Buffer-Aware Garbage Collection for NAND Flash Memory-Based Storage Systems

Sungjin Lee*, Dongkun Shin† and Jihong Kim*

*School of Computer Science and Engineering, Seoul National University, Seoul, Korea
{chamdoo, jihong}@davinci.snu.ac.kr

†School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea
dongkun@skku.edu

ABSTRACT

With continuing improvements in both the price and the capacity, flash memory-based storage devices are becoming a viable solution for satisfying high-performance storage demands of desktop systems as well as mobile embedded systems. Because of the erase-before-write characteristic of flash memory, a flash memory-based storage system requires a garbage collection, which often introduces large performance degradation due to a large number of page migrations and block erase operations. In order to improve the overall I/O performance of the flash-based storage systems, therefore, it is important to support the garbage collection efficiently. In this paper, we propose a novel garbage collection scheme, called buffer-aware garbage collection (BA-GC), for flash memory-based storage systems. In implementing two main steps of the garbage collection module, the block merge step and the victim block selection step, the proposed BA-GC scheme takes into account the contents of a buffer cache (e.g., a page cache and a disk buffer) which is used to enhance the I/O performance of storage systems. The buffer-aware block merge (BA-BM) scheme reduces the number of unnecessary page migrations by enforcing a dirty page eviction in the buffer cache during the garbage collection. The buffer-aware victim block selection (BA-VBS) scheme, on the other hand, selects a victim block so that the overall I/O performance can be maximized. Our experimental results show that the proposed BA-GC technique improves the overall I/O performance up to 45% over existing buffer-unaware schemes.

1. INTRODUCTION

Flash memory has been widely used as a storage device for mobile embedded systems because of its low-power consumption, non-volatility, high random access performance and high mobility. For the past several years, there has been a significant growth in the NAND flash market due to the exponential growth in MP3 players and digital cameras which require a large amount of data storage. Recently, solid-state disks (SSDs), which are based on NAND flash memory, are quickly expanding their market share in the general-purpose storage market, replacing hard disks [1].

Flash memory has unique operation features which are not found in other storage devices such as magnetic hard disks. First, flash memory is based on the “erase-before-write" architecture. It means that flash media must be erased before it is to be reused for new data. Second feature is that the unit sizes of erase and read/write operations are asymmetric. While flash memory is erased by the unit of block, read/write operations are performed by the smaller unit of page. A single block is composed of multiple pages.

Due to these two features, a special software called flash translation layer (FTL) is usually employed, which makes flash memory fully emulates block devices. Although several FTL schemes have been proposed, log buffer-based FTL schemes [2, 3, 4] are widely used in many flash memory applications since they show a good performance while requiring a relatively low system resource.

The basic concept of log buffer-based FTLs is quite similar to that of LFS [5]. In this scheme, all the physical blocks are separated between log blocks and data blocks. While log blocks are used for storing update data temporally, data blocks are used for ordinary storage space. Therefore, when an update request is sent to FTL, the data is first written into a log block and the corresponding old data in a data block is invalidated. However, when the log blocks are full, FTL should do a garbage collection to make free space for new data. The garbage collector first selects a victim log block to be erased. Before erasing the victim log block, all valid pages related to the victim log block should migrate to new data blocks. Especially, this page migration step is called a block merge since valid pages in both data blocks and log blocks are merged into new data blocks. Therefore, we can divide the garbage collection into two steps, the *victim block selection* and the *block merge*.

Due to the high erase and write costs of flash memory, the overall performance of flash memory systems heavily depends on the number of erase and write operations. Especially, since the garbage collection incurs many erase and write operations, the garbage collection overhead accounts for a significant portion of the I/O execution time. Therefore, most existing FTLs were focusing on reducing erase and write operations during the garbage collection.

In this paper, we propose a novel garbage collection scheme, called buffer-aware garbage collection (BA-GC). The main difference of the BA-GC scheme over existing garbage collection algorithms is that the BA-GC scheme takes into account the contents of a buffer cache[1] during the garbage collection steps. By exploiting the contents of the buffer cache, better decisions can be made during the garbage collection, thus improving the performance of NAND flash-based storage systems. For the block merge step, we propose a buffer-aware block merge (BA-BM) technique. The main novelty of the BA-BM scheme is that it can avoid many *unnecessary* page migrations (which are required by existing techniques) by examining the contents of the buffer cache. For the victim block selection step, we propose a buffer-aware victim block selection (BA-VBS) technique. The BA-VBS scheme selects a victim block by considering a temporal locality of pages in the buffer cache so that the overall I/O performance can be maximized.

Experimental results based on a trace-driven simulator show that the proposed BA-GC scheme significantly improves the efficiency of the garbage collection, and reduces the overall I/O execution times by 10%-45% over existing buffer-unaware garbage collection schemes for various benchmark programs such as Iozone, Postmark, Bonnie++ and realistic workloads.

The rest of the paper is organized as follows. In Section 2, we briefly explain the related works. Section 3 presents the motivation of the buffer-aware garbage collection. Sections 4 and 5 describe the buffer-aware block merge technique and the buffer-aware victim block selection technique, respectively. Experimental results are presented in Section 6. Section 7 concludes with a summary and future works.

2. RELATED WORKS

[1]In order to improve the performance of a file system or a storage device, various storage buffers (e.g., a page cache and a disk buffer) are used. In this paper, we call such storage buffers as the buffer caches.

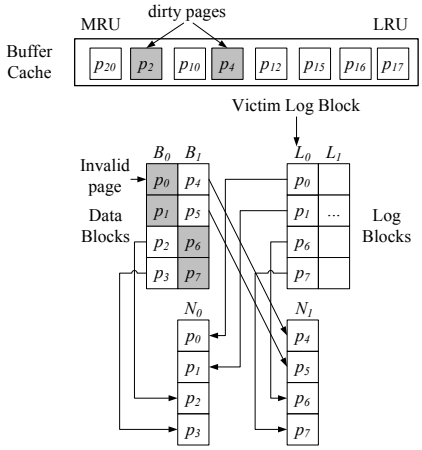

Figure 1: An example of unnecessary page migrations.

The garbage collection techniques for FTL are seriously studied. Especially, in log buffer-based FTLs, there are three kinds of approaches depending on the block association policy, i.e., block-associative sector translation (BAST) [2], fully-associative sector translation (FAST) [3] and set-associative sector translation (SAST) [4]. The block association policy means how many data blocks can share a log block at a time. In the BAST scheme, a log block is used for only one data block. In the FAST scheme, a log block can be used for several data blocks. In the SAST scheme, a set of log blocks can be used for a set of data blocks. Generally, a round robin policy is used in selecting a victim log block in [2, 3]. FAST provides a quite efficient garbage collection for the random write access pattern by increasing the number of data blocks sharing for one log block. The superblock scheme [4], one example of SAST, uses a utilization-based policy which selects a block with the smallest number of valid pages. It also exploits the locality of write accesses by combining several blocks into a superblock. All these existing schemes are quite different from our proposed scheme in that they have no consideration on the buffer cache.

There are not many researches on the buffer cache in flash memory systems. Jo et al. [6] proposed a flash-aware buffer management scheme. Using a block-level buffer replacement which evicts all the pages of a block at a time, it reduced the garbage collection cost. Park et al. [7] proposed a clean-first LRU (CF-LRU) replacement policy which delays the eviction of a dirty page in the buffer cache to reduce the write requests to flash memory. Recently, Kim et al. [8] presented a buffer management scheme especially for improving the performance of random writes. This scheme also uses the block-level buffer replacement like [6], but it tries to enhance the sequentiality of flash writes when evicting a victim block in order to reduce the garbage collection overhead. These three schemes handled only the buffer cache management, but proposed no techniques on the garbage collection and did consider neither the unnecessary page migration nor the victim log block selection.

3. MOTIVATION

In this section, we explain the main benefits of making a garbage collector *buffer-aware* using a simple scenario. When the garbage collector selects a victim block and reclaims the block, a large number of page migrations are necessary. One of our main observations is that many of these page migrations can be avoided if we refer to the contents of the buffer cache.

Figure 1 shows an example in the FAST scheme [3]. There are a buffer cache and flash memory blocks. The buffer cache has eight pages and two of them (p_2 and p_4) are dirty pages. B_0 and B_1 are data blocks, and L_0 is a log block. Each block consists of four pages. While the block-level mapping is used for the data blocks, the page-level mapping is used for the log blocks. Due to the updates on the pages p_0, p_1, p_6 and p_7, the log block L_0 has new updated pages, thus making original pages in B_0 and B_1 invalidated. If the garbage collector selects the log block L_0 as a victim block, new data blocks N_0 and N_1 are allocated and the valid pages in B_0, B_1 and L_0 are moved to N_0 and N_1. After the page migration, we can erase the blocks B_0, B_1 and L_0.

Under this scenario, the valid pages p_2 and p_4 in the data blocks B_0 and B_1 are moved to N_0 and N_1 to erase B_0 and B_1. However, if we can examine the buffer cache, we know that moving p_2 and p_4 is unnecessary because they will be invalidated shortly when the dirty pages in the buffer cache are evicted into flash memory. If we can move p_2 and p_4 from the buffer cache instead of B_0 and B_1, we fill the new data blocks, N_0 and N_1, with up-to-date data. Therefore, we can know that an FTL should be *buffer-aware* to prevent unnecessary page migrations.

By using the buffer-aware approach, we can get two kinds of benefits. First, we can reduce the number of page write requests generated by the evicted dirty pages. Since the modified data of p_2 and p_4 are written to new data blocks of flash memory at the garbage collection time, the pages p_2 and p_4 in the buffer cache are changed into clean state. Therefore, when the pages p_2 and p_4 are evicted from the buffer cache by the buffer replacement policy, they do not generate write requests to flash memory. Second, we can eliminate the block merge which will occur in the near future. For instance, if the pages p_2 and p_4 are moved from the blocks B_0 and B_1 instead of the buffer cache, the dirty pages in the buffer cache will be written in the log block (for example, L_1) shortly. Once the pages are written into the log block L_1, they should be merged with the corresponding data blocks, N_0 and N_1, at the garbage collection time when all the empty pages in log blocks are exhausted. However, by directly writing the pages p_2 and p_4 from the buffer cache to the data blocks N_0 and N_1 at the garbage collection time of L_0, the block merge for L_1 can be avoided.

We have observed that 5%-50% of total page migrations are unnecessary depending on characteristics of workloads. It means that the buffer-aware approach can significantly reduce the garbage collection overhead. Especially, since the page migration during the garbage collection is inevitable to all kinds of FTLs, eliminating the unnecessary page migration will be quite beneficial to all FTL schemes regardless of their block association policies. In this paper, we evaluate the proposed BA-GC scheme under FAST FTL [3] because it is the most representative FTL scheme.

For the garbage collector to be buffer-aware, it should be possible that FTL accesses the buffer cache. When flash memory is used for the removable storage device using USB or ATA interface, the FTL resides in the external storage device but the page cache is managed by the host system. However, many CE devices such as mobile phones and MP3 players have NAND flash memory as embedded storage. In this case, the page cache manager and the FTL can share their information because both of them are executed at the same processor. If the page cache manager can notify the FTL of the dirty page information, we can prevent the unnecessary page migrations. SSD is also a good target. Generally, there is an SDRAM buffer exploited as a disk buffer inside SSD [9], and a controller which manages all of the address mapping, garbage collection and buffer management. Therefore, the garbage collection module can access the disk buffer state. So, our buffer-aware garbage collection technique can be targeted for the embedded flash storage and the solid-state disk.

4. BUFFER-AWARE BLOCK MERGE

In order to prevent the unnecessary page migrations, the proposed techniques refer to contents of the buffer cache during the block merge. If there are the corresponding dirty pages in the buffer cache for pages to be moved by the block merge at the garbage collection time, we can take two kinds of approaches for an efficient

```
1: Buffer_Aware_Block_Merge(L_i) {
2:   for D_j ∈ 𝔸(L_i) {
3:     get the new data block D_j^new from the free block list;
4:     for p_k ∈ D_j {
5:       if p_k exists in DPL {
6:         move p_k from DPL into D_j^new;
7:         remove p_k from DPL;
8:       } else {
9:         if p_k ∈ D_j^valid
10:          move p_k from D_j into D_j^new;
11:        else { /* p_k is invalid */
12:          find log block L_j, which has a valid p_k, from 𝕃(D_j);
13:          move p_k from L_j into D_j^new;
14:        }
15:      }
16:      invalidate p_k in either L_j or D_j;
17:    }
18:    erase the data block D_j;
19:    insert the erased block D_j into the free block list;
20:  }
21:  erase the log block L_i;
22:  insert the erased block L_i into the free block list;
23: } /* end of function */
```

Figure 2: The buffer-aware block merge algorithm.

FTL. First is to move the up-to-date pages in the buffer cache into new allocated data blocks. Second, if the corresponding pages will not be evicted to flash memory in the near future (for example, they have high temporal locality), it is more beneficial to find another victim log block, which is related to pages to be evicted soon. The former is related to the block merge step and the latter is related to the victim block selection step of the garbage collection. In this section, we first propose a buffer-aware block merge (BA-BM) scheme which eliminates the unnecessary page migrations by fetching dirty pages from the buffer cache during the block merge. The buffer-aware victim block selection (BA-VBS) technique is presented in Section 5.

If the buffer cache is used for a read cache or a prefetch buffer as well as a write buffer, it is more efficient to manage only the information about the dirty pages. Therefore, we use the *dirty page list* (DPL) which has the addresses of dirty pages in the order of written time. Figure 2 shows the algorithm of the buffer-aware block merge. Before the block merge, we should identify the set of associated data blocks, $\mathbb{A}(L_i)$, of a victim log block L_i. $\mathbb{A}(L_i)$ is composed of all data blocks which have the corresponding invalid pages for valid pages in the log block L_i. We can represent $\mathbb{A}(L_i)$ formally as follows:

$$\mathbb{A}(L_i) = \{D_j | \exists p_k \; s.t. \; p_k \in L_i^{valid} \wedge p_k \in D_j^{invalid}\}, \qquad (1)$$

where D_j means a block and p_k means a page. $D_j^{invalid}$ and L_i^{valid} denote the set of invalid pages of D_j and the set of valid pages of L_i, respectively. For example, $\mathbb{A}(L_0)$ in Figure 1 is $\{B_0, B_1\}$.

For each page, p_k, in an associated data block D_j in $\mathbb{A}(L_i)$, DPL is examined. If the page exists in DPL, the dirty page is moved from the buffer cache into the new data block D_j^{new}. Otherwise, the page is moved from D_j or from another log block L_j if p_k in D_j is invalidated. In order to find L_j, we need to identify the set of associated log blocks, $\mathbb{L}(D_j)$, of an associated data block D_j. This is because one data block can be associated with several log blocks. $\mathbb{L}(D_j)$ is composed of all log blocks which have the corresponding valid pages for invalid pages in the data block D_j. Therefore, we can represent $\mathbb{L}(D_j)$ formally as follows:

$$\mathbb{L}(D_j) = \{L_j | \exists p_k \; s.t. \; p_k \in D_j^{invalid} \wedge p_k \in L_j^{valid}\}, \qquad (2)$$

where $D_j^{invalid}$ and L_j^{valid} denote the set of invalid pages of D_j and the set of valid pages of L_j, respectively. For example, $\mathbb{L}(B_0)$ in Figure 1 is $\{L_0\}$. After the migration, the dirty page in the buffer cache is changed into a *clean* state and removed from DPL. The data block D_j and the log block L_i are erased and moved into the free block list for future use.

To know the effect of buffer-aware block merge, consider the block merge cost for the victim log block L_i. In the *buffer-unaware* block merge (BU-BM), we can represent the block merge cost for L_i, $\delta_{merge}^{BU}(L_i)$, as follows:

$$\begin{aligned} \delta_{merge}^{BU}(L_i) &= |\mathbb{M}(L_i)| \cdot C_{f \to f}, \\ \text{where } \mathbb{M}(L_i) &= \bigcup_{b_j \in \mathbb{A}(L_i)} (M_d(b_j) \cup \bigcup_{b_k \in \mathbb{L}(b_j)} M_l(b_j, b_k)), \\ M_d(b_j) &= \{p_k | p_k \in b_j^{valid}\}, \\ M_l(b_j, b_k) &= \{p_k | p_k \in b_j^{invalid} \wedge p_k \in b_k^{valid}\}, \end{aligned} \qquad (3)$$

where b_j is an associated data block and b_k is an associated log block. $\mathbb{M}(L_i)$ is the set of valid pages to be moved when L_i is selected as a victim log block. For example, $\mathbb{M}(L_0)$ in Figure 1 is $\{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$. $C_{f \to f}$ denotes a page migration cost between flash memory blocks. $C_{f \to f}$ can be represented as $C_r + C_p$ ignoring the computation cost, where C_r and C_p are the read cost and the program cost for a flash memory page.

In the *buffer-aware* block merge, the block merge cost for L_i, $\delta_{merge}^{BA}(L_i)$, can be represented as follows:

$$\begin{aligned} \delta_{merge}^{BA}(L_i) &= |\mathbb{B}(L_i)| \cdot C_{b \to f} + |\mathbb{F}(L_i)| \cdot C_{f \to f}, \\ \text{where } \mathbb{B}(L_i) &= \bigcup_{b_j \in \mathbb{A}(L_i)} B(b_j), \\ \mathbb{F}(L_i) &= \bigcup_{b_j \in \mathbb{A}(L_i)} (F_d(b_j) \cup \bigcup_{b_k \in \mathbb{L}(b_j)} F_l(b_j, b_k)), \\ \text{where } B(b_j) &= \{p_k | p_k \in DPL \wedge p_k \in b_j\}, \\ F_d(b_j) &= \{p_k | p_k \notin DPL \wedge p_k \in b_j^{valid}\}, \\ F_l(b_j, b_k) &= \{p_k | p_k \notin DPL \wedge p_k \in b_j^{invalid} \wedge p_k \in b_k^{valid}\}, \end{aligned} \qquad (4)$$

where $\mathbb{F}(L_i)$ is the set of valid pages to be moved from flash memory when L_i is selected as a victim log block. $\mathbb{B}(L_i)$ is the set of valid pages to be moved from the buffer cache. For example, $\mathbb{F}(L_0)$ and $\mathbb{B}(L_0)$ in Figure 1 are $\{p_0, p_1, p_3, p_5, p_6, p_7\}$ and $\{p_2, p_4\}$, respectively. $C_{b \to f}$ denotes a page migration cost from the buffer cache to flash memory block. $C_{b \to f}$ can be represented as $C_b + C_p$ ignoring the computation cost, where C_b is the read cost for a page in the buffer cache. Generally, $C_{b \to f}$ is smaller than $C_{f \to f}$ since the flash read cost is larger than the volatile memory (SDRAM) read cost. So, as there are more dirty pages in the buffer cache to replace the unnecessary page migrations, $|\mathbb{B}(L_i)|$ increases and the block merge cost decreases.

While the $|\mathbb{B}(L_i)|$ number of dirty pages in DPL should be written to log blocks after the garbage collection under BU-BM, the pages will not be written to log blocks under BA-BM unless they are changed again into dirty states after it is changed into the clean state by BA-BM. Therefore, we can say that BA-BM reduces write requests to the log block. So, it is necessary to investigate the potential benefits, $\delta_{benefit}^{BA}(L_i)$, of BA-BM, which represents the gain from the reduced number of write requests. When we denote the dirty page eviction cost after the garbage collection for L_i under BU-BM and under BA-BM as $\delta_{evict}^{BU}(L_i)$ and $\delta_{evict}^{BA}(L_i)$ respectively, the $\delta_{benefit}^{BA}(L_i)$ can be denoted by $\delta_{evict}^{BU}(L_i) - \delta_{evict}^{BA}(L_i)$. If we assume that all the pages changed into clean state by BA-BM are not changed again into dirty state before they are evicted from the buffer cache, we can say that $\delta_{benefit}^{BA}(L_i) = |\mathbb{B}(L_i)| \cdot C_p$.

Moreover, there is an additional benefit of BA-BM. After dirty pages are written into log blocks, these log blocks should be eventually merged with the corresponding data blocks. Since BA-BM

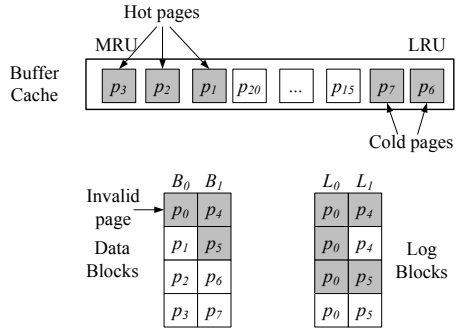

Figure 3: A motivational example for the victim block selection

reduces the number of dirty page evictions to log blocks, it has the benefit of reducing the block merge cost. Therefore, we can express this additional benefit as $(\delta^{BU}_{evict}(L_i) - \delta^{BA}_{evict}(L_i)) \cdot \alpha$ where α is the weight value which represents the block merge cost to be incurred by one dirty page eviction. The α value can be expressed as $(C_p + C_r) / C_p$ because one dirty page eviction typically incurs one flash read cost and one flash program cost during the block merge. Therefore, we can express the potential benefits, $\delta^{BA}_{benefit}(L_i)$, as follows:

$$\delta^{BA}_{benefit}(L_i) = (\delta^{BU}_{evict}(L_i) - \delta^{BA}_{evict}(L_i)) \cdot (1+\alpha). \quad (5)$$

Finally, we can represent the gain of the buffer-aware block merge technique over the buffer-unaware block merge technique assuming that $\delta^{BU}_{evict}(L_i) - \delta^{BA}_{evict}(L_i) = |\mathbb{B}(L_i)| \cdot C_p$.

$$\begin{aligned} &\delta^{BU}_{merge}(L_i) - \delta^{BA}_{merge}(L_i) + \delta^{BA}_{benefit}(L_i) \\ &= |\mathbb{M}(L_i)| \cdot C_{f\to f} - |\mathbb{F}(L_i)| \cdot C_{f\to f} - |\mathbb{B}(L_i)| \cdot C_{b\to f} + \\ &\quad |\mathbb{B}(L_i)| \cdot C_p \cdot (1+\alpha) \\ &= |\mathbb{B}(L_i)| \cdot (C_r - C_b + C_p \cdot (1+\alpha)). \end{aligned} \quad (6)$$

Eq. (6) shows the maximum gain of the buffer-aware block merge since we assume that all the pages changed into clean state by BA-BM are not changed again into dirty state before they are evicted from the buffer cache. However, some pages will be changed into dirty state before their evictions, and then the benefit of BA-BM will be decreased. This will be discussed in the following section.

5. BUFFER-AWARE VICTIM BLOCK SELECTION

5.1 Motivation

To reduce the I/O cost and maximize the flash memory life time, the previous studies considered the number of valid pages or the wear-level of a block when selecting the victim block. For the buffer-aware garbage collection, the merge cost of BA-BM for a candidate victim block should be considered in the victim block selection step. Especially, we should consider the potential benefits of BA-BM to select the optimum victim. Otherwise, the victim block selection cannot select the optimum victim block which will produce the minimum I/O cost. The benefit of BA-BM will depend on the characteristics of the dirty pages in the buffer cache.

Consider the snapshot of flash memory and the buffer cache shown in Figure 3. In this example, the buffer cache has eight pages and five of them (i.e., p_1, p_2, p_3, p_6 and p_7) are dirty pages. While the pages p_1, p_2 and p_3 are hot pages, i.e., will be updated frequently, the pages p_6 and p_7 are cold pages, i.e., will not be updated before they are evicted from the buffer cache by the replacement policy. There are two data blocks, B_0 and B_1, and two log blocks, L_0 and L_1. If the log block L_0 is selected as a victim by the garbage collector, the block merge cost under BA-BM, $\delta^{BA}_{merge}(L_0) = 1 \cdot C_{f\to f} + 3 \cdot C_{b\to f}$. If the log block L_1 is selected as a victim, the block merge cost, $\delta^{BA}_{merge}(L_1) = 2 \cdot C_{f\to f} + 2 \cdot C_{b\to f}$. Therefore, selecting the block L_0 as a victim seems to be better assuming $C_{b\to f} < C_{f\to f}$.

However, migrating the pages p_1, p_2, and p_3 from the buffer cache to flash memory is useless because they will be updated shortly and become dirty pages. Since the dirty pages will be eventually written to flash memory, they will incur the write requests on the three flash memory pages. Then, the benefit of BA-BM, $\delta^{BA}_{benefit}(L_0)$ becomes 0. On the other hand, if the log block L_1 is selected as a victim, the pages p_6 and p_7 will not incur write requests when they are evicted from the buffer cache since the pages will be clean. Then, the benefit of BA-BM, $\delta^{BA}_{benefit}(L_1)$, becomes $2 \cdot C_p \cdot (1+\alpha)$. Consequently, regarding the reduction of the overall garbage collection cost, choosing the block L_1 as a victim is more beneficial.

5.2 Locality-aware victim block selection

In the proposed buffer-aware victim block selection, we use the garbage collection cost of a log block L_i, $\Delta(L_i)$, as a priority in selecting a victim log block; the log block with the lowest cost is selected. The garbage collection cost consists of the block merge cost, the block erase cost and the potential benefit. Since we should select the log block with the lowest block merge cost, the lowest block erase cost and the highest potential benefits, we define the $\Delta(L_i)$ as follows:

$$\begin{aligned} \Delta(L_i) = \ & \delta^{BA}_{merge}(L_i) + \delta_{erase}(L_i) - \delta^{BA}_{benefit}(L_i) \\ & \text{where } \delta_{erase}(L_i) = (|\mathbb{A}(L_i)| + 1) \cdot C_e, \end{aligned} \quad (7)$$

where C_e stands for the erase cost of a flash block.

In order to know the potential benefits of the buffer-aware block merge, we should estimate the exact value of $(\delta^{BU}_{evict}(L_i) - \delta^{BA}_{evict}(L_i))$ in Eq. (5). Though we know that the value of $\delta^{BU}_{evict}(L_i)$ is $|\mathbb{B}(L_i)| \cdot C_p$, the value of $\delta^{BA}_{evict}(L_i)$ cannot be known. By the buffer-aware merge, all the dirty pages in $\mathbb{B}(L_i)$ are moved into flash memory. After the migration, the dirty pages are changed into *clean* state. If there is no update on the clean pages until it is evicted from the buffer cache by a page replacement policy, there is no write cost due to the page eviction. Then, $\delta^{BA}_{evict}(L_i) = 0$. However, if all the pages in $\mathbb{B}(L_i)$ are changed again into *dirty* state after the buffer-aware block merge and before the page eviction, $\delta^{BA}_{evict}(L_i)$ is equal to $|\mathbb{B}(L_i)| \cdot C_p$. In this case, it will be more profitable to prevent BA-BM from fetching the dirty pages in the buffer cache. To do that, we should select another log block as a victim block for the garbage collection. In this paper, we predict the value of $\delta^{BA}_{evict}(L_i)$ based on the update probabilities of pages in $\mathbb{B}(L_i)$.

In order to identify how many pages in $\mathbb{B}(L_i)$ will be updated before it is evicted from the buffer cache, we divide $\mathbb{B}(L_i)$ into two subsets, $\mathbb{B}_c(L_i)$ and $\mathbb{B}_d(L_i)$. If a page $p_k \in \mathbb{B}(L_i)$ is dirty when it is evicted from the buffer cache by a page replacement policy, $p_k \in \mathbb{B}_d(L_i)$. Otherwise, $p_k \in \mathbb{B}_c(L_i)$. It is better to exclude the block L_i from a victim block if $|\mathbb{B}_d(L_i)|$ is large. We should assign a low benefit to a log block L_i when most of dirty pages are in $\mathbb{B}_d(L_i)$ to prevent L_i from being selected as a victim log block. Using $\mathbb{B}_d(L_i)$ for a log block L_i, we can write the value of $\delta^{BA}_{evict}(L_i)$ as follows:

$$\delta^{BA}_{evict}(L_i) = |\mathbb{B}_d(L_i)| \cdot C_p. \quad (8)$$

Using this value, we can select a log block with the lowest garbage collection cost as a victim block. Since $|\mathbb{B}_d(L_i)|$ depends on how frequently each page will be updated, we can get the approximated value of $|\mathbb{B}_d(L_i)|$ as follows:

$$|\mathbb{B}_d(L_i)| \ \simeq \sum_{p_k \in \mathbb{B}(L_i)} P(\mathcal{U}_{p_k}), \quad (9)$$

where $\mathcal{U}_{p_k}$ indicates an event that a page p_k will be updated before

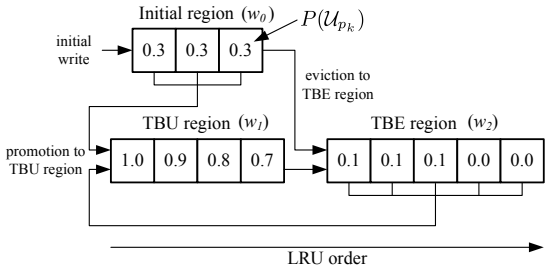

Figure 4: The 3-region LRU buffer.

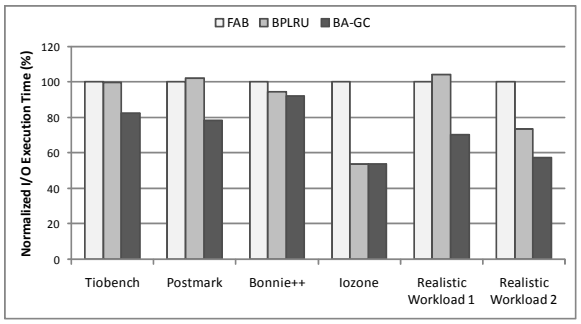

Figure 5: A comparison of the normalized execution times of three schemes

it is evicted by a page replacement policy, and $P(\mathcal{U}_{p_k})$ indicates the probability that the event $\mathcal{U}_{p_k}$ occurs.

However, it is impossible to know the exact value of $P(\mathcal{U}_{p_k})$ at the garbage collection time because the future behavior of the buffer cache is unknown. To predict $P(\mathcal{U}_{p_k})$, we use a locality-aware approach. Due to a temporal locality, it is probable that a recently-updated page will be updated again. Therefore, as p_k has been written more frequently and recently, we assign a larger value to $P(\mathcal{U}_{p_k})$.

To assign $P(\mathcal{U}_{p_k})$ of each p_k easily, we invented a novel buffer architecture called 3-region LRU buffer where the buffer cache is divided into three regions: a TBU (To-Be-Updated) region, a TBE (To-Be-Evicted) region, and an initial region, as shown in Figure 4. While the $P(\mathcal{U}_{p_k})$ of a page in the TBU region is set to a value close to 1, the value in the TBE region is set to a value close to 0. To prevent the write-once data, which is not updated after it is written to the buffer cache, from being in the TBU region just after it is first written, the initial region is used. When a page is first written at the buffer cache, it is located at the initial region. If the data is updated afterward, it is promoted to the TBU region. Otherwise, it is evicted to the TBE region in the order of the arrival time to the initial region. The page in the TBE region is promoted to the TBU region if it is updated. In each region, pages are managed by the LRU order. If there is a temporal locality in the buffer cache write pattern, the page with a small $P(\mathcal{U}_{p_k})$ value (i.e., no access during long time) will be sent to flash memory shortly, but the page with a large $P(\mathcal{U}_{p_k})$ value (i.e., accessed recently and frequently) will reside in the buffer cache for a long time.

The sizes of the initial region (w_0), the TBU region (w_1), and the TBE region (w_2) should be dynamically adjusted depending on the characteristic of the buffer cache access pattern because they determine $\sum P(\mathcal{U}_{p_k})$. Although the region size adjustment process has been adopted in the proposed BA-GC scheme, due to the limited space of the paper, we omit detailed explanation of the size adjustment process. We encourage readers to refer to [10].

6. EXPERIMENTS

6.1 Experimental environments

In order to evaluate the performance of the proposed BA-GC scheme, we have developed a trace-driven simulator. In simulations, we used a flash memory model based on Samsung large block NAND flash memory (K9WBG08U1M) with 64 2 KB pages in each block. The access times of page read, page write and block erase are 25 μsec, 200 μsec and 2 msec, respectively.

We compared the proposed BA-GC scheme with two existing buffer management schemes, FAB [6] and BPLRU [8]. For a fair comparison, all three schemes were implemented using the same FAST FTL scheme [3]. For the BA-GC scheme, the block merge and the victim log block selection modules of the FAST scheme were modified to be buffer-aware as mentioned in Sections 4 and 5. All the data stored on the 3-region LRU buffer were managed at the block level like in FAB and BPLRU. Therefore, all the pages that belong to the same logical block had the same $P(\mathcal{U}_{p_k})$ value, and they were evicted from the buffer cache by the buffer replacement policy at a time. Regarding the parameters used in the BA-GC scheme, $P(\mathcal{U}_{p_k})$ values for pages of the initial, TBU, and TBE regions were set as 0.3, 1.0 and 0.0, respectively. The α value was set to as 1.125 because C_p=200 μsec and C_r=25 μsec.

The main performance metric used in our experiments was the I/O execution time, which includes the normal flash write cost as well as the garbage collection cost. Since the BPLRU scheme was designed for the write buffer in the storage device, for the comparison, read requests issued from file system are filtered by the simulator. We ignored the time spent for accessing data in the buffer cache because it was negligible compared to the access time of flash memory. In addition, we also adopted the compulsory flush policy for dirty pages to ensure data consistency. Therefore, dirty pages that stayed in the buffer cache more than 30 seconds were flushed into flash memory and their states were changed into clean.

We used six benchmarks: Iozone, Bonnie++, Postmark, Tiobench and two realistic desktop workloads. All the traces were collected from Windows XP operating system with NTFS file system. Two realistic desktop workloads were gathered from a desktop PC, running several applications, such as documents editors, music players, web browsers and games.

6.2 Evaluation results

Figure 5 shows the execution times of the evaluated schemes. In this figure, X-axis denotes the evaluated benchmarks and Y-axis represents the I/O execution time normalized to that of the FAB scheme. Unless otherwise stated, the size of the buffer cache is assumed to be 16 MB while 128 log blocks are used [2].

As shown in Figure 5, the proposed BA-GC scheme effectively reduces the execution times by 30% and 15%, on average, compared to FAB and BPLRU, respectively. BPLRU exhibits a better performance than FAB, but BA-GC also outperforms BPLRU for most of the benchmarks. In BPLRU, before evicting dirty pages from the buffer cache, it reads some pages that are not in a victim block from flash memory, and then writes all pages to flash memory. Although this approach increases the sequentiality of flash writes, it also incurs extra I/O operations on flash memory [8]. Therefore, BPLRU shows a better I/O performance than FAB only for benchmarks where victim blocks have relatively many pages.

In the BA-GC scheme, the overall I/O performance is substantially improved across all the benchmarks, but the degree of performance improvement is quite different depending on the characteristics of the evaluated workloads. To analyze these differences in detail, we have examined how much garbage collection overhead is reduced by BA-GC for each benchmark.

Figure 6 shows the garbage collection overhead normalized to

[2]We have evaluated the performance of the proposed scheme while varying the number of log blocks assigned to FTL. For more detailed experimental results, see the reference [10].

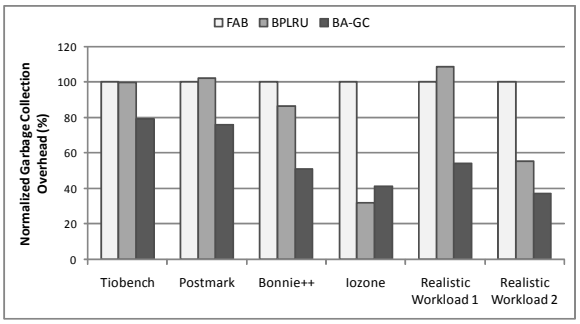

Figure 6: A comparison of the normalized garbage collection overhead of three schemes

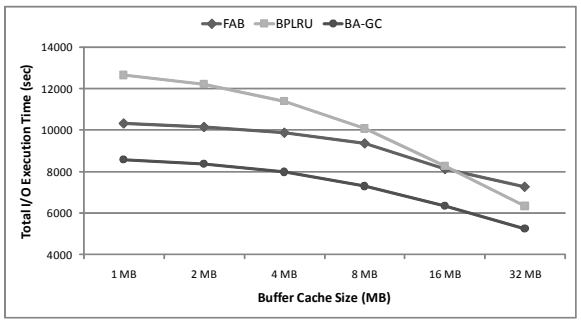

Figure 7: I/O execution time variations under different buffer cache sizes (Postmark benchmark)

FAB. In cases of Tiobench and Postmark, the garbage collection overhead reduced by BA-GC is smaller than the other benchmarks, when compared to FAB. From the observation on the write access patterns of these benchmarks, we found that Tiobench and Postmark exhibit low temporal locality (i.e., write hit ratios for Tiobench and Postmark are 4.4% and 3.1%, respectively). This means that there are many write-once requests, and thus most pages will not be rewritten to the buffer cache after they are evicted to log blocks by the buffer-aware block merge. For this reason, when merging the victim log block, the small number of dirty pages can be moved from the buffer cache to flash memory. This is why the performance improvement by BA-GC is relatively small.

For Bonnie++, Iozone and two realistic workloads, on the other hand, the reduction in the garbage collection overhead is more significant than Tiobench and Postmark. These benchmarks have high temporal locality, and thus many dirty pages can be moved from the buffer cache during the block merge step, reducing the number of writes to flash memory. In addition, by considering temporal locality of dirty pages in the buffer cache, BA-GC can find a proper victim log block which can make a significant impact on the reduction of the overall merge cost.

In the Bonnie++ benchmark, although the garbage collection overhead is significantly reduced, the overall performance gain by BA-GC is small as shown in Figure 5. This is because the garbage collection overhead accounts for a small portion of the I/O time (about 20%). For the Iozone benchmark, BPLRU outperforms BA-GC. However, the 3-region LRU buffer exhibits a higher write hit ratio than BPLRU, and thus the overall performance of BA-GC is almost equivalent to that of BPLRU. In order to efficiently manipulate sequential writes, BPLRU compulsorily evicts the block where all pages are fully sequentially written, regardless of its recency. However, in the Iozone benchmark, since there are many sequential writes and lots of them are likely to be rewritten shortly, BPLRU incurs many write misses. On the other hand, by evicting a block in the LRU order, BA-GC can achieve a higher write hit ratio.

Figure 7 shows the execution times for each evaluated scheme when the size of the buffer cache varies from 1 MB to 32 MB. Overall, the execution time decreases as the size of the buffer cache increases. BA-GC shows a better performance over FAB and BPLRU, regardless of the buffer cache size. Especially, BA-GC exhibits a good I/O performance when the size of the buffer cache is small. For example, for FAB and BPLRU to achieve a similar improvement compared to BA-GC, they require 16 MB buffer cache while BA-GC only needs 2-4 MB buffer cache.

7. CONCLUSION

We have presented a buffer-aware garbage collection technique called BA-GC which exploits the contents of the buffer cache to reduce the garbage collection cost. The buffer-aware block merge (BA-BM) technique improves the efficiency of the block merge by reducing the number of unnecessary page migrations that turns out to be useless if we consider the contents of the buffer cache. The buffer-aware victim block selection (BA-VBS) technique improves the overall I/O performance by selecting a proper victim block considering the potential benefits of BA-BM.

As a future work, we will try to refine the parameter values used in this work. We have observed from the experiments that different combinations of BA-GC parameters have a relatively large effect on the overall I/O performance. Therefore, it would be an interesting future work to extend the proposed BA-GC scheme.

8. ACKNOWLEDGEMENTS

This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the National Research Lab. Program funded by the Ministry of Education, Science and Technology (No.R0A-2007-000-20116-0). This work was also supported in part by the Brain Korea 21 Project in 2008. The ICT at Seoul National University provides research facilities for this study.

9. REFERENCES

[1] 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,” Journal of Systems Architecture, vol. 53, no. 9, pp. 644-658, 2007.

[2] 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 Transactions on Consumer Electronics, vol. 48, no. 2, pp. 366-375, 2002.

[3] S. W. Lee, D. J. Park, T. S. Chung, W. K. Choi, 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.

[4] J. U. Kang, H. Jo, J. S. Kim, and J. Lee. “A superblock-based flash translation layer for NAND flash memory,” in Proc. International Conference on Embedded Software, pp. 161-170, 2006.

[5] M. Rosenblum and J. Ousterhout. “The design and implementation of a log-structured file system,” ACM Transactions on Computer Systems, vol. 10, no. 1, 1992.

[6] 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 Transactions on Consumer Electronics, vol. 52, no. 2, pp. 485-493, 2006.

[7] S. Y. Park, D. Jung, J. U. Kang, J. S. Kim, and J. Lee. “CFLRU: a replacement algorithm for flash memory,” in Proc. of International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 234-241, 2006.

[8] H. Kim and S. Ahn “BPLRU: A buffer management scheme for improving random writes in flash storage,” in Proc. of USENIX Conference on File and Storage Technologies, pp. 239-252, 2008.

[9] Y. H. Bae. “Design of a high performance flash memory-based solid state disk,” Journal of Korean Institute of Information Scientists and Engineers, vol. 25, no 6, 2007.

[10] S. Lee, D. Shin, and J. Kim, “Buffer-aware garbage collection techniques for NAND flash memory-based storage systems,” Technical Report TR-CARES-07-08, Seoul National University, 2008. http://cares.snu.ac.kr/download/tr-cares-07-08.pdf