which locate at the near start of the buffer, while clod blocks are those which locate at the near end of the buffer. Since hot blocks are accessed frequently, we should avoid evict hot blocks in advance. Otherwise, the buffer hit ratio will decrease. To address this issue, the buffer needs a threshold line to divide which to hot block area and cold block area. Only blocks which reside in clod block area could be flushed to flash memory. Fig. 5 shows the position of hot blocks and cold blocks in buffer cache.

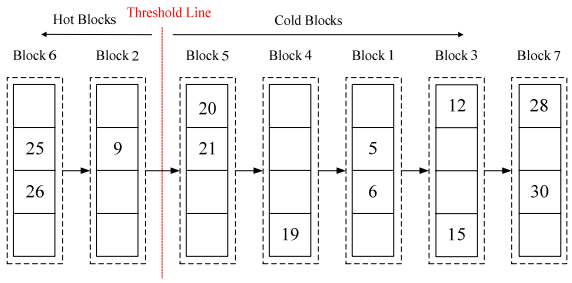

Fig. 5. The position of hot blocks and cold blocks

Here we use a novel concept named locality probability to judge the importance of the blocks in the buffer. The idea of locality probability is described as follows:

We assume the current number of blocks in buffer is n, the basic locality probability of the block which reside in the end of the buffer list is p, which is the lowest value, the difference value between two blocks is x, this means the locality probability of the front block is larger than its back block, and their difference value is x, the total sum of all block locality probability is 1. So we get:

$$P+ (p+x) + (p+2x) + \cdots\cdots + (p+(n-1)x) = 1 \qquad \text{Eq. (1)}$$

If we set the difference value x a specific value, according to

the number n, we can calculate the basic locality probability p and the locality probability of every block. In the real world, set a threshold of the locality probability depending on specific devices. The blocks whose locality probability larger than the threshold considered as hot blocks, and less than the threshold considered as least recently used blocks.

The buffer space is partitioned into two areas by the threshold line. Usually, the hot block area holds a small part of the buffer, while the cold block area holds most of the buffer. We use a partition parameter α ($0 \le \alpha \le 1$) to divide the buffer. The partition parameter is defined as the ratio of hot blocks to total buffer blocks. If α = 0.1, then 10 percent of the total number of blocks in the buffer are hot blocks and the remaining 90 percent of the blocks are cold blocks. This measure is a variant of locality probability. According to the definition of locality probability, we could calculate the locality probability of every block. For example, n = 10, set x = 0.01, we get the basic locality probability p = 0.055 according to Eq. 1. In turn, we could calculate every block's locality probability, from the top of buffer to the end, 0.145, 0.135, 0.125, 0.115, 0.105, 0.095, 0.085, 0.075, 0.065 and 0.055. If the threshold value is 0.130, 20 percent of the total number of blocks art hot blocks and 80 percent of the blocks are cold blocks. For the buffer management policies which manage blocks with hot and cold blocks, we need not divide the buffer to two partitions. For example, using Cold and CLC(Largest Cluster Policy) [15] to manage buffer, our proposed method only need to treat the size-independent LRU cluster list as hot block area and size-dependent LRU cluster list as cold block area. In this paper, we use the parameter α to describe the hot block area and cold block area since it is convenient to implement. Our experiments treat the buffer for the general purpose and use LRU policy to manage the buffer.

In the section of existing cleaning policies, we listed several cleaning policies: *greedy* policy, *cost-benefit* policy and *Cost Age Times (CAT)* policy. In garbage collection process, a main task is to choose a block as victim block among lots of used blocks. Through calculating under different cleaning policies, used blocks are listed in a queue. The block with the highest satisfied condition will be selected as victim block. Used blocks are selected one by one in the queue.

In our proposed method, based on the used block queue, we will check whether or not a used block can be selected as victim block in Fig. 6.

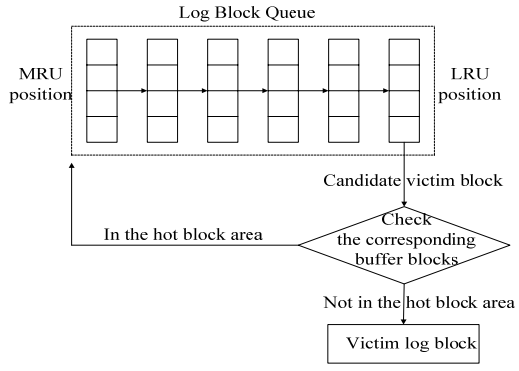

Fig. 6. The process of selecting victim log block

For simplicity, we use LRU policy to manage the log blocks instead of the existing cleaning policies in this paper. That means log blocks are managed in a queue under LRU policy, the least recently used log block is first selected as a candidate victim block. Then check the corresponding buffer blocks of this log block, if the corresponding buffer blocks do not reside in the hot block area, this log block is selected as victim block. If the corresponding buffer blocks reside in the hot block, move this log block to the MRU position of the queue and select next log block as candidate victim block.

In the process of checking the corresponding buffer blocks, we will check whether or not the corresponding buffer blocks locate at the hot block area. If the corresponding buffer blocks do not locate at the hot block area, it may locate at the cold block area or it may do not appear at the buffer cache. According to this, we will make the correct decision that merging blocks with buffer blocks or not. Therefore, our proposed method comes. Fig. 7 shows the details of the processing.

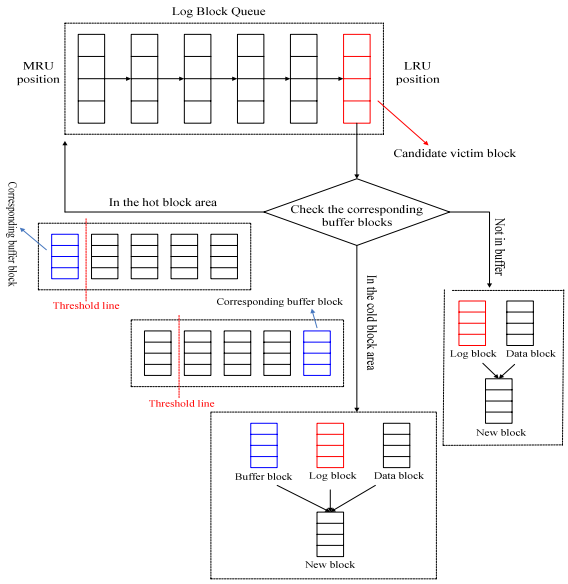

Fig. 7. The whole process of the proposed method

Above all, our victim block selection depends on the current contents in the buffer cache, which is dynamic and different from the others cleaning policies.

4. PERFORMANCE EVALUATIONS

4.1 Experimental Environment

As analyzed before, the main advantage of proposed method is to reduce the potential unnecessary page migration costs and block erase costs. In other words, during the process of garbage collection, our approach can improve flash wear-leveling performance and prolong the lifetime of flash memory by reducing unnecessary page copy costs and block erase costs.

The experiments of proposed method will generate the block erasure numbers and page copying numbers, so we get this information as the results to measure the experimental performance. We assume that every block consists of 4 pages, the flash memory has 100 blocks, and the log block group has

10 log blocks, the buffer can maintain 10 blocks when it is full. The system considers the number of pages as input. With different input page numbers, different garbage collection scheme will produce different block erasure numbers and page copying numbers. The fewer numbers produced, the better that scheme is. The simulation environment is shown in Table 1.

Table 1. Experimental Environments

Item	Configuration
Computer	PC
CPU	Intel Core2 Duo 2Ghz
Main Memory	2 GB
HDD	500GB
OS	Microsoft Windows XP
Language	MS Visual C++ 2005

4.2 Experimental Results

We divide the experimental results as two parts according to the access mode of random access and high locality access. High locality access means that accessing more frequently and more intensively concentrate at a specific area of flash memory. Because the cost-benefit cleaning policy and CAT cleaning policy is similar besides the number of cleaning characteristic, we only compare our proposed method with the greedy cleaning policy and the cost-benefit cleaning policy.

4.2.1 Performance of random access

First we compare the performance of buffer-aware merge and buffer-unaware merge. Buffer-aware merge is our proposed scheme and buffer-unaware merge is the traditional merge operation which does not refer to the contents of buffer.

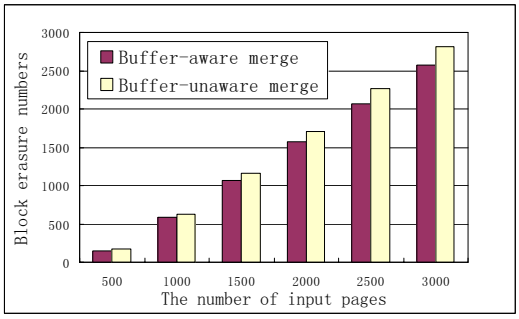

Fig. 8. Block erasure numbers

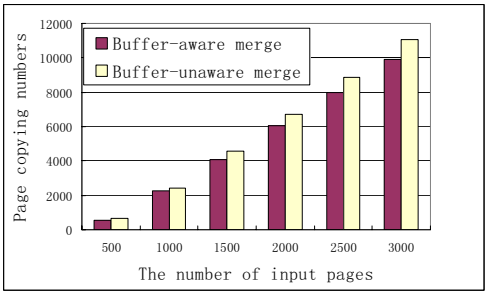

Fig. 9. Page copying numbers

In Fig. 8, the x-axis denotes the random input page numbers for write to flash memory and the y-axis represents the block erasure numbers when collecting garbage. In Fig. 9, the x-axis denotes the same meaning to Fig. 8 and the y-axis represents the page copying numbers when collecting garbage.

From these two figures, we find that buffer-aware merge scheme can save more block erasure numbers and page copying numbers than buffer-unaware merge scheme.

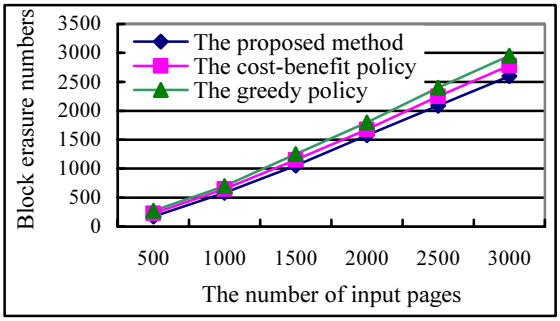

Fig. 10. Erasure numbers of comparing with existing cleaning policies

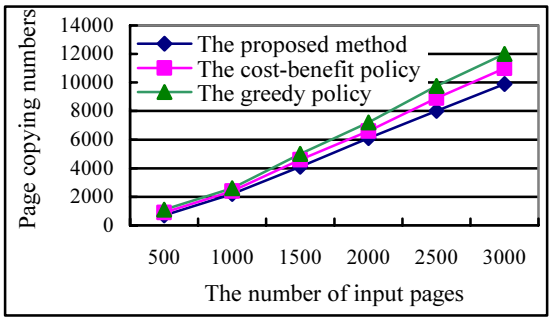

Fig. 11. Copying numbers of comparing with existing cleaning policies

Next, we compare our proposed scheme with the existing cleaning policies. Fig. 10 and 11 shows the results of block erasure numbers and page copying numbers respectively. In Fig. 10 and 11, the line marked star denotes the greedy policy, the line marked triangle denotes the cost-benefit policy and the line marked rectangle represents our proposed method.

In random access mode, the cost-benefit cleaning policy is better than the greedy cleaning policy, because the cost-benefit policy not only considers the block utilization but also considers the age of blocks. Our proposed method outperforms the two policies, since we consider the dynamic contents of buffer.

4.2.2 Performance of high locality access

The case of high locality access is different from random access. Fig. 12 and 13 show the compared results of buffer-aware merge scheme and buffer-unaware merge scheme.

Compared to random access, buffer-aware merge scheme can reduce more block erasure numbers and page copying numbers than buffer-unaware merge scheme in high locality access mode. This is because in high locality access mode, data are concentrated and buffered in buffer cache. Through checking the contents of buffer cache in advance, many unnecessary block erase operations and page copy operations

can be avoided. Therefore, the performance can be improved more than in random access.

In Fig. 13, the page-copying numbers is reduced much more than block-erasure numbers in Fig. 12. The reason is that high locality access causes blocks with full obsolete pages much more than partial obsolete pages. Therefore, switch merge operations is performed much more than partial merge operations and full merge operations. In switch merge, only block erase operations are necessary.

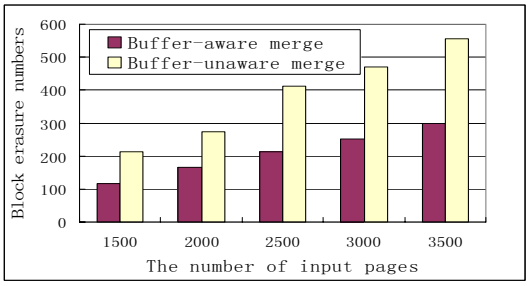

Fig. 12. Erasure numbers of high locality

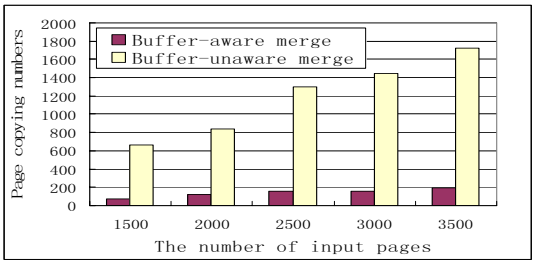

Fig. 13. Copying numbers of high locality

Finally, we compared the performance of high locality access with the existing cleaning policies.

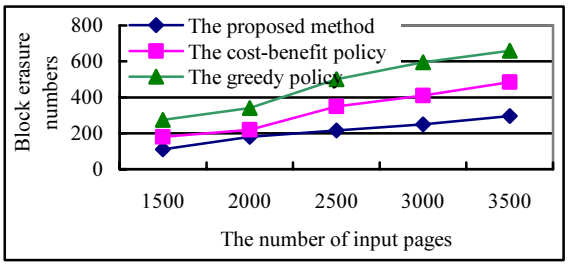

Fig. 14. Erasure numbers compared with existing cleaning policies

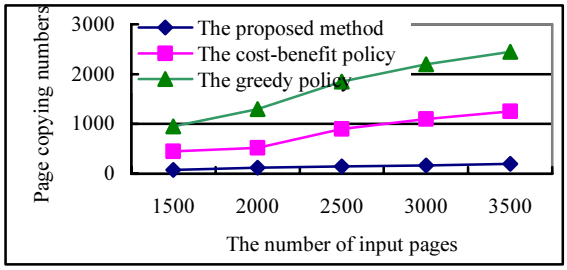

Fig. 15. Copying numbers compared with existing cleaning policies

Both in random access and high locality access, our proposed method outperforms the greedy policy and the cost-benefit policy.

5. CONCLUSIONS

In this paper, a block-level buffer aware garbage collection technique which searches the contents of buffer cache during the process of merging blocks is presented. The approach is divided into two parts: buffer-aware block merge and victim block selection. The former focuses on how to merge buffer blocks with data blocks and log blocks, while the latter focuses on how to select victim blocks so as to ensure the buffer hit ratio not decreased. Our victim block selection approach is dynamic, which selects blocks depending on the contents of buffer cache and is different from the existing cleaning policies. Compared with the existing garbage collection methods, the proposed method reduces the block erase operations and page migration operations when collecting the same amount of garbage.

In the future works, we will make the explicit analysis about locality probability to make the best performance between the buffers hit ratio and the flash block merges cost.

REFERENCES

[1] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," ACM Computing Surveys, 2005.

[2] S.W. Lee, D.J. Park, T.S. Chung, D.H. Lee, S.W. Park, and H.J. Song, "A Log Buffer-based Flash Translation Layer using Fully-Associative Sector Translation," ACM Transactions on Embedded Computing Systems, vol. 6, no.3, 2007.

[3] C. Park, W.M. Cheon, J.G. Kang, K.G. Roh, W.H. Cho, and J.S. Kim, "A Reconfigurable FTL Architecture for NAND Flash-based Applications," ACM Transactions on Embedded Computing Systems, vol. 7, no. 4, 2008.

[4] J.U. Kang, H.S. Jo, J.S. Kim, and J.W. Lee, "A Super Block-based Flash Translation Layer for NAND Flash Memory," Proc. International Conference on Embedded Software, 2006, pp. 161-170.

[5] J. Kang, J.M. Kim, S.H. Noh, S.L. Min and Y. Cho, "A Space-efficient Flash Translation Layer for Compact Flash Systems," IEEE Transactions on Consumer Electronics, vol. 48, no.2, 2006, pp. 366-375.

[6] S.Y. Park, D.W. Jung, J.U. Kang, J.S. Kim and J.W. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2006, pp. 234-241.

[7] H. Kim and S.J. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," Proceedings of the 6th USENIX Conference on File and Storage Technologies, 2008.

[8] H. Jo, J.U. Kang, J.S. Kim, and J. Lee, "FAB: Flash-aware Buffer Management Policy for Portable Media

Players," IEEE Transactions on Consumer Electronics, vol. 52, no.2, 2006, pp. 485-493.
[9] S.W. Lee and B.K. Moon, "Design of Flash-based DBMS: An In-page Logging Approach," International conference on Management of Data, Beijing, China, 2007, pp. 55-66.
[10] L.P. Chang and T.W. Kuo, "An Efficient Management Scheme for Large-scale Flash-memory Storage Systems," Symposium on Applied Computing, 2004, pp. 862-868.
[11] K.H. Park and S.H. Lim, "An Efficient NAND Flash File System for Flash Memory Storage," IEEE Transactions on Computers, vol. 55, 2006, pp. 906-912.
[12] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification," White Paper, http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf, 1998.
[13] A. Kawaguchi, S. Nishioka, and H. S. Motoda, ”A Flash-memory based File System,” USENIX Association, 1995, pp. 13-23.
[14] S. Y. Kang, S. M. Park, H. Y. Jung, H. K. Shim, and J. Y. Cha, ”Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices,” IEEE Computer Society, vol. 58, no. 6, 2009, pp. 744-758.
[15] M. L. Chiang and R. C. Chang, “Cleaning Policies in Mobile Computers using Flash Memory,” Elsevier Science Inc, vol. 48, no. 3, 1999, pp. 213-231.
[16] I. Koltsidas and S. D. Viglas, “Flashing up the Storage Layer,” VLDB Endowment, vol. 1, no. 1, 2008, pp. 514-525.
[17] A. Birrell, M. Isard, C. Thacker, and T. Wobber, “A Design for High-performance Flash Disks,” ACM SIGOPS Operating Systems Review, vol. 41, no. 2, 2007, pp. 88-93.
[18] S. W. Lee, B. K. Moon, C. N. Park, J. M. Kim and S. W. Kim, “A Case for Flash Memory SSD in Enterprise Database Applications”, International Conference on Management of Data, 2008, pp. 1075-1086.

Weonil Chung
He received the B.S., Ph.D. in computer science and Information Engineering from Inha University, Korea in 1998, 2004 respectively. Since 2007, he has been with Hoseo University. His main research interests include spatial data stream, and database security.

Liangbo Li
He received a B.S. degree in computer engineering from Chongqing University, China in 2009. Currently he is taking up M.S. course in Computer and Information Engineering at Inha University. He research interests include spatial database, POI, and data stream.