BAGC: Buffer-Aware Garbage Collection for Flash-Based Storage Systems

Sungjin Lee, Dongkun Shin, *Member*, *IEEE*, and Jihong Kim, *Member*, *IEEE*

Abstract—NAND flash-based storage device is becoming a viable storage solution for mobile and desktop systems. Because of the erase-before-write nature, flash-based storage devices require garbage collection that causes significant performance degradation, incurring a large number of page migrations and block erasures. To improve I/O performance, therefore, it is important to develop an efficient garbage collection algorithm. In this paper, we propose a novel garbage collection technique, called buffer-aware garbage collection (BAGC), for flash-based storage devices. The BAGC improves the efficiency of two main steps of garbage collection, a block merge step and a victim block selection step, by taking account of the contents of a buffer cache, which is typically used to enhance I/O performance. The buffer-aware block merge (BABM) scheme eliminates unnecessary page migrations by evicting dirty data from a buffer cache during a block merge step. The buffer-aware victim block selection (BAVBS) scheme, on the other hand, selects a victim block so that the benefit of the buffer-aware block merge is maximized. Our experimental results show that BAGC improves I/O performance by up to 43 percent over existing buffer-unaware schemes for various benchmarks.

Index Terms—NAND flash memory, flash translation layer (FTL), buffer management, garbage collection

1 Introduction

NAND flash memory is widely used as a storage device replacing hard disk drives, because of its low-power consumption, high performance, and high reliability [3]. Unlike HDDs, NAND flash memory operates differently in two aspects. First, its "erase-before-write" architecture requires that previous data has to be erased before new data is written to it. Second, the unit size of an erasure operation is not the same as that of a read or write operation. Reads and writes are performed in a unit of a page whose size is 2-8 KB [4], but erasure operations are performed in a unit of a block consisting of multiple pages.

To handle these unique characteristics and to emulate the functionality of a normal block device, a special software layer, called a flash translation layer (FTL), is usually employed between a file system and flash memory [5], [6], [7], [8], [9], [10], [11], [12], [13]. In designing the FTL, there are two kinds of important issues: address translation and garbage collection. Because of the erase-before-write constraint, the FTL uses an *out-place update* policy that writes up-to-date data to a new free page instead of updating the original page. For this purpose, the FTL provides an address translation policy which maps a logical page address to a physical page address. An out-place update policy generates invalid pages with out-of-date data that must be reclaimed by garbage collection later.

There are a variety of FTL schemes, including page-level FTLs [5], [6], [7] and block-level FTLs [8], but hybrid-level FTLs [9], [10], [11] are widely used in many flash devices, including USB sticks [12] and solid-state drives (SSDs) [13], [14]. The popularity of the hybrid-level FTLs is mainly due to the fact that they enable to maintain a small mapping table while providing good performance. In the hybrid-level FTLs, physical blocks are grouped into log blocks or data blocks. Log blocks are used for storing incoming data temporarily and are managed by a page-level mapping table. Data blocks are used as ordinary storage space with a block-level mapping table. When all free log blocks are exhausted, the FTL performs garbage collection to make a free log block. This garbage collection involves two main steps: *victim block selection* and *block merge*. The victim block selection step finds a victim log block with invalid pages to be reclaimed. All valid pages in the victim log block are copied to a free block during the block merge step. The victim log block is then erased and becomes a new free block.

Garbage collection incurs significant overhead because it requires many page migrations as well as block erasures. One of the promising approaches to reduce the garbage collection overhead is to use a buffer cache on top of the FTL. By using a buffer cache, we can reduce a large number of page writes to the FTL, which in turn incur garbage collection, and can improve the sequentiality of writes so that less migration overhead is required. Many flash devices, thus, employ a buffer cache as one of the essential components.

However, with a simple combination of a buffer cache and the FTL, it is difficult to take full advantage of using a buffer cache. In our observation, with a buffer cache, the FTL performs *unnecessary page migrations* frequently that

- *S. Lee is with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139. E-mail: chamdoo@csail.mit.edu.*
- *D. Shin is with the School of Information and CommunicationEngineering, Sungkyunkwan University, Suwon, Gyeonggi-do 440-746, Korea. E-mail: dongkun@skku.edu.*
- *J. Kim is with the Department of Computer Science and Engineering, Seoul National University, Gwanak-ro, Gwanak-gu, Seoul 151-742, Korea. E-mail: jihong@davinci.snu.ac.kr.*

Manuscript received 9 Apr. 2012; revised 4 Aug. 2012; accepted 29 Aug. 2012; published online 11 Sept. 2012.
Recommended for acceptance by E. Miller.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2012-04-0259.
Digital Object Identifier no. 10.1109/TC.2012.227.

move obsolete pages during garbage collection. This unnecessary page migration occurs when a page in a buffer cache is newly updated, but the FTL is not aware of its update because the page is not actually written to the FTL. In this case, the FTL can move out-of-date pages in flash memory for garbage collection, incurring lots of unnecessary page migrations.

These useless page migrations can be avoided if a buffer cache manager lets the FTL know which pages in flash memory are out-of-date so that obsolete pages are not copied during garbage collection [7], [18]. However, this approach seriously degrades reliability of data storage. This is because, after garbage collection, the obsolete pages are erased in the flash memory and the up-to-date data of those pages is kept only in a buffer cache. If a critical reliability-affecting event (e.g., a sudden power failure) occurs, the system cannot be recovered or roll-backed to the previous status. Therefore, a more sophisticated strategy is required which eliminates useless page migrations while ensuring high data reliability.

In this paper, we propose a novel garbage collection scheme, called buffer-aware garbage collection (BAGC). The proposed BAGC scheme identifies and eliminates unnecessary page migrations by means of examining the contents of a buffer cache. Our BAGC scheme consists of two techniques, the buffer-aware block merge (BABM) technique and the buffer-aware victim block selection (BAVBS) technique. BABM eliminates useless page migrations by *writing* up-to-date pages in a buffer cache to flash memory during block merges. By doing so, BABM not only reduces the number of future page writes to flash memory, but also lowers a future block merge cost, while providing a high degree of data reliability. BAVBS chooses a victim log block to maximize the benefit of buffer-aware block merges. BAVBS exploits the locality of pages in a buffer cache for a better decision in selecting a victim block. We have evaluated the proposed BAGC scheme in the context of several state-of-the-art FTL and buffer management schemes using a trace-driven simulator. Our experimental results show that BAGC improves the I/O performance by up to 43 percent over buffer-unaware schemes for various workloads.

This paper is organized as follows: After reviewing previous works in Section 2, we explain the motivation of our work in Section 3. We describe our target system architecture in Section 4. The proposed BABM and BAVBS schemes are described in Sections 5 and 6, respectively. Experimental results are presented in Section 7, and Section 8 concludes with a summary.

2 Related Work

There has been a considerable amount of research on a flash translation layer and a buffer management layer. However, little attention has been paid to approaches that consider two layers simultaneously.

Existing research on the FTL has focused on reducing the garbage collection overhead with a small mapping table. Thus, the hybrid-level FTLs have received serious attention. The hybrid-level FTLs can be categorized into three types depending on a block association policy: block-associative sector translation (BAST) [9], fully associative sector translation (FAST) [10], and set-associative sector translation (SAST) [11]. A block association policy determines how many data blocks share a log block, but gives no consideration to the correlation between a buffer cache and flash memory. With regard to victim selection, both BAST and FAST use the round-robin policy that chooses the least recently written log block as a victim block. The SuperBlock scheme [11], which is based on SAST, uses the utilization-based policy that selects the block with the fewest valid pages. However, none of them consider the contents of a buffer cache in selecting a victim block.

There also have been a lot of studies on a buffer cache of a flash device. The FAB scheme [15] is based on a block-level LRU buffer management (BLRU) policy, which evicts all pages in the same logical block to flash memory at the same time to improve the sequentiality of writes. FAB further improves the sequentiality of writes by evicting the block with the largest number of dirty pages from a buffer cache. The BPLRU scheme [16] is also based on the BLRU policy, but it eliminates random writes to flash memory by using the page padding technique. All these schemes reduce the garbage collection cost by lowering the number of writes or by improving the sequentiality of writes, but they consider neither useless page migrations nor victim block selection. A recently evicted-first (REF) buffer replacement policy takes account of log blocks to reduce the cost of block merge operations, but it has the same limitation in that it does not consider useless page migrations.

More recently, Li et al. [7] propose a duplication-aware garbage collection (DA-GC) technique for a virtual memory system with a flash device. DA-GC detects pages that reside on both main memory and flash memory, and then prevents them from being moved during garbage collection to avoid useless page migrations. The ignored pages containing duplicate data are erased from flash memory after garbage collection, leaving only the copies in main memory. Ji and Shin [18] propose a locality and duplication-aware garbage collection (LDA-GC) technique, which improves DA-GC for the hybrid-level FTLs. Similar to BAGC, both DA-GC and LDA-GC eliminate unnecessary page migrations. However, they are not suitable for a buffer cache which is used as a cache for a storage device; if a system failure occurs before duplicate data is written to flash memory, the data is inevitably lost. BAGC removes useless page migrations by flushing duplicate data *to flash memory*. Thus, BAGC does not adversely affect reliability of data storage.

3 Motivation

We first explain the benefit of making a garbage collector *buffer-aware* using a simple scenario. When a garbage collector selects a victim log block and performs a block merge operation, many page migrations are necessary. Our primary observation is that many of them would be *unnecessary* if a garbage collector could take into account the contents of a buffer cache.

Fig. 1 shows an example of a block merge in the FAST FTL [10]. The buffer cache has eight pages and two of

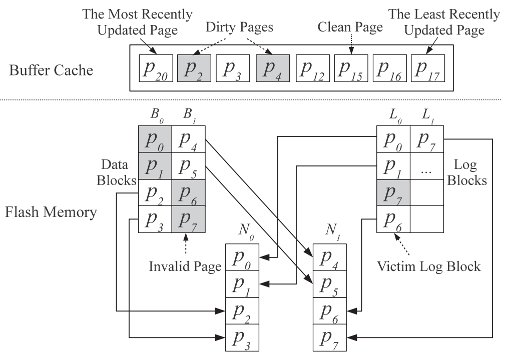


Fig. 1. An example of unnecessary page migrations.

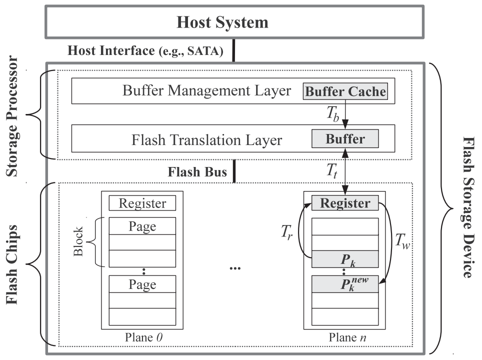


Fig. 2. An architectural overview of a target flash device.

them, p_2 and p_4, are dirty pages. In the flash memory, B_0 and B_1 are data blocks and L_0 and L_1 are log blocks. Each block is composed of four pages. Because of the updates on the pages p_0, p_1, p_6, and p_7, the log blocks L_0 and L_1 have the most recent version of these pages, thus making the original pages in B_0 and B_1 invalid. If the log block L_0 is selected as a victim block, two new data blocks N_0 and N_1 are allocated and then the valid pages in B_0, B_1, L_0, and L_1 are moved to N_0 and N_1. After the page migrations, the blocks B_0, B_1, and L_0 are erased. This type of a block merge is called a *full merge*.

To reclaim L_0, there were eight page migrations. However, p_2 (in B_0) and p_4 (in B_1) were moved *uselessly* because p_2 (in N_0) and p_4 (in N_1) are invalidated soon when the dirty pages p_2 and p_4 in the buffer cache are evicted to the flash memory. If p_2 and p_4 in the buffer cache were moved to N_0 and N_1 instead of p_2 and p_4 in B_0 and B_1, N_0 and N_1 have the most recent version without the useless page migrations for p_2 and p_4. To decide that p_2 and p_4 in the buffer cache should be moved, we need to take account of the contents of the buffer cache. This is the main motivation of our buffer-aware garbage collection technique.

Making garbage collection buffer-aware has two positive impacts on FTL performance. *First, it can reduce dirty page writes to flash memory*. Since dirty pages p_2 and p_4 are written to the data blocks when L_0 is being merged, these pages become clean and do not need to be written to the flash memory when they are evicted from the buffer cache. *Second, it can eliminate or delay a block merge that will occur in the near future*. If p_2 and p_4 are moved from B_0 and B_1, instead of from the buffer cache, these pages must be written to the log block (e.g., L_1) when they are evicted from the buffer cache. The log block L_1 is merged with the corresponding data blocks (e.g., N_0 and N_1) in the near future when all the free pages in L_1 are exhausted. However, if p_2 and p_4 in the buffer cache are directly written to N_0 and N_1 when L_0 is being merged, the block merge for L_1 can be delayed with more free pages. These positive impacts of buffer-aware garbage collection are called *potential benefits* because they are obtained at a later time.

If p_2 and p_4 in Fig. 1 are updated in the buffer cache after being written to flash memory, they must be rewritten to flash memory when they are evicted from the buffer cache later. In that case, the buffer-aware garbage collection for p_2 and p_4 becomes useless, limiting its positive effects on performance. Furthermore, if buffer-aware garbage collection is often performed for pages frequently updated in a buffer cache, it could negatively impact performance because it causes a lot of useless page writes to flash memory. For this reason, buffer-aware garbage collection must be carefully performed by considering the update probabilities of pages in a buffer cache. This is the motivation of our buffer-aware victim block selection technique.

In our experimental analysis, we found that 4.7-85.4 percent of the total page migrations were useless for various benchmark traces. On average, about 19.6 percent of the total page migrations were unnecessary. The performance of a storage device is thus improved greatly if useless page migrations are eliminated effectively.

4 Target System Architecture

Fig. 2 shows an architectural overview of our target flash device. Our target storage device interacts with a host system through a standard interface such as SATA and eMMC. On the storage side, the main storage processor is connected to the flash chips through the flash bus (e.g., an 8-bit serial bus) and executes the buffer management layer and the flash translation layer. The buffer management layer manages the buffer cache in the storage device. In many flash devices, the buffer cache is usually used for write buffering [16], [19] because a write operation is much slower than a read operation. Thus, in this work, the buffer cache is used as a write buffer. The FTL emulates the functionality of a normal block device, providing an interface between the upper layer and the flash chips. The FTL maintains a small internal buffer for use in internal operations such as garbage collection.

The flash chip is divided into several blocks, each of which consists of multiple pages. It has on-chip registers that are used as temporary storage for data transfers between the FTL buffer and the flash chip. The size of an on-chip register is the same as that of a page. A set of pages that share the same on-chip register is called a plane and there are usually 2-4 planes in a chip [4].

Writing a page from the buffer cache to the flash chip requires several data transfers. A page in the buffer cache is first moved to the internal buffer of the FTL through a system bus. Then, it is sent to the on-chip register of the flash chip via the flash bus. The page data is finally written to the target flash page. The time taken to write a page from the buffer cache to the flash chip is $(T_b + T_t + T_w)$, denoted by $T_{b\to f}$, where T_b is the time to move a page between the FTL and the buffer cache, T_t is the time to transfer a page through the flash bus, and T_w is the time to write a page to the flash chip from the on-chip register. The time taken to read a page from the flash chip to the FTL is $(T_r + T_t)$, where T_r is the time to read a page from the flash chip to the on-chip register. Note that if there is a host read request, data loaded into the FTL buffer is directly transferred to the host interface because the buffer cache is used as a write buffer [19]. Typical values of T_r, T_w, and T_t are 25, 200, and 100 μs [4], respectively. T_b is assumed to be 0 because data is transferred using a high-speed system bus.

A page migration also involves several data transfers. Suppose that the page p_k in Fig. 2 is moved to the page p_k^{new}. The page p_k is first moved to the on-chip register and then is sent to the FTL buffer. The data is returned to the on-chip register and written to the destination page p_k^{new}. The total time required to move the page p_k to its new location p_k^{new} is $(T_r + T_t) + (T_t + T_w)$, which is also denoted by $T_{f\to f}$. To reduce the cost of a page migration, most flash chips employ a specialized page copy operation, called a *copy-back* operation. With a copy-back operation, the page p_k loaded in the on-chip register is directly written to the destination page p_k^{new}. Thus, the time taken for a page migration is reduced to $(T_r + T_w)$ because data transfers between the processor and the flash chip are eliminated. Note that a copy-back operation can be used only when both the source and destination pages belong to the same plane.

For the FTL to be buffer-aware, it should be able to access the contents of a buffer cache. In our target device, the buffer management layer and the flash translation layer run on the same system, so it is easy to share information between two layers. Many flash devices such as embedded flash devices (e.g., CF cards) and solid-state drives satisfy our target architecture.

5 Buffer-Aware Block Merge

The proposed BAGC scheme consists of two schemes, one for a block merge step and the other for a victim block selection step, respectively. The first approach is to write data in a buffer cache to flash memory directly if a buffer cache has data for pages that are moved during a block merge. By doing so, useless page migrations can be eliminated. The second one is to decide a victim block during a victim selection step so that the potential benefits of buffer-aware block merges is to be maximized. We first introduce the buffer-aware block merge (BABM) scheme in this section and then explain the buffer-aware victim block selection scheme in Section 6.[1]

5.1 Buffer-Aware Block Merge Algorithm

There are two different types of block merges in the FAST FTL: a *full merge* and a *partial merge*. We develop the buffer-aware versions of these block merge operations.

Fig. 3 shows how a buffer-aware full merge works in detail. Suppose that a log block L_i is selected as a victim log block. The FTL identifies a set, $\mathbb{D}(L_i)$, of data blocks that are

```
Buffer_Aware_Full_Merge ($L_i$) {
    for $D_j \in \mathbb{D}(L_i)$ {
        get a new data block $D_j^{new}$ from a free block list;
        for $p_k \in D_j$ {
            if $p_k$ exists in a buffer cache {
                write $p_k$ in a buffer cache to $p_k^{new}$ in $D_j^{new}$;
                make $p_k$ in a buffer cache clean if it is dirty;
            } else {
                if $p_k$ is valid in $D_j$ {
                    move $p_k$ from $D_j$ to $p_k^{new}$ in $D_j^{new}$;
                } else { /* $p_k$ is invalid */
                    find a log block $L_j$ with a valid $p_k$;
                    move $p_k$ in $L_j$ to $p_k^{new}$ in $D_j^{new}$;
                    invalidate $p_k$ in $L_j$;
                }
            }
        }
        erase a data block $D_j$;
        insert an erased block $D_j$ into a free block list;
    }
    erase a log block $L_i$;
    insert an erased block $L_i$ into a free block list;
} /* end of function */
```

Fig. 3. A buffer-aware full merge algorithm.

involved in the block merge operation of L_i. A data block, D_j, whose updated pages are stored in L_i is included in $\mathbb{D}(L_i)$. For example, $\mathbb{D}(L_0)$ in Fig. 1 is $\{B_0, B_1\}$. For each page p_k in D_j, the FTL sees if p_k exists in a buffer cache. If a buffer cache has p_k and it is dirty (e.g., p_2 and p_4 in Fig. 1), it means that p_k in D_j is out-of-date. Copying p_k in D_j to a new data block D_j^{new} is, thus, useless. To prevent useless page migrations, the FTL writes p_k in a buffer cache to p_k^{new} in D_j^{new}, and then makes it clean. Note that if the cleaned page p_k is not updated before it is evicted from a buffer cache, a dirty page write for p_k is eliminated. If a buffer cache has p_k and it is clean (e.g., p_3 in Fig. 1), the FTL writes it to flash memory, instead of moving p_k in D_j to D_j^{new}, because a buffer cache already has the up-to-date data for p_k. If p_k is not in a buffer cache and is valid in D_j (e.g., p_5 in Fig. 1), it is moved from D_j to D_j^{new}. However, if p_k is not in a buffer cache and is invalid in D_j (e.g., p_0, p_1, p_6, and p_7 in Fig. 1), the FTL searches a log block L_j holding the valid version of p_k. Then, p_k in L_j is moved to D_j^{new} and is invalidated. Finally, the FTL erases D_j and L_i and inserts them to a free block list.

The partial merge is the different type of a merge operation, optimized for sequential writes [10]. In the partial merge, only one log block, called a sequential log block, is associated with one data block. The FTL performs the partial merge by copying valid pages in a data block to a sequential log block. The data block is then erased, and the sequential log block becomes the new data block. The difference between the partial merge and the buffer-aware partial merge is that the FTL copies pages to the sequential log block from a buffer cache if these pages exist in a buffer cache.

The buffer-aware full merge is more general and has a higher impact on FTL performance because the cost of the full merge is much higher than that of the partial merge. In this paper, therefore, we explain the buffer-aware full merge in detail. A detailed description of the buffer-aware partial merge can be found in [2].

5.2 The Effect of the Buffer-Aware Block Merge

By eliminating useless page migrations, the buffer-aware block merge performs a block merge operation at a lower cost than the buffer-unaware block merge. To understand the effect of the buffer-aware block merge on performance,

1. Our description is based on the FAST FTL [10]. However, it can be easily extended to other FTLs. For more detailed descriptions, see [2].

we first compare the buffer-unaware block merge cost and the buffer-aware block merge cost.

The buffer-unaware and buffer-aware block merge cost. In the *buffer-unaware* block merge, the block merge cost is determined by the number of pages that are moved between flash blocks during a block merge.

Definition 1. *Let $\mathbb{M}(L_i)$ be a set of flash pages that are moved during a block merge of a log block L_i and let $|\mathbb{M}(L_i)|$ be the number of pages in $\mathbb{M}(L_i)$. If the time taken to move a single page is $T_{f\rightarrow f}$, the buffer-unaware block merge cost of L_i, denoted by $C^{BU}_{merge}(L_i)$, is defined as follows:*

$$C^{BU}_{merge}(L_i) = |\mathbb{M}(L_i)| \cdot T_{f\rightarrow f}. \quad (1)$$

In the example of Fig. 1, $\mathbb{M}(L_0)$ is $\{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ and $|\mathbb{M}(L_0)|$ is 8. As pointed out in Section 4, $T_{f\rightarrow f}$ is $(T_r + T_t) + (T_t + T_w)$.

To eliminate useless page migrations, if a buffer cache has pages for flash pages that are moved during a block merge, the *buffer-aware* block merge directly writes them to flash memory. This not only eliminates useless page migrations, but also reduces the number of flash read operations because the FTL does not need to read pages that are already stored in a buffer cache. Therefore, the buffer-aware block merge cost is defined as follows:

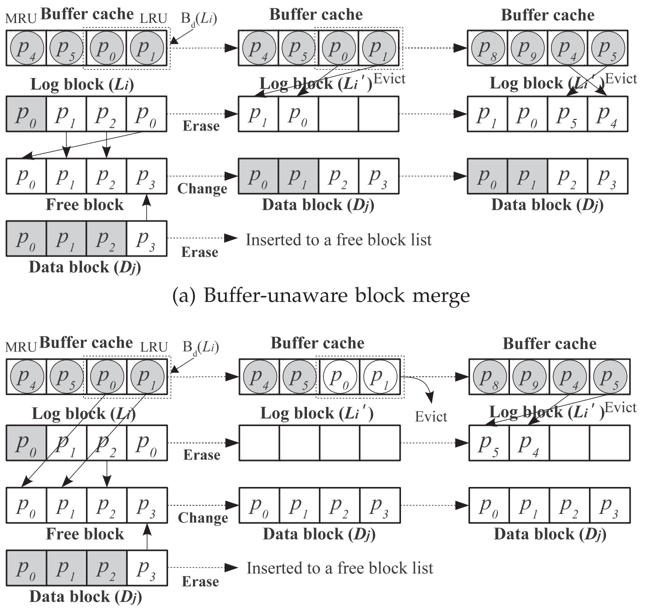
Definition 2. *Let $\mathbb{B}_d(L_i)$ be a set of flash pages that are moved during a block merge of L_i and have dirty data in a buffer cache. Let $\mathbb{B}_c(L_i)$ be a set of flash pages that have clean data in a buffer cache. $\mathbb{F}(L_i)$ is a set of flash pages that do not have any data in a buffer cache. $|\mathbb{B}_d(L_i)|$, $|\mathbb{B}_c(L_i)|$, and $|\mathbb{F}(L_i)|$ refer to the number of pages in each set. If the time taken to write a page to flash memory from a buffer cache is $T_{b\rightarrow f}$, the buffer-aware block merge cost of L_i, denoted by $C^{BA}_{merge}(L_i)$, is defined as follows:*

$$C^{BA}_{merge}(L_i) = (|\mathbb{B}_d(L_i)| + |\mathbb{B}_c(L_i)|) \cdot T_{b\rightarrow f} + |\mathbb{F}(L_i)| \cdot T_{f\rightarrow f}. \quad (2)$$

$T_{b\rightarrow f}$ is also denoted by $T_b + (T_t + T_w)$. In the example of Fig. 1, $\mathbb{B}_d(L_0)$ and $\mathbb{F}(L_0)$ are $\{p_2, p_4\}$ and $\{p_0, p_1, p_5, p_6, p_7\}$, respectively. $\mathbb{B}_c(L_0)$ is $\{p_3\}$. In (1) and (2), $|\mathbb{F}(L_i)| + |\mathbb{B}_d(L_i)| + |\mathbb{B}_c(L_i)|$ is $|\mathbb{M}(L_i)|$. Therefore, $C^{BA}_{merge}(L_i) \leq C^{BU}_{merge}(L_i)$ because $T_{b\rightarrow f} < T_{f\rightarrow f}$. In Fig. 1, three read operations for p_2, p_3, and p_4 are not required with the buffer-aware block merge.

As shown in (2), as a buffer cache has many pages for flash pages that are moved during a block merge, the buffer-aware block merge cost becomes smaller because many read operations can be avoided. This is true even for clean pages in a buffer cache. If there are clean pages in a buffer cache for flash pages to be moved (e.g., p_3 in Fig. 1), those clean pages can be directly written to flash memory, without additional read operations from flash memory. The benefit of buffer-aware block merges, however, mainly comes from dirty pages that are cleaned by buffer-aware block merges (e.g., p_2 and p_4 in Fig. 1). This is because these cleaned pages potentially reduce future dirty page writes and future block merges.

The potential benefits of the buffer-aware block merge. The buffer-aware block merge has two potential benefits that reduce the *future eviction cost* and the *future merge cost*.



(a) Buffer-unaware block merge

(b) Buffer-aware block merge

Fig. 4. A comparison of buffer-aware and buffer-unaware block merges.

As discussed above, the first one comes from the reduction in future page writes and the other one comes from the reduction in future block merge operations.

For better understanding of the potential benefits, we compare the behaviors of the buffer-unaware block merge and the buffer-aware block merge using examples shown in Fig. 4. This figure illustrates the contents of the buffer cache, the log block L_i, and the data block D_j, respectively. The numbers in circles as well as in rectangles represent a logical page number. The shaded circles indicate dirty pages while the shaded rectangles represent invalid pages. Initially, the buffer cache has four dirty pages, p_0, p_1, p_4, and p_5. There are also four pages in the log block L_i. One page p_0 out of the four log block pages is invalid because the new version of p_0 is in L_i. The data block D_j also has four pages, p_0, p_1, p_2, p_3, and three pages, p_0, p_1, p_2, are invalid because there are new pages in the log block.

- *The reduction in the future eviction cost.* Suppose that L_i is merged to make free pages. Here, $\mathbb{B}_d(L_i)$ is $\{p_0, p_1\}$; hence, $|\mathbb{B}_d(L_i)|$ is 2. $\mathbb{B}_c(L_i)$ is ϕ and $|\mathbb{B}_c(L_i)|$ is 0. If the buffer-unaware block merge is used as shown in Fig. 4a, two dirty pages, p_0 and p_1, remain dirty in the buffer cache, whereas they become clean with the buffer-aware block merge as depicted in Fig. 4b. After the block merge, the FTL obtains four new free pages. The new log block is denoted by L_i' to differentiate it from the old one. The former data block D_j becomes the free block and is inserted into the free block list. Finally, the former free block becomes the new data block D_j. Further suppose that p_0 and p_1 in $\mathbb{B}_d(L_i)$ are evicted from the buffer cache by a buffer replacement policy without any updates. In the buffer-unaware block merge, they must be written to the new log block L_i'. The eviction cost, $C^{BU}_{evict}(L_i)$, of the buffer-unaware block merge is thus $|\mathbb{B}_d(L_i)| \cdot T_{b\rightarrow f}$, which is $2 \cdot T_{b\rightarrow f}$ in Fig. 4. On the other hand, the eviction cost, $C^{BA}_{evict}(L_i)$, of the buffer-aware block merge is 0. This is because both p_0 and p_1 become clean when the old log block L_i is merged and there are no further updates on them before they are evicted from the buffer cache.[2] With the buffer-aware block merge, therefore, two page writes can be eliminated. This benefit that comes from the elimination of future page writes is called an *eviction-cost benefit*. The eviction-cost benefit is expressed as $C^{BU}_{evict}(L_i) - C^{BA}_{evict}(L_i)$, which is $2 \cdot T_{b\rightarrow f}$ in the example of Fig. 4. Note that the number of dirty page writes eliminated is denoted by $(C^{BU}_{evict}(L_i) - C^{BA}_{evict}(L_i))/T_{b\rightarrow f}$.

- *The reduction in the future merge cost.* The buffer-aware block merge requires a smaller number of future block merges because it writes fewer pages to log blocks over the buffer-unaware block merge. For example, consider the case in Fig. 4 where two pages, p_4 and p_5, in the buffer cache are evicted to L'_i by a buffer replacement policy. With the buffer-unaware block merge in Fig. 4a, the FTL should invoke a block merge operation because there are no free pages in L'_i. On the other hand, with the buffer-aware block merge in Fig. 4b, it is not necessary to perform a block merge because there are still two free pages. This benefit that comes from the reduction of future block merges is called a *merge-cost benefit*.

  To estimate the merge-cost benefit, the cost of a block merge induced by a single page write is to be estimated first. A block merge operation occurs when free pages in an empty log block have been entirely used up. If the number of pages per block is N_{ppb}, a block merge operation is invoked every N_{ppb} page writes to a log block. This means that if N_{ppb} dirty page writes are reduced, one block merge operation is eliminated. Assuming that the average cost of the buffer-aware block merge is given by C^{BA}_{avg}, the block merge cost eliminated by the reduction of one dirty page write is (C^{BA}_{avg}/N_{ppb}) on average. In this work, C^{BA}_{avg} is calculated using a moving average of recent block merge costs. The number of dirty page writes reduced by the buffer-aware block merge is $(C^{BU}_{evict}(L_i) - C^{BA}_{evict}(L_i))/T_{b\rightarrow f}$. The potential *merge-cost benefit* of L_i is, thus, expressed as $(C^{BU}_{evict}(L_i) - C^{BA}_{evict}(L_i)) \cdot \alpha$, where α is $(C^{BA}_{avg}/N_{ppb})/T_{b\rightarrow f}$.

  The following definition formalizes the potential benefits of the buffer-aware block merge.

Definition 3. *Let $C^{BU}_{evict}(L_i)$ be the eviction cost after a log block L_i is merged with the buffer-unaware block merge and let $C^{BA}_{evict}(L_i)$ be the eviction cost of L_i with the buffer-aware block merge. The eviction-cost benefit and the merge-cost benefit of L_i are $C^{BU}_{evict}(L_i) - C^{BA}_{evict}(L_i)$ and $(C^{BU}_{evict}(L_i) - C^{BA}_{evict}(L_i)) \cdot \alpha$, respectively. Thus, the total potential benefits of L_i, denoted by $B^{BA}_{benefit}(L_i)$, are defined as follows:*

$$B^{BA}_{benefit}(L_i) = \left(C^{BU}_{evict}(L_i) - C^{BA}_{evict}(L_i)\right) \cdot (1+\alpha). \tag{3}$$

Since $C^{BU}_{evict}(L_i)$ is larger than or equal to $C^{BA}_{evict}(L_i)$, $B^{BA}_{benefit}(L_i) \geq 0$. As explained in (1) and (2), $C^{BA}_{merge}(L_i)$ is also smaller than or equal to $C^{BU}_{merge}(L_i)$. Thus, the buffer-aware block merge at least performs better than the buffer-unaware block merge.

If pages cleaned by the buffer-aware block merge are updated later, there are no potential benefits because up-to-date data must be written to flash memory. The potential benefits, $B^{BA}_{benefit}(L_i)$, are thus changed depending on the update probability of pages in $\mathbb{B}_d(L_i)$ after the buffer-aware block merge. For example, in Fig. 4b, if p_0 and p_1 in $\mathbb{B}_d(L_i)$ are updated before they are evicted, $C^{BA}_{evict}(L_i)$ increases by $2 \cdot T_{b\rightarrow f}$ because the new version of data must be written to flash memory. In this case, $B^{BA}_{benefit}(L_i)$ is 0. That is, as many pages in $\mathbb{B}_d(L_i)$ are modified before they are evicted, the potential benefits become less significant. In Section 6, we discuss the way to maximize the potential benefits by considering the update probability of pages in a buffer cache.

5.3 The Effect of a Copy-Back Operation

To reduce the block merge cost, most flash chips support a specialized page migration operation, called a copy-back operation. As noted in Section 4, a copy-back operation eliminates expensive data transfers between a processor and a flash chip by using an on-chip register in a flash chip. Note that a copy-back operation can be performed for pages that share the same on-chip register.

With a copy-back operation, the cost of a single page migration is reduced to $T_r + T_w$ (i.e., 225 μs) from $T_r + 2 \cdot T_t + T_w$ (i.e., 425 μs). On the other hand, the cost of a single page migration of the buffer-aware block merge is $T_t + T_w$ (i.e., 300 μs) when the potential benefits, $B^{BA}_{benefit}(L_i)$, are assumed to be 0. In that case, a copy-back operation requires lower merge costs than the buffer-aware block merge. If $B^{BA}_{benefit}(L_i)$ is high enough, however, using the buffer-aware block merge is a better choice. To minimize the garbage collection overhead, therefore, we must carefully choose a proper operation for block merges depending on their benefits. In this section, we analyze the effect of a copy-back operation on the buffer-aware block merge. Our strategy that takes advantage of both of a copy-back operation and the buffer-aware block merge is presented in Section 6.

We first look at the case where only a copy-back operation is used without the buffer-aware block merge.

Definition 4. *Let $\mathbb{M}^p(L_i)$ be a subset of $\mathbb{M}(L_i)$, which only includes pages that can be copied back, and let $T_{p\rightarrow p}$ be the time taken to move a page using a copy-back operation. The buffer-unaware block merge cost of L_i, denoted by $C^{BU.CB}_{merge}(L_i)$, is derived from (1) as follows:*

$$C^{BU.CB}_{merge}(L_i) = |\mathbb{M}^p(L_i)| \cdot T_{p\rightarrow p} + (|\mathbb{M}(L_i)| - |\mathbb{M}^p(L_i)|) \cdot T_{f\rightarrow f}. \tag{4}$$

$T_{p\rightarrow p}$ in (4) is also denoted by $(T_r + T_w)$.

In the buffer-aware block merge, all the pages that can be copied back are moved using a copy-back operation because of its lower cost. For the pages that have dirty

2. If there are updates on the cleaned pages, our assumption that $C^{BA}_{evict}(L_i)$ is equal to 0 is no longer true. We discuss this issue at the end of this subsection.

pages in a buffer cache, however, dirty data in a buffer cache is directly written to flash memory, so as to take advantage of the potential benefits.

Definition 5. *Let $\mathbb{F}^p(L_i)$ be a subset of $\mathbb{F}(L_i)$ of a log block L_i, which only includes pages in flash memory that can be copied back. Let $\mathbb{B}_c^p(L_i)$ be a subset of $\mathbb{B}_c(L_i)$, which has clean pages in a buffer cache and has valid pages in flash memory that can be copied back. The buffer-aware block merge cost of L_i, denoted by $C_{merge}^{BA.CB}(L_i)$, is derived from (2) as follows:*

$$\begin{aligned} C_{merge}^{BA.CB}(L_i) = {} & \left(|\mathbb{F}^p(L_i)| + |\mathbb{B}_c^p(L_i)|\right) \cdot T_{p\to p} \\ & + \left(|\mathbb{F}(L_i)| - |\mathbb{F}^p(L_i)|\right) \cdot T_{f\to f} \\ & + \left(|\mathbb{B}_d(L_i)| + |\mathbb{B}_c(L_i)| - \left|\mathbb{B}_c^p(L_i)\right|\right) \cdot T_{b\to f}. \end{aligned} \tag{5}$$

When a copy-back operation is used, it is no longer true that the buffer-aware block merge is always more efficient than the buffer-unaware block merge. That is, in (4) and (5), $C_{merge}^{BA.CB}(L_i)$ can be larger than $C_{merge}^{BU.CB}(L_i)$. For example, suppose that all pages to be moved during a block merge can be copied back. (i.e., $|\mathbb{M}(L_i)| - |\mathbb{M}^p(L_i)| = 0$ in (4)). Further suppose that all those pages have dirty pages in a buffer cache (i.e., $|\mathbb{F}(L_i)| = |\mathbb{F}^p(L_i)| = |\mathbb{B}_c(L_i)| = |\mathbb{B}_c^p(L_i)| = 0$ in (5)) and there are no potential benefits (i.e., $B_{benefit}^{BA}(L_i) = 0$). In this case, $C_{merge}^{BA.CB}(L_i) > C_{merge}^{BU.CB}(L_i)$ because $|\mathbb{B}_d(L_i)| \cdot T_{b\to f} > |\mathbb{M}(L_i)| \cdot T_{p\to p}$. Thus, it is better to use the buffer-unaware block merge. However, if the potential benefits are high enough, the buffer-aware block merge can outperform the buffer-unaware block merge. Suppose that the potential benefits are the maximum in the example above (i.e., $C_{evict}^{BU}(L_i) - C_{evict}^{BA}(L_i) = |\mathbb{B}_d(L_i)| \cdot T_{b\to f}$). The FTL takes advantage of the potential benefits, $|\mathbb{B}_d(L_i)| \cdot T_{b\to f} \cdot (1+\alpha)$ according to $(C_{evict}^{BU}(L_i) - C_{evict}^{BA}(L_i)) \cdot (1+\alpha)$ in (3). These potential benefits are high enough to compensate for its high block merge cost.

6 Buffer-Aware Victim Block Selection

Selecting a victim block affects the performance of garbage collection in a significant fashion. Many previous studies, thus, have used a victim block selection policy to meet various design goals [5], [6]. In buffer-aware garbage collection, the cost of the buffer-aware block merge needs to be taken into account in selecting a victim block. In addition, the potential benefits of the buffer-aware block merge and the benefit of a copy-back operation should be considered. We first present an example that shows the need for better victim selection and then explain buffer-aware victim block selection in detail.

6.1 Example of Buffer-Aware Victim Selection

Consider the snapshot of the flash memory and the buffer cache in Fig. 5. The buffer cache has four dirty pages, p_0, p_1, p_8, and p_9. p_0 and p_1 are hot pages that are updated frequently while p_8 and p_9 are cold pages that are not updated before its eviction. There are two data blocks, B_0 and B_1, and two log blocks, L_0 and L_1. We assume that each block is composed of eight pages.

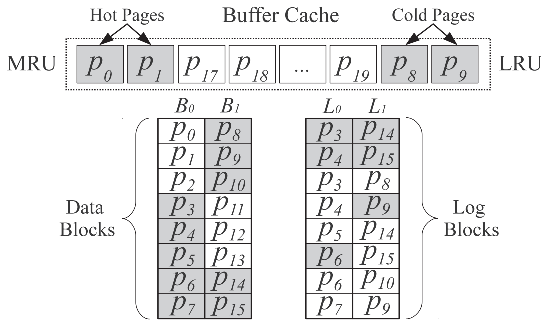


Fig. 5. An example of victim block selection.

Suppose that the log block L_0 is selected as a victim block and none of the pages can be moved using a copy-back operation. The FTL copies six pages, $p_2, \ldots, p_7$, from L_0 and B_0 to a new data block and copies p_0 and p_1 from the buffer cache. Therefore, the cost, $C_{merge}^{BA.CB}(L_0)$, of the buffer-aware block merge of L_0 becomes $6 \cdot T_{f\to f} + 2 \cdot T_{b\to f} = 6 \cdot 425\ \mu\text{s} + 2 \cdot 300\ \mu\text{s} = 3{,}150\ \mu\text{s}$. If the log block L_1 is selected instead as a victim block, the block merge cost, $C_{merge}^{BA.CB}(L_1)$, is 3,150 μs as well because two pages p_8 and p_9 are copied from the buffer cache.

However, when L_0 is chosen as a victim, moving the pages p_0 and p_1 from the buffer cache to the flash memory is wasted because they will be updated shortly and then be written to the flash memory. Thus, the benefit of the buffer-aware block merge, $B_{benefit}^{BA}(L_0)$, becomes 0. If L_1 is selected as a victim, p_8 and p_9 will not have to be written to the flash memory because it will remain clean. Therefore, $B_{benefit}^{BA}(L_1)$ is $2 \cdot T_{b\to f} \cdot (1+\alpha)$. In this case, even if the value of α is assumed to be 0 (i.e., the potential merge-cost benefit is 0), $B_{benefit}^{BA}(L_1)$ is 600 μs. That is, $B_{benefit}^{BA}(L_1)$ is at least larger than 600 μs. This benefit is potential in that it reduces the *future* write cost by eliminating the evictions of p_8 and p_9, but it is possible to estimate the garbage collection cost of L_1 when it is being merged by subtracting the potential benefits from the block merge cost. Thus, the garbage collection cost of L_i can be estimated as $3{,}150\ \mu\text{s} - 600\ \mu\text{s} = 2{,}550\ \mu\text{s}$. As a result, considering the potential benefits, it is better to choose L_1 as a victim block.

The situation becomes more interesting if some pages can be moved by a copy-back operation. For example, if eight pages, $p_0, \ldots, p_7$, can be copied back, then the cost, $C_{merge}^{BU.CB}(L_0)$, of the buffer-unaware block merge of L_0 is $8 \cdot T_{p\to p} = 1{,}800\ \mu\text{s}$. Therefore, even considering the benefit of the buffer-aware block merge, L_0 is a better choice. However, if all the pages $p_0, \ldots, p_{15}$ can be copied back, then $C_{merge}^{BA.CB}(L_1)$ reduces to $6 \cdot (T_{p\to p}) + 2 \cdot (T_{b\to f}) = 1{,}950\ \mu\text{s}$. Accounting for $B_{benefit}^{BA}(L_1)$, the cost of garbage collection is $1{,}950\ \mu\text{s} - 600\ \mu\text{s} = 1{,}350\ \mu\text{s}$.

This example illustrates that both the potential benefits of the buffer-aware block and the benefit of a copy-back operation are significant factors that determine the garbage collection cost. Thus, we should carefully consider them in deciding a victim log block.

6.2 Victim Log Block Selection Algorithm

In BAGC, the garbage collection cost of a log block L_i is determined by 1) the block merge cost, 2) the potential benefits of the buffer-aware block merge, and 3) the benefit of a copy-back operation. Our victim selection strategy is to choose a log block whose garbage collection cost is the smallest among all available log blocks.

For each log block L_i, we first calculate the garbage collection costs for different types of merge operations. The garbage collection cost, $GC_{BA}(L_i)$, of L_i with the buffer-aware block merge is defined as follows:

$$GC_{BA}(L_i) = C_{merge}^{BA.CB}(L_i) + C_{erase}(L_i) - B_{benefit}^{BA}(L_i),$$
$$\text{where } C_{erase}(L_i) = (|\mathbb{D}(L_i)| + 1) \cdot T_e, \tag{6}$$

where T_e is the time taken to erase a block and T_e is 1.5 ms according to [4]. $C_{erase}(L_i)$ is the time spent to erase all the blocks involved in garbage collection. $C_{merge}^{BA.CB}(L_i) + C_{erase}(L_i)$ is the cost for reclaiming L_i. This cost is compensated by the potential benefits, $B_{benefit}^{BA}(L_i)$. In the example of Fig. 5, if the pages $p_0, \ldots, p_{15}$ can be copied back, $GC_{BA}(L_0)$ and $GC_{BA}(L_1)$ are 4,950 μs and 4,350 μs, respectively, because $C_{erase}(L_0) = C_{erase}(L_1) = 2 \cdot T_e = 3{,}000$ μs.

The garbage collection cost, $GC_{BU}(L_i)$, of L_i with the buffer-unaware block merge is defined as follows:

$$GC_{BU}(L_i) = C_{\text{merge}}^{BU.CB}(L_i) + C_{\text{erase}}(L_i). \tag{7}$$

In Fig. 5, $GC_{BU}(L_0) = GC_{BU}(L_1) = 4{,}800$ μs.

For each log block L_i, the block merge operation that requires the smaller cost is chosen for garbage collection. Therefore, the garbage collection cost, $GC(L_i)$, of L_i is formally expressed as follows:

$$GC(L_i) = \min(GC_{BA}(L_i), GC_{BU}(L_i)). \tag{8}$$

In the example of Fig. 5, for the log block L_0, the buffer-unaware block merge requires the smallest merge cost because $GC_{BU}(L_0) = 4{,}800\ \mu s < GC_{BA}(L_0) = 4{,}950\ \mu s$. On the other hand, for the log block L_1, the buffer-aware block merge requires the smallest merge cost because $GC_{BA}(L_1) = 4{,}350\ \mu s < GC_{BU}(L_1) = 4{,}800\ \mu s$. Finally, the log block with the smallest merge cost is chosen as a victim block. In the example of Fig. 5, L_1 with the buffer-aware block merge is the best choice.

Fig. 6 describes the buffer-aware victim block selection algorithm. For all log blocks, $GC(L_i)$ is first obtained. The total number of log blocks in the FTL is denoted by N_{lb}. The FTL selects the log block with the smallest $GC(L_i)$ as a victim block using (6) and (7). The FTL then performs a block merge for the victim block using the corresponding block merge operation. `Buffer_Unaware_Full_Merge`(L_i) in Fig. 6 is the original block merge of the FAST FTL [10], except that it uses a copy-back operation. `Buffer_Aware_Full_Merge`(L_i) is the same as the algorithm in Fig. 3. Since a copy-back operation is enabled, in lines 10 and 13 of Fig. 3, p_k is copied back to p_k^{new} if they belong to the same plane.

To calculate the garbage collection cost, $GC(L_i)$, of L_i, the potential benefits of the buffer-aware block merge as well as the benefit of a copy-back operation should be accurately estimated. First, to know how a copy-back operation affects the cost of garbage collection, the values of $|\mathbb{M}^p(L_i)|$, $|\mathbb{F}^p(L_i)|$, and $|\mathbb{B}_c^p(L_i)|$ in (4) and (5) should be known. These values can be easily obtained by looking at

```
Buffer_Aware_Garbage_Collection () {
    ($L_i$, $op\_type_i$) := Buffer_Aware_Victim_Selection($N_{lb}$); /* $N_{lb}$ is the
      number of log blocks available */
    if ($op\_type_i$ is buffer-unaware block merge)
        Buffer_Unaware_Full_Merge($L_i$); /* original full merge */
    else
        Buffer_Aware_Full_Merge($L_i$); /* buffer-aware full merge */
}
Buffer_Aware_Victim_Selection ($N_{lb}$) {
    $cost_{min}$ := $\infty$;
    $L_{min}$ := null;
    $op\_type_{min}$ := null;
    for ($i$ := 0; $i$ <$N_{lb}$; $i$++) { /* check all available log blocks */
        ($cost$, $op\_type$) := Get_Garbage_Collection_Cost($L_i$);
        if ($cost$ < $cost_{min}$) {
            $cost_{min}$ := $cost$;
            $L_{min}$ := $L_i$;
            $op\_type_{min}$ := $op\_type$;
        }
    }
    return ($L_{min}$, $op\_type_{min}$);
}
Calculate_Garbage_Collection_Cost ($L_i$) {
    $op\_type$ := null;
    /* get the GC cost with the potential benefits of $L_i$ */
    $cost_{BA}$ := $GC_{BA}(L_i)$; /* Eq. (6) */
    /* get the GC cost with the benefit of a copy-back operation of $L_i$ */
    $cost_{BU}$ := $GC_{BU}(L_i)$; /* Eq. (7) */
    /* choose the smaller one */
    if ($cost_{BA}$ < $cost_{BU}$) {
        $op\_type$ := buffer-aware block merge;
        return ($cost_{BA}$, $op\_type$);
    } else {
        $op\_type$ := buffer-unaware block merge;
        return ($cost_{BU}$, $op\_type$);
    }
}
```

Fig. 6. A buffer-aware victim block selection algorithm.

source planes where source pages are placed and destination planes where source pages will be written. Second, the value of $(C_{evict}^{BU}(L_i) - C_{evict}^{BA}(L_i))$ in (3) should be estimated to know the potential benefits of the buffer-aware block merge. We know that $C_{evict}^{BU}(L_i)$ is $|\mathbb{B}_d(L_i)| \cdot T_{b\to f}$. However, $C_{evict}^{BA}(L_i)$ is not easily estimated because it depends on the *future* update probability of pages in $|\mathbb{B}_d(L_i)|$.

6.3 Locality-Aware Potential Benefit Prediction

The estimation of the potential benefits. To estimate the potential benefits, we should know in advance how many pages in $\mathbb{B}_d(L_i)$ are updated before they are evicted from a buffer cache. Suppose that $\mathbb{B}_d(L_i)$ is divided into two subsets, $\mathbb{B}_d^{tbe}(L_i)$ and $\mathbb{B}_d^{tbu}(L_i)$, where $\mathbb{B}_d^{tbe}(L_i)$ is a set of pages to be evicted from a buffer cache without further updates after the buffer-aware block merge, and $\mathbb{B}_d^{tbu}(L_i)$ is a set of pages to be updated and to be dirty again before their eviction. For instance, in Fig. 5, $\mathbb{B}_d^{tbe}(L_i) = \{p_8\}$ and $\mathbb{B}_d^{tbu}(L_i) = \{p_0\}$.

If pages in a log block L_i are mostly associated with dirty pages in $\mathbb{B}_d^{tbu}(L_i)$, the potential benefits, $C_{benefit}^{BA}(L_i)$, are close to 0 because $C_{evict}^{BA}(L_i)$ approaches $C_{evict}^{BU}(L_i)$. Thus, $C_{evict}^{BA}(L_i)$ can be written as follows:

$$C_{evict}^{BA}(L_i) = \left|\mathbb{B}_d^{tbu}(L_i)\right| \cdot T_{b\to f}. \tag{9}$$

Since $|\mathbb{B}_d^{tbu}(L_i)|$ depends on the frequency with which each page will be updated, the value of $|\mathbb{B}_d^{tbu}(L_i)|$ can be approximated as follows:

$$\left|\mathbb{B}_d^{tbu}(L_i)\right| \simeq \sum_{p_k \in \mathbb{B}_d(L_i)} P(\mathcal{U}_{p_k}), \tag{10}$$

where $\mathcal{U}_{p_k}$ is the event that a page p_k is updated before its eviction, and $P(\mathcal{U}_{p_k})$ is the probability that $\mathcal{U}_{p_k}$ occurs.

However, the exact value of $P(\mathcal{U}_{p_k})$ is not available at garbage collection time because the future behavior of a

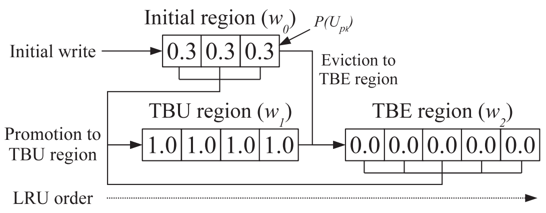


Fig. 7. A three-region LRU buffer.

buffer cache is unknown. We therefore predict $P(\mathcal{U}_{p_k})$ by exploiting temporal locality of I/O references. It is probable that a recently updated page will be updated again. Therefore, if p_k is written more frequently in recent times, we assign a larger value to $P(\mathcal{U}_{p_k})$.

Three-region LRU buffer management. To determine temporal locality of each page p_k, we employ a novel buffer architecture called a three-region LRU buffer, which is depicted in Fig. 7. The three-region LRU buffer is divided into three regions: an initial region, a TBU (to-be-updated) region, and a TBE (to-be-evicted) region. Each region is a normal buffer cache, which is managed by an LRU replacement policy. When data is first written to a buffer cache, it is placed in the initial region. The initial region is used to separate write-once data, which is not updated after being written to a buffer cache, from frequently updated data. If data is subsequently updated, it is promoted to the TBU region. Otherwise, it is evicted to the TBE region in the order of its arrival in the initial region. If data is updated in the TBE region, it is promoted to the TBU region. If data is not updated for a long time, it is evicted to TBU from TBE.

The benefit of using the three-region LRU buffer is that it helps us to categorize pages in a buffer cache depending on their update probabilities. For example, if a page is placed in the TBU region, it will be updated soon. Thus, assigning a higher update probability to that page is probable. Through a series of experiments, we found that assigning an extreme probability value to each region is generally useful. That is, we assign $P(\mathcal{U}_{p_k})$ of 1.0 to pages in the TBU region. All pages in the TBE region have $P(\mathcal{U}_{p_k})$ of 0.0. The initial region may contain both hot and cold pages. Thus, $P(\mathcal{U}_{p_k})$ for the initial region is dynamically adjusted by the ratio of the TBU region size to a buffer cache size because it indicates a proportion of hot pages in a buffer cache.

The size of each region has an effect on deciding $|\mathbb{B}_d^{tbu}(L_i)|$. For example, if the TBU region size is much larger than the actual number of hot pages in a buffer cache, the TBU region holds many cold pages and all of them are regarded as hot pages. Therefore, the value of $|\mathbb{B}_d^{tbu}(L_i)|$ becomes high even though there are many cold pages in the TBU region. To avoid this problem, the three-region LRU buffer adjusts the size of each region depending on I/O access patterns, by considering the rate at which pages move between three regions.

Fig. 8 shows a finite state machine (FSM) corresponding to a page in a buffer cache. This FSM has six states (`Init`, $\texttt{TBE}_0$, $\texttt{TBE}_1$, $\texttt{TBU}_0$, $\texttt{TBU}_1$, and `Evicted`) and two inputs (`update` and `evict`). The `update` input means that a page is to be updated in response to a write request. The `evict` input means that a page is to be evicted from a region by the LRU replacement policy. Whenever the four transitions (i.e., $\texttt{TBU}_0 \rightarrow \texttt{TBE}_1$,

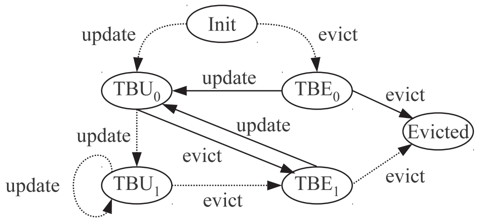


Fig. 8. A state transition diagram of the three-region buffer.

$\texttt{TBE}_1 \rightarrow \texttt{TBU}_0$, $\texttt{TBE}_0 \rightarrow \texttt{TBU}_0$, and $\texttt{TBE}_0 \rightarrow \texttt{Evict}$) of FSM occur, the size of each region is adjusted. First, if there are many pages whose states change from $\texttt{TBU}_0$ to $\texttt{TBE}_1$ (i.e., $\texttt{TBU}_0 \rightarrow \texttt{TBE}_1$), it means that there are many pages with a low update probability in the TBU region. Thus, we need to decrease the TBU region size, w_1, and increase the TBE region size, w_2. Second, if there are many transitions from $\texttt{TBE}_1$ to $\texttt{TBU}_0$ (i.e., $\texttt{TBE}_1 \rightarrow \texttt{TBU}_0$), we increase w_1 and decrease w_2 because many pages with a high update probability exist in the TBE region. Third, if there are many transitions from $\texttt{TBE}_0$ to $\texttt{TBU}_0$ (i.e., $\texttt{TBE}_0 \rightarrow \texttt{TBU}_0$), it means that many pages that had to be sent from the initial region to the TBU region were actually sent to the TBE region. Therefore, we increase the Initial region size, w_0, and decrease w_2. Finally, if many pages are evicted from a buffer cache in the $\texttt{TBE}_0$ state (i.e., $\texttt{TBE}_0 \rightarrow \texttt{Evict}$), we decrease w_0 and increase w_2 to reduce the time that a page stays in the initial region.

The size of each region in the three-region LRU buffer is initially set to the same and is adjusted at page granularity whenever the transition occurs. For example, when a page is evicted from $\texttt{TBU}_0$ to $\texttt{TBE}_1$, the TBU region size decreases by a page, whereas the TBE region size increases by a page.

6.4 Reducing Computational Complexity

Whenever the FTL selects a victim log block, it is required to compute the value of $|\mathbb{B}_d^{tbu}(L_i)|$ for each log block L_i, so as to estimate the potential benefits. For all the pages to be moved during the block merge of each log block L_i, BAVBS needs to examine the region where each page is placed. To reduce the computational overhead required for obtaining $|\mathbb{B}_d^{tbu}(L_i)|$, the three-region LRU buffer is managed at the granularity of a block, instead of a page. That is, all the pages belonging to the same logical block stay in the same region of a buffer cache and move together when they are promoted to or evicted from another region. This means that all the pages in the same logical block have the same $P(\mathcal{U}_{p_k})$. Therefore, the value of $|\mathbb{B}_d^{tbu}(L_i)|$ can be obtained at a lower cost by using the number of dirty pages in a buffer cache for logical blocks (which are involved in the block merge of a log block L_i) and the regions where those logical blocks are placed. This approach sacrifices the accuracy of detecting the update probability of an individual page, but it reduces the computational overhead greatly.[3]

If a processor and a memory module in a storage device have limited performance, the computational overhead could be high even if the three-region LRU buffer is managed at the granularity of a block. To reduce this computational overhead, we propose a limited version of

3. A more detailed analysis of the computational overhead of the three-region LRU buffer can be found in Appendix (which is available from our on-line supplemental material.)

TABLE 1
Descriptions of Benchmark Programs

Trace	Description	Written Data
Bonnie++	Create and delete files in sequential and random orders, while performing different types of file system operations.	0.78 GB
Tiobench	Create 8 × 120 MB files from eight threads. Each thread generates 4K random writes.	1.11 GB
Postmark	Create 30 K files whose sizes are 4-16 KB, while generating 200 K transactions.	4.45 GB
Iozone	Execute writes/re-writes and reads/re-reads on a 1 GB file with 2 KB records. The stripped access was disabled and the flush was enabled.	3.46 GB
PC	Collect I/O activities from a desktop PC for 1 day while running several applications such as document editors, web browsers, messenger, and photoshop.	4.78 GB
Mobile	Create and delete several multimedia files including movies, pictures, and MP3s.	4.42 GB

BAGC which examines a limited number, N_{lb}^{limit}, of log blocks starting from the least-recently written (LRW) log block. The intuition behind this approach is that log blocks close to the LRW log block tend to have a relatively low merge cost because they are likely to have few valid pages. Since the goal of BAVBS is to choose a victim log block with both the low block merge cost and the high potential benefits, it is reasonable to estimate the potential benefits of those log blocks if it is difficult to examine all available log blocks. There is a tradeoff between performance and computational overhead in the limited version of BAGC. If the value of N_{lb}^{limit} is set to 1, BAVBS works like BABM and there is a negligible overhead in choosing a victim. As N_{lb}^{limit} approaches N_{lb}, better performance is achieved at a cost of computation.

7 Experimental Results

7.1 Experimental Setup

To evaluate the performance of BAGC, we developed a trace-driven simulator that models a flash device depicted in Fig. 2. We compared BAGC with three buffer management schemes, BLRU [15], [16], FAB [15], and BPLRU [16], running on top of various FTLs, including BAST [9], FAST [10], and SuperBlock [11]. For BAGC, the block merge and victim selection modules were modified as described in Sections 5 and 6. The three-region LRU buffer was managed at the granularity of a block because the page-level buffer management requires too much computation.

In our evaluation, a buffer cache was used as a write buffer. Using a write buffer in a storage device inevitably lowers data reliability because all dirty data in a buffer cache will be lost when a power failure occurs or the system suddenly stops working. To prevent data loss from such exceptional events and provide a high degree of data reliability, any dirty pages staying in a buffer cache for more than 30 seconds were flushed to flash memory.

The flash parameters were based on Samsung's NAND flash memory with 64 2-KB pages in a block [4]. The page read time, the page write time, and the block erasure time were set to 25 μs, 200 μs, and 1.5 ms, respectively, and the page transfer time through the flash bus was 100 μs. The value of α was dynamically decided by taking the average block merge cost using a moving average algorithm. The value of T_b was assumed to be 0. The values of $P(\mathcal{U}_{p_k})$ for the TBU and TBE regions were set to 1.0 and 0.0, respectively. The value of $P(\mathcal{U}_{p_k})$ for the initial region was dynamically adjusted as described in Section 6.3. The flash simulator was configured with nine flash chips, each of which is 1 GB with four planes. For evaluation with aged devices, all blocks except for log blocks were initially filled with valid data.

TABLE 2
I/O Time of *BUBM*, *BABM*, and *BABM+BAVBS*

Trace	I/O Times (seconds)		
	BUBM	*BABM*	*BABM+BAVBS*
Bonnie++	160	154	142
Tiobench	391	347	316
Postmark	4,617	3,180	2,688
Iozone	1,434	1,214	1,063
PC	1,635	1,513	1,394
Mobile	788	763	763

7.2 Benchmarks

Our evaluation was conducted with six benchmark programs, which are listed in Table 1. We used four well-known benchmarks, Bonnie++, Tiobench, Postmark, and Iozone, to assess performance under I/O intensive environments where I/O performance really matters. We also evaluated BAGC with a real-world trace recorded from real-user activities on a desktop PC. Finally, we used the Mobile trace that captures the workload of a portable media player, which is one of the representative mobile applications. Microsoft Windows XP was used for our trace collection, and all the traces were extracted from a disk driver using the Diskmon utility [20].

The benchmark programs have distinctive characteristics in terms of data locality and I/O reference patterns. Bonnie++ and Postmark are small-file-oriented and metadata-intensive workloads. Iozone is designed to measure streaming performance for large files, but it also incurs many updates to metadata and data. Therefore, they exhibit relatively high locality. Tiobench incurs many random writes, so it exhibits quite low locality. Mobile incurs many sequential writes for multimedia files with small metadata updates. PC is a real-life trace, containing many sequential writes for large files, repetitive updates for small files, and many random writes. PC exhibits higher locality than Tiobench and Mobile.

7.3 Performance Analysis

In this section, we evaluate two main techniques of the BAGC scheme, BABM and BAVBS, so as to understand their effects on performance. We use the FAST FTL as our default FTL scheme and employ the three-region LRU buffer for buffer management. The buffer cache size is set to 32 MB and 512 log blocks are used. The evaluation results with other FTL and buffer management schemes are given in Section 7.4. To know the maximum performance that BAGC can achieve, N_{lb}^{limit} is set to 512. We evaluate the limited version of BAGC in Section 7.6.

7.3.1 Overall Performance

Table 2 shows the I/O time under the different configurations. The I/O time is the total amount of time taken to perform all I/O operations, including page reads, page writes, and block erasures. We analyze the performance of the following three schemes: *BUBM*, *BABM*, and

TABLE 3
A Comparison of Dirty Page Writes of *BUBM*, *BABM*, and *BABM+BAVBS*

Trace	*Dirty*			*Clean*	
	BUBM	*BABM*	*BABM+ BAVBS*	*BABM*	*BABM+ BAVBS*
Bonnie++	65,625	56,335	56,655	16,338	37,230
Tiobench	89,297	76,670	76,241	13,667	17,008
Postmark	1,889,516	1,348,280	1,260,648	560,755	647,634
Iozone	652,461	580,503	498,841	152,774	288,725
PC	710,036	625,436	572,872	88,978	150,304
Mobile	48,342	9,230	9,230	36,566	36,566

This table also shows the number of pages cleaned by buffer-aware block merges.

BABM+BAVBS. *BUBM* employs the buffer-unaware block merge (BUBM) scheme and selects a victim block using the round-robin policy [10], which is a default victim selection policy used in the FAST FTL. *BABM* employs the buffer-aware block merge scheme with the round-robin victim selection policy. *BABM+BAVBS* is the same as *BABM* except that it uses the buffer-aware victim block selection scheme.

It is clear from Table 2 that *BABM+BAVBS* shows the best performance among all the schemes evaluated. *BABM* improves performance by 12.1 percent, on average, over *BUBM*. This benefit comes from the elimination of unnecessary page migrations during block merges. *BABM+-BAVBS* chooses a victim log block to maximize the benefit of the buffer-aware block merge, so it further reduces I/O time by 19.4 percent, on average, over *BUBM*.

The performance of buffer-aware garbage collection varies from one benchmark to another. Thus, we investigate some of the factors that might affect performance. We first look at how many page writes are removed by eliminating unnecessary page migrations and then explain how this affects the garbage collection cost.

7.3.2 Reduction in Page Writes

Since a buffer cache is used as a write buffer, all the pages that are written to a buffer cache are initially dirty. These dirty pages can be cleaned by buffer-aware block merges. If there are no further updates on them, they are evicted as clean pages from a buffer cache, and thus the number of dirty page writes is reduced. As expected, as many useless page migrations are eliminated, more clean pages are evicted from a buffer cache.

Table 3 compares the numbers of dirty page writes according to three different schemes, which are denoted by *Dirty*. We only consider dirty pages written to random log blocks because they incur expensive full merges. For *BABM* and *BABM+BAVBS*, we show the numbers of pages that become clean by buffer-aware block merges and then be evicted from a buffer cache without further updates. These numbers are denoted by *Clean* in Table 3.

As shown in Table 3, *BABM* reduces the number of dirty page writes by 11-80 percent over *BUBM*. *BABM+BAVBS* further reduces the number of dirty page writes by 0.6-15 percent over *BABM* because it eliminates more useless page migrations, thereby increasing clean pages evicted from a buffer cache. The increase in the number of the clean pages with *BABM+BAVBS* also shows that the proposed BAVBS technique effectively chooses a victim log block that has cold pages in a buffer cache. In our observation, on average, only 1.3 percent of the clean pages are updated before being evicted from a buffer cache.

TABLE 4
A Percentage of Block Merges by Type, Normalized to *BUBM*

Trace	*BABM*			*BABM+BAVBS*		
	Switch	Partial	Full	Switch	Partial	Full
Bonnie++	97.9%	115.6%	71.7%	91.8%	111.0%	72.7%
Tiobench	100.0%	89.5%	78.1%	100.0%	86.5%	76.9%
Postmark	99.9%	82.1%	70.8%	100.2%	81.9%	66.1%
Iozone	95.0%	97.0%	88.3%	90.2%	90.7%	75.2%
PC	100.0%	99.0%	87.5%	100.1%	95.4%	79.7%
Mobile	100.1%	100.1%	0%	100.1%	100.1%	0%

In the cases of Bonnie++ and Tiobench, dirty page writes do not decrease as much as the increase in clean pages. This is because many dirty pages that are to be written to a sequential log block are changed to clean pages by buffer-aware block merges. Thus, the number of dirty pages written to random log blocks is not changed greatly. In the case of Mobile, a large number of dirty pages are eliminated by *BABM* and *BABM+BAVBS*, but its performance is not greatly improved as shown in Table 2. In the Mobile benchmark, almost all write requests are sent to a sequential log block, so the garbage collection overhead is very low. This is the reason why the effect of buffer-aware garbage collection on performance is trivial.

7.3.3 Reduction in Garbage Collection Overhead

The buffer-aware garbage collection technique not only reduces the number of dirty page writes to flash memory, but also decreases the number of block merge operations. Table 4 shows the percentage of block merge operations by type, which are normalized to *BUBM*. *BABM* and *BABM+BAVBS* eliminate lots of full merges and partial merges that require many page migrations, whereas the proportion of switch merges is not changed greatly.

For Bonnie++, full merge operations performed with *BABM+BAVBS* are slightly increased in comparison to those with *BABM*. However, the number of pages moved during full merges is reduced to 84 percent of *BABM*, and thus the garbage collection overhead is accordingly lowered. In the case of Mobile, full merges are not observed with *BABM* and *BABM+BAVBS*. The Mobile benchmark writes only a small number of pages to random log blocks, and many of them become clean in a buffer cache by buffer-aware partial merges. As a result, a full merge operation is not invoked because random log blocks are not fully filled with data.

7.3.4 Reduction in Block Erasure Operations

Fig. 9 shows the number of block erasure operations, normalized to *BUBM*. *BABM+BAVBS* shows the smallest

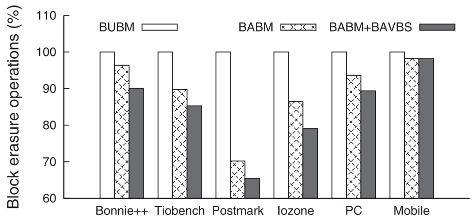


Fig. 9. Block erasure operations, normalized to *BUBM*.

TABLE 5
An Attribution of Performance Improvement to Processes within *BABM+BAVBS*

Trace	Utilizing a copy-back operation	Removing unnecessary migrations
Bonnie++	3.0%	97.0%
Tiobench	29.1%	70.9%
Postmark	22.9%	77.1%
Iozone	15.5%	84.5%
PC	23.9%	76.1%
Mobile	1.2%	98.8%

erasure operations among all the schemes. This is because *BABM+BAVBS* eliminates a large number of useless page migrations during garbage collection, decreasing the number of blocks involved in block merges.

7.3.5 Impact of a Copy-Back Operation

We finally analyze the effect of a copy-back operation on performance. Table 5 attributes the performance improvement achieved by *BABM+BAVBS* over *BUBM* to the elimination of unnecessary page migrations and the use of a copy-back operation. We can see that a copy-back operation improves the overall performance by 16 percent, on average. In the cases of Bonnie++ and Mobile, there are not many chances to exploit a copy-back operation when moving pages, and thus the benefit of using copy-back operations is limited.

We evaluate the performance of *BABM+BAVBS* when it chooses a victim log block without consideration of the benefit of a copy-back operation. In our observation, the number of useless page migrations eliminated increases by up to 5 percent because it always chooses a log block with many dirty pages in a buffer cache. The overall performance, however, is reduced by up to 10 percent due to a high page transfer time.

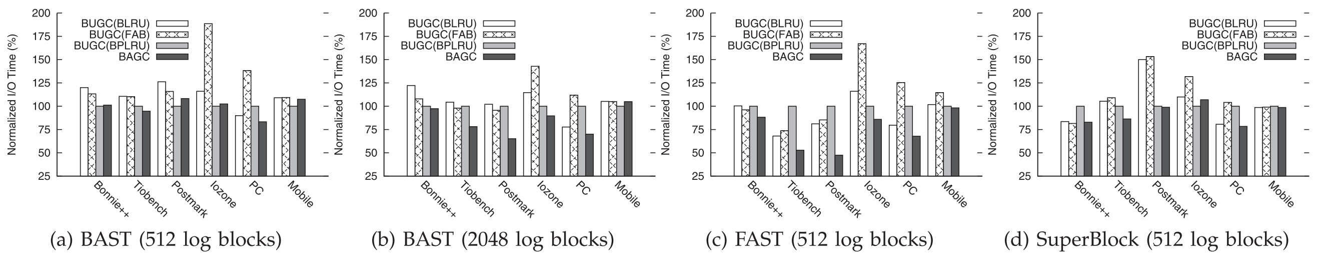
7.4 Performance Comparisons with Existing Buffer Management and FTL Schemes

After the evaluation of several subtechniques that compose BAGC, we compare the performance of BAGC with three buffer management schemes, BLRU, BPLRU, FAB running under three FTL schemes, BAST, FAST, and SuperBlock. The buffer cache size is set to 32 MB and 512 log blocks are used. N_{lb}^{limit} is set to 512.

Fig. 10 shows our evaluation results. In this figure, the block-level LRU scheme (BLRU), the BPLRU scheme, and the FAB scheme are referred to as *BUGC(BLRU)*, *BUGC(BPLRU)*, and *BUGC(FAB)*, respectively. All those schemes do not use any buffer-aware techniques, so they use the original FTL schemes. The proposed buffer-aware garbage collection scheme is the one labeled as *BAGC* in Fig. 10. For *BAGC*, the underlying FTL schemes are modified to support buffer-aware block merge and buffer-aware victim block selection operations. *BAGC* uses the three-region LRU buffer scheme for buffer management because it must know the update probabilities of pages in a buffer cache to determine a victim log block. Note that *BAGC* is the same configuration to *BABM+BAVBS* in Section 7.3. The results shown in Fig. 10 are normalized to *BUGC(BPLRU)*. *BUGC(BPLRU)* always shows the *same* performance for each benchmark, regardless of the underlying FTL algorithm, because of the page padding technique [17].

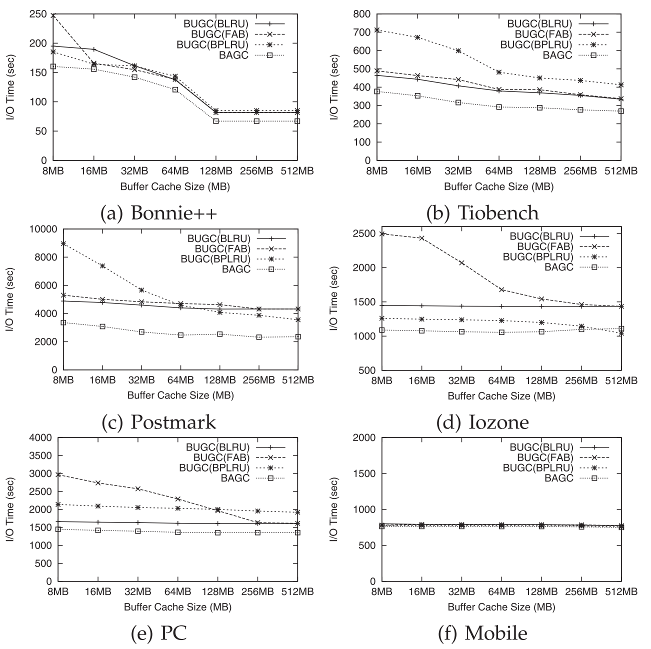
In BAST with 512 log blocks, *BAGC* has 11 and 21 percent shorter I/O time than *BUGC(BLRU)* and *BUGC(FAB)*, on average, respectively. However, the performance of *BAGC* is somewhat worse than that of *BUGC(BPLRU)*. This problem is due to the log block thrashing problem [10], which is typically observed when available log blocks are smaller than a working set size. Unlike other schemes, *BUGC(BPLRU)* is unaffected by the block thrashing problem because it does not utilize log blocks for its page padding technique [16]. This is the reason why *BUGC(BPLRU)* shows better performance than *BAGC*. To examine what happens with enough log blocks, we evaluate performance with 2,048 log blocks. As expected, *BAGC* exhibits the best performance because the block thrashing problem disappears; it outperforms *BUGC(BLRU)*, *BUGC(FAB)*, and *BUGC(BPLRU)* by 20, 31, and 23 percent, respectively. The FAST FTL is designed to prevent the log block thrashing problem by increasing associativity between log blocks and data blocks [10]. Therefore, with 512 log blocks, *BAGC* achieves 21, 40, and 43 percent shorter I/O time than *BUGC(BLRU)*, *BUGC(BPLRU)*, and *BUGC(FAB)*, on average, respectively. Using the SuperBlock FTL, *BAGC* runs 11, 29, and 6 percent faster, on average, than *BUGC(BLRU)*, *BUGC(FAB)*, and *BUGC(BPLRU)*, respectively, but the performance improvement is less significant than the change in BAST and FAST. This is due to the inflexibility of victim selection in SuperBlock FTL. Unlike BAST and FAST that choose the most cost-effective victim among all available log blocks, SuperBlock allows us to select a victim from a small number of log blocks (e.g., eight blocks) within a superblock [11].

An important observation shown in Fig. 10 is that with *BAGC*, FAST exhibits the best performance among all the FTL schemes evaluated; it outperforms BAST and SuperBlock by 19 and 37 percent, respectively, on average. Therefore, we can conclude that the combination of the FAST FTL and *BAGC* is the most effective way to minimize the garbage collection cost.



(a) BAST (512 log blocks) (b) BAST (2048 log blocks) (c) FAST (512 log blocks) (d) SuperBlock (512 log blocks)

Fig. 10. Performance comparisons of buffer management and FTL schemes. These results are normalized to BUGC+BPLRU.



(a) Bonnie++ (b) Tiobench (c) Postmark (d) Iozone (e) PC (f) Mobile

Fig. 11. I/O time against the buffer cache size.

7.5 The Effect of the Buffer Cache Size

Fig. 11 shows the performance of BAGC when a buffer cache size varies from 8 to 512 MB. We use the FAST FTL and the number of log blocks is 512. Overall, *BAGC* exhibits the best performance, regardless of a buffer cache size. It improves the effective hit ratio of writes by eliminating some writes to flash memory, and thus its performance is maintained when the buffer cache is small. For example, in Postmark, *BAGC* achieves the write hit ratio of 2.1 percent with the buffer cache of 8 MB, but it requires 18 percent fewer page writes than other schemes.

BUGC(FAB) gives the worst performance on some benchmarks (e.g., Bonnie++, Iozone, and PC) when the size of a buffer cache is small. *BUGC(FAB)* selects the block with the largest number of valid pages in a buffer cache as a victim block. However, it often evicts blocks with high locality, thus increasing the number of writes to flash memory [17]. *BUGC(BPLRU)* performs poorly on some benchmarks (e.g., Tiobench, Postmark, and PC) where the utilization of a block (i.e., the number of pages belonging to a block) evicted from a buffer cache is relatively low. Due to its page padding technique [17], poorly utilized blocks incur many extra I/Os to flash memory, increasing the garbage collection overhead. Both *BAGC* and *BLRU* are free from these side effects because they use the pure block-level LRU policy.

In some benchmarks (e.g., including Iozone, PC, and Mobile), the I/O performance is not greatly improved even when the buffer cache size is relatively large. This is mainly due to the effect of a flush policy, which writes dirty pages staying in a buffer cache for a long time to flash memory. This flush policy is essential to a write buffer for ensuring a high degree of data reliability. However, with a flush policy, hot pages must be written to flash memory, despite of their high localities. Therefore, the impact of using a large buffer cache on performance is less effective.

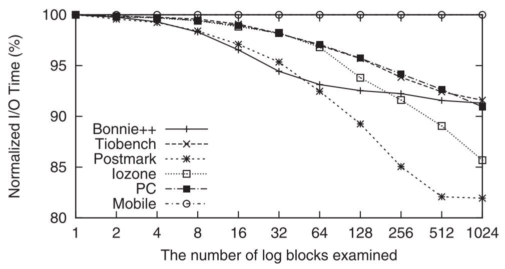


Fig. 12. An evaluation of the limited version of BAGC.

7.6 The Effect of the Number of Log Blocks Examined

We evaluate the performance of the limited version of BAGC while varying the number, N_{lb}^{limit}, of log blocks examined from 1 to 1,024. The FAST FTL is used as the FTL scheme and 1,024 log blocks are available in the FTL. The buffer cache is set to 32 MB. The results, shown in Fig. 12, demonstrate that increasing N_{lb}^{limit} generally reduces the I/O time. For the Bonnie++, the performance saturates somewhere between 128 and 256 log blocks. The performance of *BAGC* on Tiobench, Postmark, Iozone, and PC continues to improve as N_{lb}^{limit} increases. The performance of Mobile is not changed much because the garbage collection overhead itself is trivial. These results clearly indicate that it is necessary to determine the value of N_{lb}^{limit} with great consideration of a characteristic of a workload.

8 Conclusion

We have presented a new buffer-aware garbage collection scheme called BAGC, which combines two principal techniques: buffer-aware block merge and buffer-aware victim block selection. BABM improves the efficiency of a block merge by eliminating unnecessary page migrations. BAVBS improves I/O performance by selecting a victim block in a way that takes account of the potential benefits of the buffer-aware block merge. Experimental results show that BAGC improves I/O performance by up to 43 percent compared to existing buffer-unaware schemes.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable suggestions that greatly improved this paper. This work was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Education under Grant 2012-0006417. The ICT at Seoul National University and IDEC provided research facilities for this study. An earlier version of this paper was presented at the 2008 International Workshop on Software Support for Portable Storage [1].

References

[1] S. Lee, D. Shin, and J. Kim, “Buffer-Aware Garbage Collection for NAND Flash Memory-Based Storage Systems,” *Proc. Int'l Workshop Software Support for Portable Storage,* pp. 27-32, 2008.
[2] S. Lee, D. Shin, and J. Kim, “Buffer-Aware Garbage Collection Technique for NAND Flash-Based Storage Devices,” Technical Report TR-CARES-04-11, http://cares.snu.ac.kr/download/TR-CARES-04-11.pdf, 2011.

[3] J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee, "A Multi-Channel Architecture for High-Performance NAND Flash-Based Storage System," *J. Systems Architecture,* vol. 53, no. 9, pp. 644-658, 2007.

[4] "K9WBG08U1M NAND Flash Memory," Samsung Corp., 2007.

[5] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 86-97, 1994.

[6] H.-J. Kim and S.-G. Lee, "A New Flash Memory Management for Flash Storage System," *Proc. Computer Software and Applications Conf.,* pp. 284-289, 1999.

[7] H.-L. Li, C.-L. Yang, and H.-W. Tseng, "Energy-Aware Flash Memory Management in Virtual Memory System," *IEEE Trans. Very Large Scale Integration Systems,* vol. 16, no. 8, pp. 952-964, Aug. 2008.

[8] A. Ban, "Flash File System," US patent 5,404,485, Washington, D.C.: Patent and Trademark Office, Apr. 1995.

[9] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compact Flash Systems," *IEEE Trans. Consumer Electronics,* vol. 48, no. 2, pp. 366-375, May 2002.

[10] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer Based Flash Translation Layer Using Fully Associative Sector Translation," *ACM Trans. Embedded Computing Systems,* vol. 6, no. 3, pp. 1-27, 2007.

[11] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A Superblock-Based Flash Translation Layer for NAND Flash Memory," *Proc. Int'l Conf. Embedded Software,* pp. 161-170, 2006.

[12] S. Boboila and P. Desnoyers, "Write Endurance in Flash Drives: Measurements and Analysis," *Proc. USENIX Conf. File and Storage Technologies,* pp. 115-128, 2010.

[13] S. Boboila and P. Desnoyers, "Performance Models of Flash-Based Solid-State Drives for Real Workloads," *Proc. Symp. Mass Storage Systems and Technologies,* pp. 1-6, 2011.

[14] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 1075-1086, 2008.

[15] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: Flash-Aware Buffer Management Policy for Portable Media Players," *IEEE Trans. Consumer Electronics,* vol. 52, no. 2, pp. 485-493, May 2006.

[16] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," *Proc. USENIX Conf. File and Storage Technologies,* pp. 239-252, 2008.

[17] D. Seo and D. Shin, "Recently-Evicted-First Buffer Replacement Policy for Flash Storage Devices," *IEEE Trans. Consumer Electronics,* vol. 54, no. 3, pp. 1228-1235, Aug. 2008.

[18] S. Ji and D. Shin, "Locality and Duplication-Aware Garbage Collection for Flash Memory-Based Virtual Memory Systems," *Proc. Int'l Conf. Computer and Information Technology,* pp. 1764-1771, 2010.

[19] Y.J. Seong, E.H. Nam, J.H. Yoon, H. Kim, J.-Y. Choi, S. Lee, Y.H. Bae, J. Lee, Y. Cho, and S.L. Min, "Hydra: A Block-Mapped Parallel Flash Memory Solid-State Disk Architecture," *IEEE Trans. Computers,* vol. 59, no. 7, pp. 905-921, July 2010.

[20] M. Russinovich, "DiskMon for Windows v2.01," http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx, 2006.

Sungjin Lee received the BE degree in electrical engineering from Korea University in 2005 and the MS and PhD degrees in computer science and engineering from the Seoul National University in 2007 and 2013, respectively. He is currently working as a postdoctoral associate in the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. His research interests include storage systems, operating systems, and embedded software.

Dongkun Shin received the BS degree in computer science and statistics, the MS degree in computer science, and the PhD degree in computer science and engineering from Seoul National University, Korea, in 1994, 2000, and 2004, respectively. He is currently an assistant professor in the School of Information and Communication Engineering, Sungkyunkwan University (SKKU). Before joining SKKU in 2007, he was a senior engineer at Samsung Electronics Co., Korea. His research interests include embedded software, low-power systems, computer architecture, and real-time systems. He is a member of the IEEE.

Jihong Kim received the BS degree in computer science and statistics from Seoul National University (SNU), Korea, in 1986, and the MS and PhD degrees in computer science and engineering from the University of Washington, Seattle, in 1988 and 1995, respectively. Before joining SNU in 1997, he was a technical staff member at the DSPS R&D Center of Texas Instruments in Dallas, Texas. He is currently a professor at the School of Computer Science and Engineering, Seoul National University. His research interests include embedded software, low-power systems, computer architecture, and storage systems. He is a member of the IEEE.