An Effective Flash Memory Manager for Reliable Flash Memory Space Management

Han-joon KIM†, Regular Member and Sang-goo LEE†, Nonmember

SUMMARY We propose a new effective method of managing flash memory space for flash memory-specific file systems based on a log-structured file system. Flash memory has attractive features such as non-volatility and fast I/O speed, but it also suffers from inability to update in situ and from limited usage (erase) cycles. These drawbacks necessitate a number of changes to conventional storage (file) management techniques. Our focus is on lowering cleaning cost and evenly utilizing flash memory cells while maintaining a balance between these two often-conflicting goals. The proposed cleaning method performs well especially when storage utilization and the degree of locality are high. The cleaning efficiency is enhanced by dynamically separating cold data and non-cold data, which is called 'collection operation.' The second goal, that of cycle-leveling, is achieved to the degree that the maximum difference between erase cycles is below the error range of the hardware. Experimental results show that the proposed technique provides sufficient performance for reliable flash storage systems.

key words: flash memory, logging, cleaning algorithm, cycle leveling, data collection

1. Introduction

Flash memory is non-volatile solid-state memory with a number of powerful features. Recently, its density and I/O performance have improved to a level at which it can be used not only as auxiliary storage for mobile computers but also as mass storage in a general computing system [1]–[4]. Flash memory is as fast as DRAM in reading operations and much faster than a hard disk in writing*. Also, flash memory is about 30% smaller than an equivalent DRAM cell, yet it is more scalable than DRAM due to its simple cell structure and insensitivity to cell failure. This feature enables the creation of a large memory space. Furthermore, data in a flash memory-based storage system can be processed directly without the need to be brought into main memory. In other words, flash memory space can be used as an extended part of the main memory. In these respects, flash memory has prospective features useful to a storage system which enables it to provide much better performance than conventional memory.

Although flash memory has numerous advantageous features, it cannot be a complete solution for storage systems. Before we employ flash memory as the primary persistent storage device, we have to consider its two critical drawbacks. First, flash memory cannot be updated in situ; its contents should be erased before new data can be stored. An ‘erase’ operation that requires more time than a read or write operation (about 0.6–0.8 seconds) resets the memory cells with either all ones or all zeros. This is because flash memory technology only allows the toggling of individual bits or bytes in one direction for writes. The new free space must have been created by an erase (or cleaning) operation beforehand. Thus, updates to flash memory are slower than updates to conventional memory. The second drawback is that the number of rewrite operations allowed on each memory cell is limited. Typically, the cycling limit is between 100,000 and 1,000,000 erases, depending on the manufacturer [5],[6]. A flash region that approaches its cycling limit will experience frequent write failure; each chip cannot be guaranteed to program and erase within specific time frames although existing data will still remain readable. Thus, when all the memory blocks are not evenly utilized, the size of available memory space decreases rapidly. Therefore, this has a critical effect on the system’s lifetime (or the mean-time-to-failure), so it becomes a great obstacle in developing a ‘reliable’ flash memory-based storage system that is as highly reliable as a disk-based system.

In order to solve the first drawback (inability to update in situ), we must consider how to update data on flash memory blocks and how to manage the free memory space. For updates, it is necessary to de-couple write and erase operations; in other words, an update should be implemented as a write of the new data at new free space and invalidation of the old version at the original location. In addition, since the size of available space is limited, a flash memory system requires a cleaning operation or garbage collection that incurs a certain amount of cost to reclaim the invalidated space within the segments for future usage. Moreover, the cleaning cost should be minimized.

The second drawback requires the flash storage system to wear down all memory blocks as evenly as possible; this process is called ‘cycle-leveling’ or ‘wear-leveling.’ As the amount of flash memory space gets larger, cycle-leveling becomes more essential [3]. The

---

*It has been reported that read operations in flash memory take about 85–100 ns, and write operations take 6–9 µs per byte [5],[6].
basic principle of cycle-leveling is that frequently updated data is put on the less utilized memory blocks. Thus if cycles are too skew over segments, a large number of data should be rewritten at other less utilized regions for cycle-leveling.

Consequently, for the flash storage system, we have to solve several issues related to the cleaning operation and cycle-leveling. The problem is that the objective of minimizing cleaning cost conflicts with that of cycle-leveling. If flash memory management focuses on lowering the cleaning cost, cycles are skew over segments, and so the system’s reliability is dramatically affected. On the contrary, excessive cycle-leveling generates a large number of invalidated blocks, which degrade cleaning performance.

In this paper, we describe a new way of memory management for flash storage systems that accommodate effectively these two often-conflicting objectives. As a flash space management strategy, we adopt append-only logging (or logging) proposed in log-structured file systems (LFS) [7], since free space should be claimed before write operations. (In append-only logging, all updates are carried out on a sequential structure called a ‘log.’) Based on the cleaning mechanism of the LFS, we propose a mechanism that can reduce cleaning cost and evenly utilize flash memory cells. The cleaning efficiency is enhanced through a ‘collection’ operation whose activity is to collect cold data (infrequently updated) that are fragmented over the log. The conflict between cleaning and cycle-leveling is resolved by integrating cycle-leveling into the cleaning process through a special cleaning criterion for selecting victim segments.

The remainder of this paper is organized as follows. In Sect. 2, we state how free space is managed under the logging strategy for flash memory and present performance metrics and related goals. In Sect. 3, we introduce a way of achieving the presented goals along with its analytic discussion. Section 4 presents performance results. Section 5 discusses related work. Section 6 concludes the paper.

2. Flash Memory Management

As a basic strategy for flash memory space management, we adopt append-only logging as in the log-structured file system. In previous works such as in [3], [8], [11], a logging approach has been recommended for managing flash memory since logging is quite effective in handling flash memory in several ways. First, logging automatically solves the inability to update in situ since an update results in a new write at the end of the log and invalidation of the old. The natural separation of asynchronous erases (cleanings) from writes allows write operations to fully utilize the fixed I/O bandwidth, and thus prevents performance degradation that may occur when writes and erases are performed simultaneously. In addition, logging has the positive side effect of cycle-leveling because append-only writes lead to sequential use of unused space. More importantly, our work can use a cleaning mechanism proposed in LFS since a flash storage system should perform an erase operation for generating new free space.

In order to support logging, the flash memory space is partitioned into segments. Each segment is a set of erase blocks that are logically clustered to be erased together in a single erase operation. In each segment, a fixed number of blocks are organized; the block is the unit of I/O operation, which is the amount of data transferred between the flash memory and main memory. The term ‘segment’ is analogous to the segment defined as a unit of cleaning operation in Sprite-LFS [7]. However, the size of each segment in the flash storage system should be sufficiently large since an erase operation takes a fixed amount of time regardless of the number of memory cells being erased.

In this section, we first describe a prototype of flash storage system, and then present our goals for the management of flash memory space.

2.1 Architecture

The flash storage system proposed in this paper is illustrated as Fig. 1. First, the File Manager is responsible for determining and maintaining files as a set of logical blocks. Next, the Flash Memory Manager manages the physical flash memory; it determines the physical location of a logical block, and issues the necessary flash I/O operation. The activity of the flash memory manager is similar to that of a disk manager.
in disk-based storage systems. Finally, the Flash Media Interface involves a set of low-level operations for flash memory array, including erasing, programming, and error detection. Among these components, our work is mainly concerned with the Flash Memory Manager.

The Flash Memory Manager is composed of three major components: Allocator, Cleaner, and Cycle-Leveler. The Allocator is responsible for keeping a pool of free segments. It decides which of the free segments is to be assigned next. The Cleaner reclaims invalid blocks of segments to generate new free space. Finally, the Cycle-Leveler is responsible for maintaining an even distribution of erase cycles over the flash segments.

The Flash Memory Manager works as shown in Fig. 2. First, data is newly written to the blocks in a segment, and for updates, the current block containing the old data is marked ‘dead’ and the new data is written to a new block, which is then marked ‘live.’ As this process goes on, there will be lots of dead blocks in the segments of the log, and the number of available fresh segments will decline. When the number of available fresh segments gets below some threshold, the Cleaner is triggered. The cleaning operation on the flash memory log works in two phases: the rewriting and the erasing phases.

In the rewriting phase, the Cleaner first selects a segment in the log (1), allocates a segment (2), collects valid data from the selected segment and rewrites the data onto the end of log (3). In this phase, the choice of segments assigned by the Allocator to the log has an effect on the cleaning and cycle-leveling efficiency (refer to Sect. 3.3.2). Then, in the erasing phase, the erase operation is performed in parallel for all erase blocks in the selected segment. The cleaned segment is reclaimed into the free segments pool (4). Additionally, the Cleaner checks whether segments are evenly cycled. If the degree of skewness in cycles is higher than a given criterion, the Cleaner activates the Cycle-Leveler.

2.2 Problem Definition and Performance Metrics

In order to quantify the efficiency of our methods, we first need to present performance measures in terms of cleaning cost, the degree of cycle-leveling, and the system’s lifetime. Using these proposed measures, we present our aims of flash memory management.

Our cleaning cost model is based on the cleaning cost formula defined according to the eNVy flash storage system [2]. The cleaning cost in [2] is defined as \( \frac{u}{1-u} \), where \( u \) is the utilization of a segment (the fraction of space occupied by live data). That is, the cleaning cost is the amount of data that need to be rewritten \((1-u)\) per unit of new space claimed \((1-u)\). This cost model reflects the fact that only the cost of rewriting the valid blocks in a segment is considered since seek time and rotational delay addressed in disk storage are not applicable in flash memory and the read speed is much faster than the write speed.

However, this cleaning cost does not consider the fact that in flash memory, the cleaning is accompanied by a very expensive erase operation. Even though erase operations incur a fixed cost at every cleaning, frequent cleaning over a fixed period results in higher overall costs. Thus, the Cleaner should minimize the number of erase (cleaning) operations. In particular, it is very essential to reduce the utilization of the segment to be cleaned when storage utilization is high. The reason is shown in the following lemma.

**Lemma 1:** Let \( W \) denote the number of blocks of the write requests after cleaning starts, let \( u_c \) denote the utilization of the selected segment for cleaning, and let \( SegSize \) be the number of blocks per segment. Then, an approximation of the cleaning frequency \( f \) is given by \( \frac{W}{(1-u_c)SegSize} \).

**Proof:** see Appendix A.

As shown in Fig. 3, the value of \( f \) sharply increases...

![Fig. 3 Cleaning frequencies from varying utilization of the segment to be cleaned.](image-url)
as $u_i$ is higher, which means that minimizing the utilization of the segment to be cleaned leads to significantly lower the cleaning frequency especially when the utilization is high.

Obviously, the cleaning cost model in flash memory management should address the number of erase (cleaning) operations as well as the pure cleaning (writing) cost. First, we need to express the erase cost per segment ($EraseCost$) in the context of the writing cost per segment ($WriteCost$) since the two cost values are not additive to each other. It is known that the time to erase a whole segment is nearly proportional to the time to write a whole segment [1]. Thus, we assume that there exists a ratio $k_u$ that can express $EraseCost$ according to the scale of $WriteCost$. Then, $EraseCost = \omega \cdot k_u$, where $\omega$ is a (fixed) erasure cost (e.g., time cost) measured with a certain criterion. As a result, the total cleaning cost can be described as follows:

$$CleaningCost = f \cdot (\omega \cdot k_u) + \sum_{i=1}^{f} \frac{u_i}{1 - u_i} \tag{1}$$

where $f$ is the number of cleanings over a fixed amount of the write request stream, and $u_i$ is the utilization of the segment selected for the ith cleaning. However, this cost formula is not practical due to difficulty in computing $k_u$. Thus, as an alternative to Eq. (1), we use only the second term of Eq. (1), $\sum_{i=1}^{f} \frac{u_i}{1 - u_i}$, as the flash cleaning cost. This is because Eq. (1) has almost the same mathematical behavior as $\sum_{i=1}^{f} \frac{u_i}{1 - u_i}$ regardless of the value of $k_u$ (see Appendix B). Intuitively, the alternative formula represents the erasure cost as the degree of how frequently the cleaning cost at each cleaning time is accumulated.

**Definition 1** (Cumulative Cleaning Cost): The cumulative cleaning cost is $\sum_{i=1}^{f} \frac{u_i}{1 - u_i}$.

Next, as a measurement of how evenly cycles are distributed over all the segments, we use the leveling degree as defined below.

**Definition 2** (Leveling Degree): The leveling degree ($\Delta_x$) is the difference between the maximum erase count ($\epsilon_{\text{max}}$) and the minimum erase count ($\epsilon_{\text{min}}$), i.e., $\Delta_x = \epsilon_{\text{max}} - \epsilon_{\text{min}}$.

Instead of the variance or the standard deviation of erase counts, we choose to use the difference as a measure of the degree of cycle-leveling because the system’s lifetime is dependent on the lifetime of the most worn-out segments.

Generally, perfect cycle-leveling, where the leveling degree is kept at 0 or 1 at all times, need not be supported to assure the reliability of flash storage systems. This is because erase limits vary from segment to segment within the hardware error range, and also excessive cycle-leveling degrades cleaning (writing) performance. Therefore, we attempt to achieve cycle-leveling only to the extent that the leveling degree is within the error range of the erasure limit.

Lastly, we propose a measurement for evaluating the system’s lifetime. We can define the capacity of a flash storage system (flash cycle capacity) as the maximal amount of (block) write requests that the flash space can endure. This is proportional to the total number of segment cycles performed up to the failure point (failure point is when $x\%$ of the space has exceeded its erase limit, where $x$ depends on the system architecture and the workloads in the system). Thus, the used portion of flash cycle capacity up to system failure, which is called the ‘running cycle capacity,’ determines the system’s lifetime.

**Definition 3** (Running Cycle Capacity): The running cycle capacity at time $t$ is $\sum_{i=1}^{N_{\text{seg}}} \epsilon_t(S_i)$, where $N_{\text{seg}}$ is the number of segments in the system and $\epsilon_t(S_i)$ is the erasure count of segment $S_i$ at time $t$.

In relation to the Definition 3, our objective of cycle-leveling is to keep the leveling degree below the error range so that at the time of failure, $t_f$, each of $\epsilon_t(S_i), i=1,\ldots,N_{\text{seg}}$, is within the error range of the erasure limit, in which case, the flash memory would have been used to its fullest erasure capacity (within the error boundary). By increasing this running cycle capacity, we can lengthen the system’s lifetime.

Let us suppose that without cycle-leveling, the system fails after $x$ (segments·erases) are consumed, and with cycle-leveling, the system fails after $y$ (segments·erases) are used. Then, the running cycle capacity has been increased by a ratio $\frac{y}{x}$. Another way to increase the lifetime is to consume fewer erase cycles by reducing the cleaning frequency. If the cleaning frequency is further reduced by the ratio $a$, the lifetime will be increased by a ratio $\frac{y}{x(1-a)}$.

### 3. Optimization of Flash Memory Management

In this section, we first introduce a basic cleaning method mentioned in [7], [12], and then based on this method, incrementally optimize the method of managing the flash memory. The main components of our method include a ‘collection’ operation, cycle-leveling algorithm through a new cleaning index and heuristics for segment allocation.

#### 3.1 Limitation of the Greedy Cleaning Method

The performance of a system that requires a cleaning mechanism usually depends on the algorithm it uses [2], [7]. Furthermore, according to the proposed cleaning
cost model, the cleaning cost of a segment depends strongly upon its utilization. Thus, the most important aspect of the CLEANER is the segment selection policy, based on which the segments that are to be cleaned are selected. Intuitively, we can expect that it is better to select the segment with the least utilization. This is known as the greedy cleaning method [7], which we refer to as Greedy I. However, as mentioned in [2], [7], [8], the greedy cleaning method does not perform well when the degree of locality is high. Especially when the utilization is high, locality has a much bigger impact on cleaning cost.

The primary cause of the increase in cleaning cost with locality is the co-existence of cold and non-cold data within each segment on log. After a large number of logging and cleaning operations, cold data becomes mixed with non-cold data within each segment. After that time, at every cleaning, the cold data fragmented over the log area moves around uselessly together with non-cold data, without further invalidation. For this reason, the utilization of cleaned segments remains stable at a high value. For the same reason, the cleaning frequency increases as well; the segment whose utilization is higher due to cold data is more quickly filled up by writes, so it has more chance of being cleaned.

3.2 Separation of Cold Data from Non-cold Data

In order to overcome the limitation of the greedy cleaning method, we propose a ‘collection’ operation that isolates cold data from non-cold data. Actually, this is implemented as another module, called COLLECTOR of the FLASH MEMORY MANAGER. The COLLECTOR periodically invokes the COLLECTOR, which then collects fragmented cold data over the log.

The COLLECTOR works in two phases: I) search and collect fragmented cold data (file) blocks, and II) cluster them to the end of the log. Figure 4 illustrates the procedure for the collection operation.

In phase I, at each collection time, the COLLECTOR determines the collection size, and searches for cold data that are much fragmented. In the example in Fig. 4, the collection size is 10, and as much fragmented cold data in file ‘A’ and ‘B’ are collected as possible. In phase II, it writes the blocks of that file back to the end of the log until the collection size is reached. If during the process of collection the free space is not enough, the CLEANER can be invoked to generate new space.

After invoking a collection, the segments with collected cold data will be less invalidated rather than other segments, and the segments that are relatively less utilized segments due to collection operation will be are selected for cleaning.

This collection operation must have reasonable policies with regard to 1) which cold data are to be collected, 2) when cold data are to be collected (i.e., collection period), and 3) how much cold data are to be collected (i.e., collection size). As for issue 1, data being collected should be cold data with a high degree of fragmentation over the log. For this, we must identify cold data and measure how it is fragmented. Issues 2 and 3 are crucial factors that affect the efficiency of the COLLECTOR. There is a trade-off between collection size (or collection period) and cleaning cost; that is, too frequent collection or too large amounts of collection would undermine its positive effect. Here, we need to define the terms ‘collection period’ and ‘collection size’ to deal with the above issues.

**Definition 4 (Collection Period):** The collection period is the number of cleaning operations performed until a collection operation is again invoked. If collection period is \( n \), collection operation is invoked once every \( n \) cleaning times.

**Definition 5 (Collection Size):** The collection size is the number of blocks to be collected for cold data collection from the log at each collection time.

3.2.1 Identification of Cold Data

In order to tackle issue 1, the COLLECTOR uses what we call the ‘file access window’ (FAW), which keeps a list of files that are recently accessed, as shown in Fig. 5. This file access window is analogous to a ‘working set’ that is used for virtual memory management. According to the principle of the working set, the same blocks tend to be referenced repeatedly within a working set window (time) with locality [13]. Likewise, we can expect that files kept in FAW will be actively re-accessed for some time as long as locality exists. Thus, since the files in FAW are likely to be updated and then fragmented, the files outside the FAW are likely to be ‘cold.’

Next, among the cold files outside the FAW, higher priority is given to ones that are more fragmented since

![Fig. 4 Procedure of collection operation: Blocks to be appended are marked as the lower-case letter of the file identifier.](image-url)
unfragmented cold data is unnecessary to collect. Thus, we have to check what extent the data is spread over the log. To do this, we estimate the degree of fragmentation of each candidate file. The fragmentation degree $F$ (whose range is between 0 and 1) is defined as follows:

$$F = \frac{s}{b} \quad \text{(if } b > L_{\text{seg}}, \text{ then } b = L_{\text{seg}})$$

where $b$ is the number of blocks occupied by the file, $s$ is the number of segments over which the file is fragmented, and $L_{\text{seg}}$ is the number of segments in the log. If $b$ is larger than $L_{\text{seg}}$, $b$ is replaced by $L_{\text{seg}}$ since $s$ cannot be larger than $L_{\text{seg}}$. This avoids the problem that for the same $s$ value, the fragmentation degree of a large file is relatively small compared to that of a small file. Depending on this fragmentation degree, the Collector decides whether the candidates are to be collected; it selects, among the files not in the FAW, ones with higher fragmentation degree.

To identify cold data, we choose to use files instead of blocks since it is impossible to identify the degree of fragmentation at the block level. Thus, the mapping table between a file and its blocks is necessary, which is maintained in the file manager (see Fig. 4) and is referenced to locate the blocks of the file.

### 3.2.2 Collection Size and Collection Period

As cold data grows and becomes fragmented, collection is required more often. Basically, the appropriate time for collection operation is when cold data is fragmented over the log so much that it degrades the cleaning performance. Also, the collection size should be less than that of all the fragmented cold data. Unfortunately, the problem of deciding the collection size (or collection period) requires a fine-tuning knob that is chosen on an ad hoc basis. Thus, we propose a simple heuristics based on utilization and locality.

First, we set the collection size as the following formula:

$$\text{ColSize} = k_s \cdot u_{\text{avg}} \cdot l_f \cdot N_{\text{seg}} \cdot \text{SegSize} \quad \text{(3)}$$

where $u_{\text{avg}}$ is the average utilization, $N_{\text{seg}}$ is the number of segments in flash memory space, $\text{SegSize}$ is the number of blocks in a segment, $k_s$ is a constant between 0 and 1 which are experimentally adjusted so as to achieve good cleaning efficiency, and $l_f$ denotes the degree of reference locality, whose range is between 0 and 1.

The collection size is determined to be (nearly) proportional to the average utilization. This is because when locality exists, high utilization yields more cold data, which is more likely to be fragmented by continuous logging and cleaning operations. Cold data is more likely to be fragmented as the reference locality of data increases. Thus, when locality is high, the collection size needs to be set larger. By adding other factors, we may obtain a more precise collection size.

To make this collection policy complete, we need to define the degree of locality. Since in a log-based storage system the memory locations of blocks in files change, we measure the degree of locality using the file access window as follows:

$$l_f = \frac{|\text{FAW}| - |\text{FAW}|_f}{|\text{FAW}| - 1} \quad \text{(4)}$$

where $|\text{FAW}|$ is the length of the file queue in the FAW, and $|\text{FAW}|_f$ is the number of distinct files in the FAW. Intuitively, we can measure the degree of locality as $|FAW|_f$, but in order to map this to values between 0 and 1, we use Eq. (4) above. In an extreme case, if only one file is accessed for some time, that is, $|FAW|_f$ is 1, becomes 1.

**Lemma 2**: Let $u_0$ denote the utilization of the segment to be cleaned just before collection, let $k_s$ be the constant in Eq. (3), and let $l_f$ denote the degree of locality. Then, when using the greedy cleaning method, the utilization of the segment selected for collection is reduced to $u_0(1 - k_s \cdot l_f)$.

**Proof**: see Appendix C.

As stated in the above theorem, the utilization value sharply drops by a ratio $k_s \cdot l_f$ due to collection. Then it will gradually grow large and return to the value of $u_0$, since the cold data clustered is again evenly fragmented over the log during continuous cleaning operations. In any case, the cleaning cost at each cleaning time and the cleaning frequency are reduced until the utilization of cleaned segments again level off.

Similarly to the collection size, the collection period may also be determined according to the average utilization and the degree of locality; that is, it may be set to be (nearly) proportional to the average utilization, and as locality is higher, it needs to be set smaller. Fortunately, given the collection size, the collection period can be determined by the following lemma.

**Lemma 3**: Let $ColSize$ and $SegSize$, respectively denote the collection size and segment size, let $l_f$ denote the degree of locality, and $u_{\text{col}}$ denote the utilization of the segment selected for cleaning after performing a collection. Then, when using the greedy cleaning method,
an approximation of the collection period is given by
\[
\frac{ColSize}{(1-l_f)(1-u_{col})SegSize}.
\]

**Proof:** see Appendix D.

We shall refer to the algorithm that employs the above collection operation on top of the greedy method as CICL I.

### 3.2.3 Modification to Cleaning Cost

The previously defined cleaning cost needs to be modified to accommodate the cost (named ‘collection cost’) incurred by the collection operation which also requires rewriting valid data as with the cleaning operation. The cumulative cleaning cost is modified as:

\[
c = \frac{ColSize}{SegSize \times ColPeriod} \sum_{i=1}^{f} \left( \frac{u_{col,i}}{1-u_{col,i}} + \frac{c}{1-u_0} \right),
\]

where \( f \) is the cleaning frequency, \( u_{col,i} \) (resp. \( u_0 \)) denotes the utilization of \( i \)th selected segment for cleaning after performing collection (resp. just before performing collection), and \( \frac{c}{1-u_0} \) represents the collection cost, which is the additional cost required to generate \((1-u_0)\) of fresh space. This value is best explained as the cost required if one collection were to be performed at each cleaning time; in reality, only after as many cleaning operations as collection periods are executed, one collection occurs. Thus, the numerator \( c \) of the collection cost can be computed as Eq. (5), where \( SegSize \) is the number of blocks in a segment. For example, suppose \( SegSize \) is 500 blocks, and 2,000 blocks are rewritten for a collection when \( ColPeriod \) is 100, then \( c \) is \( \frac{2,000}{500} \times \frac{1}{100} = 0.04 \); that is, a cost of as much as 0.04 should be added to each of the 100 cleaning operations in order to account for the cost of one collection.

3.2.4 Analysis of Collection Operation

In this section, we will discuss the collection operation by using analytic methods. For this, we first define so-called ‘collection gain,’ which means the extent of gain obtained from a single collection operation.

**Definition 6 (Collection Gain):** The collection gain is \( \sum_{i=0}^{p} \frac{u_{col,i} - \frac{u_{col,i} \times (1-u_{col,i})}{u_0}}{u_0} \), where \( u_0 \) denotes the utilization of the selected segment for cleaning just before performing collection, \( u_{col,i} \) denotes the utilization of the segment selected for the \( i \)th cleaning after performing collection, \( c \) is the value of Eq. (5), and \( p \) denotes the collection period.

In this definition, it is assumed that the utilization of the selected segment for cleaning after \( p \) previous cleanings, i.e., \( u_{col,p} \) is equal to \( u_0 \). In fact, this assumption is found to be reasonable through our experiment. With this assumption and Lemma 3, the following theorem can be derived, which indicates when the collection operation can give a positive effect on the cleaning performance.

**Theorem 1:** If \((l_f > u_0 \text{ and } l_f > 1-u_0)\) or \((l_f \leq u_0 \text{ and } l_f > \frac{u_0}{u_0-1} + 1)\), then the collection gain is positive, where \( u_0 \) is the utilization of the segment selected for cleaning just before performing collection, and \( l_f \) denotes the degree of locality.

**Proof:** see Appendix E.

**Fig. 6** Two graphic views of Theorem 1.
the collection gain is positive only if locality increases with decreasing $u_0$. Also, when locality is sufficiently high (say, over 0.5), the collection gain increases nearly by geometric progression as $u_0$ gets higher as shown Fig. 6 (b). This analytic result of the collection operation is very encouraging since localized access patterns and high storage utilization are commonly observed in the real-world systems.

3.3 Cycle-Leveling

For any pattern of data access, uneven distribution of erasures tends to occur. In the worst case, if there are static data that are not accessed at all, severe skew in cycles will occur. Although the logging strategy has a potential effect on cycle-leveling, the flash system needs a refined leveling method, especially under write-intensive environments.

3.3.1 Cleaning Index

Cleaning index is the criterion for selecting segments to be cleaned; the segment with the lowest index value\(^1\) is selected at cleaning time. In our work, a special cleaning index is defined for both lowering the cleaning cost and promoting cycle-leveling as follows:

$$\text{Cleaning Index} = (1 - \lambda)u_i + \lambda \left( \frac{\epsilon_i}{\epsilon_{\text{max}} + 1} \right)$$

where $u_i$ and $\epsilon_i$, respectively, are the utilization and the erase count of segment $i$, and $\epsilon_{\text{max}}$ is the maximum of erase counts. The cleaning index is composed of two factors: utilization and erase count of each segment. These two factors are weighted by so-called ‘normalized leveling degree,’ denoted as $\lambda$, which maps the leveling degree to values between 0 and 1.

If the cycles are evenly distributed, the cleaning index is used for lowering the cleaning cost. On the contrary, when there is severe skew in the cycles, the cleaning index works for cycle-leveling. In other words, at normal times when the leveling degree is below a tolerable threshold (the value of $\lambda$ approaches 0), the utilization $u_i$ has to be the chief factor in selecting a segment. When the leveling degree exceeds the tolerable limit (the value of $\lambda$ approaches 1), then the cleaning index should be calculated either considering both factors, or being biased toward cycle-leveling.

Here, in order to prevent excessive cycle-leveling, the weighting variable $\lambda$ needs to affect the cleaning index only after there is more than a certain amount of cycle skewness, so that a small measure of skew in cycles can be ignored. Therefore, $\lambda$ is represented by a form of sigmoid function, which increases monotonically with the leveling degree, $\Delta_\epsilon$ (see Fig. 7).

$$\lambda = \frac{2}{1 + e^{\frac{\epsilon}{\epsilon_{\text{max}} + 1}}}$$

where $k_\epsilon$ is a constant that determines the steepness of this function. Figure 7 shows that as $k_\epsilon$ gets larger, the value of $\lambda$ increases more slowly in the initial stage. Thus, the constant can be used for controlling the degree of cycle-leveling. Should we wish to loosely achieve cycle-leveling, we have only to make $k_\epsilon$ larger. For example, when $k_\epsilon$ is small, say 10, the value of $\lambda$ rapidly increases as the leveling degree is small. Then, the cleaning index is computed considering the ‘erase count’ factor, which will result in near-perfect cycle-leveling. When $k_\epsilon$ is greater than 500, the CLEANER almost does not care about cycle-leveling until $\Delta_\epsilon$ is close to the point where the curve begins to rise steeply. Like this, cycle-leveling is achieved within the CLEANER through a special cleaning index. We refer to the method that integrates this leveling method into CICL I as CICL II.

3.3.2 Segment Allocation Policy

In terms of the Allocator module, cycle-leveling can be assisted by carefully selecting the next fresh segment from free space. For this, segment allocation by the Allocator considers the following two cases: 1) when appending data to the log for generating new free space, and 2) when collecting cold data and rewriting to the log.

In case 1), a segment with the lowest erase count is selected for replacement to increase its usage. The new segment can be expected to reach its next cleaning cycle earlier because of its hot data. Correspondingly, in case 2), a new segment with the highest erase count is allocated to the log. The idea behind this heuristics is that a large amount of cold data will be clustered onto the segment allocated for a collection operation, and thus the segment will experience less invalidation than the other segments. The method that uses this policy on top of CICL II will be referred to as CICL III.

\(^1\)In the case of the cost-benefit policy introduced in Sprite-LFS, the segment whose cleaning index ($\frac{\text{age} \times (1-u)}{u}$) is the greatest is selected [7].
4. Performance Analysis

4.1 Simulator and Workloads

We have built a simulator with 1 Gbyte of flash space that is divided into 64 flash segments. Each segment is composed of 256 erase blocks. The cleaning time in Table 1 indicates that at every 100 write requests, the CLEANER checks whether the ratio of the number of free segments to the number of total segments is below 0.1, in which case one segment is cleaned. The Collector chooses among the files not in the FAW, the cold ones with a fragmentation degree that exceeds 0.8. The constant \( k_c \) in Eq. (7) is set below 100 to achieve cycle-leveling. The other details pertaining to flash memory space and the Flash Memory Manager are described in Table 1. As for metadata, the segment usage information required by the Collector (including the number of valid blocks and the erase count of each segment) and the file-block mapping table maintained by the File Manager are kept as internal tables in the simulator without being explicitly stored in flash memory.

In our work, for reliable performance evaluation, we have synthetically generated two types of workload, named UNIX and LFS. The UNIX workloads are based on the size distribution of UNIX files described in [14], where the size of cold data is larger than that of hot data. The LFS workloads, on the other hand, are characterized by frequent modification to small data files (4 KB files); LFS (90 → 10) workload is the same as the ‘hot-and-cold’ workload used in [7].

Since our proposed method affects only updates and new write operations, read traffic is not modeled in our workload. Moreover, in each workload, write access exhibits certain amounts of reference locality. In this paper, to denote the degree of locality, we use the notation similar to that of [1]; that is, ‘\( x \rightarrow y \).’ This means that \( x \)% of write accesses go to \( y \)% of the data, and the other \((100 - x)\)% goes to \((100 - y)\)% of the data and thus ‘50 → 50’ means no locality. Moreover, to represent unchanging (static) data, we specify one more parameter for static data. For example, ‘\( x \rightarrow y, z \)' means that \( x \)% of write accesses go to \( y \)% of the data, and the other \((100 - x)\)% of the references go to \( z \)% of the data with \((100 - y - z)\)% of data unchanged.

![Image](82x40 to 514x108)

**Image 82x40 to 514x108**

4.2 Simulation Results

Since our focus is on lowering the cleaning cost and simultaneously achieving satisfactory cycle-leveling, we use different greedy methods with a simple cycle-leveling as a baseline for performance measurement. In a simple bid to promote cycle-leveling, when allocating a free segment to the log, the least worn-out segment can be chosen when the maximum difference among cycles is beyond a threshold value. Greedy II uses this policy on top of Greedy I. Another cycle-leveling solution is to swap the data in the least worn-out segment and the most worn-out segment. This swapping makes the former segment have a better chance of being erased at cleaning time. Greedy III is a method that integrates this swapping solution with Greedy I.

In addition to these greedy methods, to compare our methods with other log-based systems in terms of only the cleaning cost, we will evaluate the cost-benefit method (Cost-Benefit) proposed in Sprite-LFS. In Table 2, we enumerate the various methods of managing flash space described above.

![Image](82x109 to 514x120)

**Image 82x109 to 514x120**

### Table 1 Simulation parameters.

<table>
<thead>
<tr>
<th>Flash Memory Space Parameters</th>
<th>Flash Memory Manager Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash memory type</td>
<td>1 Mbit × 8 bytes</td>
</tr>
<tr>
<td>Size of flash memory</td>
<td>1 Gbyte</td>
</tr>
<tr>
<td>Number of segments</td>
<td>64</td>
</tr>
<tr>
<td>Number of erase blocks per segment</td>
<td>256</td>
</tr>
<tr>
<td>Block size</td>
<td>512 bytes</td>
</tr>
<tr>
<td>Identification of cold data</td>
<td>Free flash space</td>
</tr>
<tr>
<td>Cleaning period</td>
<td>Free flash space</td>
</tr>
<tr>
<td>Every 100 write requests</td>
<td>Total flash space ≤ 0.1</td>
</tr>
<tr>
<td>Cleaning time</td>
<td>1 segment</td>
</tr>
<tr>
<td>Collection size</td>
<td>(</td>
</tr>
<tr>
<td>Leveling degree</td>
<td>( k_c = 0.1–0.3 )</td>
</tr>
<tr>
<td></td>
<td>( k_c = 100 )</td>
</tr>
</tbody>
</table>

The total number of write requests is about 10 millions and the number of data items generated from them is about 2,000 files; initially, the write requests generate the files, which are then repeatedly overwritten or deleted. The detailed procedure to develop the UNIX(\( x \rightarrow y \)) workloads with locality of reference is as follows. Firstly, the initial files are generated based on the size distribution of UNIX files. The generated files are then arranged in ascending order of size. Then \( x \)% of writes are requested, in a pseudo-random number with a dynamic seed value, over a set of relatively small-sized files covering the first \( y \)% blocks of the sorted files. Lastly, the write position within a selected file is also pseudo-randomly decided. The development of LFS workloads is the same as that of UNIX workloads except that they do not require the file ordering.

4.2.1 Experiment 1: Tuning Collection Operation

The performance of the Collector is determined by the size of the FAW, the collection size, and the collection period. The FAW does not fulfill its function if its size is too large or too small. Figure 8 illustrates the cumulative cleaning cost for different workloads when...
using CICL I. As shown in this figure, when the size of the FAW is around 200, the system shows the smallest cleaning cost. Thus, most of our experiments assume a size of 200 for the FAW.

In addition, the Collector responds sensitively according to the collection size. Figure 9 shows the cumulative cleaning cost according to changes in collection size when using CICL I. From this figure, we can see that in order for collection to reduce the cleaning cost, the collection size should be greater than some threshold, but excessive cold data causes an increase in total cleaning cost. In our experiments, based on Fig. 9, $k_s$, which determines the collection size, is set to 0.1–0.3. Then using the collection size, the collection period is determined by Lemma 3.

### 4.2.2 Experiment 2: Cleaning Cost

Figure 10 shows the changes in utilization of the selected segment to be cleaned. In the case of Greedy I, the utilization of cleaned segment first increases sharply, but after having processed a certain number of cleaning cycles, it stays almost constant. In [12], the phenomenon was analytically proved for the case of uniform reference of data. The curve of utilization of cleaned segment approaches a plateau near the time it takes for cold data to be fragmented over all segments in the log. In the case of CICL I, the utilization of cleaned segment and the cleaning frequency are reduced, and the convergence value is also suppressed compared to Greedy I. On average, the cumulative cleaning cost of CICL I is 30% smaller than that of Greedy I. In addition, we can also see that the utilization of cleaned segment undergoes fluctuation. This is why cold data is clustered by the collection operation, and is then re-fragmented by continuous logging or cleaning.

Figure 11 shows the cumulative cleaning cost according to changes in average utilization when using Greedy I and CICL I. In this figure, we can observe that the benefit of collection is bigger as utilization and degree of locality are higher.

Figures 12 and 13 illustrate the cumulative cleaning cost incurred by different methods for each workload. In most workloads, our method has significant benefits over greedy methods. Note that particularly when locality exists and segment utilization is high, our

<table>
<thead>
<tr>
<th>Methods</th>
<th>Cleaning index</th>
<th>Separation of cold data</th>
<th>Segmentation allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy I</td>
<td>$u$</td>
<td>Not Available</td>
<td>Not Available</td>
</tr>
<tr>
<td>Greedy II</td>
<td>$u$</td>
<td>Not Available</td>
<td>Most worn-out segment allocation</td>
</tr>
<tr>
<td>Greedy III</td>
<td>$u$</td>
<td>Not Available</td>
<td>Data swapping</td>
</tr>
<tr>
<td>Cost-Benefit</td>
<td>$(1 - \lambda)u + \frac{\epsilon^i}{\epsilon_{\max} + 1}$</td>
<td>Collection (refer to Sect. 3.2)</td>
<td>Most worn-out segment allocation</td>
</tr>
<tr>
<td>CICL I</td>
<td>$u$</td>
<td>Age-sorting</td>
<td>Not Available</td>
</tr>
<tr>
<td>CICL II</td>
<td>$(1 - \lambda)u + \frac{\epsilon^i}{\epsilon_{\max} + 1}$</td>
<td>Collection (refer to Sect. 3.2)</td>
<td>Most worn-out segment allocation</td>
</tr>
<tr>
<td>CICL III</td>
<td>$(1 - \lambda)u + \frac{\epsilon^i}{\epsilon_{\max} + 1}$</td>
<td>Collection (refer to Sect. 3.2)</td>
<td>Allocation heuristics (refer to Sect. 3.3.2)</td>
</tr>
</tbody>
</table>

**Table 2** Comparison of various flash space management methods.
methods (CICL series) outperform the greedy methods (Greedy series) in terms of cleaning cost and leveling degree. For example, by paying a collection cost of only about 8% of the total cleaning cost in the UNIX (95 → 5) workload, CICL III reduces the cumulative cleaning cost by about 35% compared to Greedy I and Greedy II.

In CICL II, the drop in cleaning cost achieved by collection is less visible due to cycle-leveling. However, CICL III regains the lost cost reduction effect as a result of the proposed segment allocation policy. In the UNIX(95 → 5) workload, while CICL III levels erase cycles more strictly than CICL I and CICL II, CICL III lowers the cleaning cost further than the other CICL methods.

In the UNIX (90 → 10, 80) workload that includes unchanging data, the cleaning cost reduction due to collection is smaller than in other workloads, since the data are less fragmented. Also, in the LFS workloads, our methods show smaller cost reduction compared to the UNIX workloads. Since an LFS workload contains only small files, less cold data in segments moves around than that in the UNIX workload.

The Cost-Benefit method incurs a high overhead in our system although it is known to have a good performance in disk-based storage systems. This is because the size of segments used in a flash storage system are much larger than in a disk-based storage system. According to the cost-benefit policy, segments which have less utilization and greater age are selected as victims at cleaning time. However, even if blocks are arranged according to their age, the age values within large segments are not usually consistent, which results in a failure to demonstrate the capability of the method.

4.2.3 Experiment 3: Cleaning Frequency

In order to evaluate how much the cleaning frequency can be reduced regardless of the size of a given write requests, we have measured so-called ‘inter-cleaning write capacity,’ which is computed by dividing the size of the write request stream by the number of cleaning operations. This measure provides the write capacity between cleanings. That is, a larger value indicates that for a given amount of writes, cleaning is delayed, and erase commands become less frequent.

Figures 14 and 15 illustrate that when there is locality of reference, our methods have larger inter-cleaning write capacity than the greedy methods in many cases. However, CICL III shows smaller write capacity in the LFS(80 → 20), and LFS(95 → 5) workloads. Also, in the UNIX(80 → 20) workload, CICL
series perform worse than greedy method. Here, note that small inter-cleaning write capacity does not necessarily mean large cumulative cleaning cost. When the Collector works very effectively, that is, collects an optimal amount of cold data at an optimal time, the cleaning cost incurred by each cleaning can become very small. At this time, although the inter-cleaning write capacity is small, the cumulative cleaning cost is not very large.

4.2.4 Experiment 4: Cycle-Leveling

Figures 16, 17, and 18 show changes in the leveling degree as cleaning operations proceed for different workloads with locality. Some fluctuating graphs with a small value of leveling degree represent the process of cycle-leveling.

In most workloads, CICL II and CICL III allow $\Delta_t$ to fluctuate within a given error range. For workload UNIX (90 → 10, 80) in particular, CICL II and CICL III provide stable leveling effects although these methods have a smaller cleaning cost, while Greedy I and CICL I show severe skew in cycles. Greedy II outperforms Greedy I, but its effect is not satisfactory. Greedy III also has a leveling effect similar to CICL methods, but among all the methods with the exception of Cost-Benefit, Greedy III has the worst cleaning efficiency; this is because a large amount of write cost is required to swap data between the youngest and oldest segments.

Also, these figures suggest that the segment allocation policy that belongs to CICL III is helpful in cycle-leveling compared to the simple policy of selecting the segment with the lowest erase count; this policy in effect contributes to lowering the cleaning cost as well. Moreover, CICL I, which only employs collection without cycle-leveling, also shows some degree of cycle-leveling effect. This is because the collection operation invalidates the blocks in the segments that contain cold data, which makes these segments more likely to be cleaned.

As mentioned in Sect. 3.3.1, the constant $k_\epsilon$ included in the proposed cleaning index can be used for controlling the degree of cycle-leveling, as shown in Fig. 19. As $k_\epsilon$ gets larger, cycle-leveling is more strictly achieved. Even though we maintain a rather strict cycle-leveling, the cleaning efficiency does not degrade. Furthermore, we project that, for less strict cycle-leveling, the cleaning cost and frequency will be reduced further. Actually, this constant $k_\epsilon$ needs to be...
adjusted according to environment where the system is used. Moreover, if we could know the error range of limited cycles of flash memory, this could be reflected in the value of $k$.

Figures 20 and 21 represent the pattern of erase counts of all segments over time (write requests) for workload UNIX(90 → 10,80). In the case of Greedy II, some deep valleys are shown over write requests, but in the case of CICL III, a generally flat plane is formed. This indicates that CICL III distributes the erase cycles more evenly than Greedy II.

Let us consider the system’s lifetime with these figures. Suppose that the cycling limit is about 100, and a system failure occurs when the portion of flash space above this cycling limit exceeds 20% of the total space. Then at failure time, the running cycle capacity of Greedy II is about 5,700 (segment · erases), while that of CICL III is about 6,200 (segment · erases). Furthermore, for a given number of writes, the cleaning frequency is reduced further by a ratio of about 0.1. Then, we can expect that CICL III increases the system’s lifetime by about $20% \left( \frac{6,200}{5,700 \times (1-0.1)} - 1.0 \right)$ from that given by Greedy II.

5. Related Work

The methods found in previous works on flash memory management such as [1]–[3], [8], [9] have focused mainly on lowering cleaning cost. To enhance cleaning performance, a number of techniques in proposed in LFS (disk-based log-structured file system) have been applied to flash storage systems since cleaning in flash storage systems is analogous to garbage collection in LFS.

The greedy policy was used in [3], [9], and in [8] the cost-benefit policy was modified so as to dynamically adjust the threshold values of the number of free segments and segment utilization for triggering cleaner. Also, as a solution to separating cold data from non-cold data, LFS proposed a technique by which valid blocks in the buffer are grouped by age before being flushed into the disk [7], [10]. This age-sorting method was used as a part of the DAC algorithm proposed in [1] in order to migrate accessed data to their write frequencies. This data migration method is similar to locality gathering in eNVy system [2] in which valid data in cleaned segments are sorted out in approximate order of the degree of locality. Different from such data migration methods, [8] employs separate segments for cleaning the cold segments (whose data is rarely updated) under the modified cost-benefit policy so that cold data cannot be mixed with non-cold data. These data separation methods make cold data and non-cold data separate at all cleaning (or data update) times. This constant separation activity can disturb normal I/O operation and incur extra overhead. In contrast, under our flash memory management policy, the moderate amount of cold data are segregated from non-cold data only at an appropriate time after recognizing current data access pattern.

As for cycle-leveling, most previous studies have proposed naive and simple methods of a “flip-flop” nature. The cycle-leveling approach in [3] is similar to Greedy II, and [2] uses the swapping approach as in Greedy III. However, these methods consume a lot of system resources and time, which disturbs normal I/O execution. For example, swapping data between two segments requires providing buffer memory as large as a segment, erasing two segments, and rewriting swapped
data. Furthermore, the rewriting should wait until the erasing ends. In contrast, our method can avoid this problem by considering cycle-leveling at the cleaning time.

6. Conclusion

We have proposed a way of managing flash memory space for flash storage system that copes effectively with the features of flash memory while overcoming its constraints. We submit that our flash memory management objectives have been achieved; the proposed method overcomes several problems raised by flash memory’s one-way writing and limited cycling.

Simulation results show that the cleaning method involving collection operation performs especially well when storage utilization and the degree of locality are both high. This feature was also discussed with the analysis of collection operation. Moreover, we can ensure that the proposed cycle-leveling method works well together with our cleaner while avoiding internal conflicts between them. Previous cycle-leveling methods disturb cleaning operation and normal I/O, but our method works harmoniously with other components. This effect is possible since logging, collection operation, and segment allocation policy have the potential functionality of cycle-leveling. Consequently, our cycle-leveling method allows maximal storage space to be available at all times, so as to allow the construction of a highly reliable flash storage system.

The proposed method can be particularly attractive for flash storage systems in an office/engineering or OLTP environment, where small random writes with locality are common. Additionally, the core components of our method, namely the collection operation and cleaning index, do not depend on logging even though we use a logging strategy for flash space management; other strategies can be easily applied in conjunction with our method.

Acknowledgements

This work was supported by the Brain Korea 21 project. The RACT at Seoul National University also provided research facilities for this work.

References


Appendix A: Proof of Lemma 1

The cleaning frequency is obtained by dividing all write requests $W$ by the write requests for one cleaning. Within the selected segment, there are a total of $u_i\cdot\text{SegSize}$ in-use blocks, and after cleaning that segment, the resulting free space will be filled with $(1 – u_i)\cdot\text{SegSize}$ writes; that is, for every cleaning, $(1 – u_i)\cdot\text{SegSize}$ writes are requested. For simplicity, we assume the greedy cleaning policy (refer to Sect. 3.1). In that case, after performing sufficient cleaning operations, the utilization of cleaned segments becomes constant [12]. We also assume that $W$ equals the number of blocks of the remaining rewrite requests after the utilization value becomes constant. Thus, $f$ can be given by $f = \frac{W}{(1 – u_i)\cdot\text{SegSize}}$.

Appendix B: An Alternative of Cost Formula

Suppose $u_i$ for all $i$ equals $u_c$. Then substituting the equation of Lemma 1 into Eq. (1) yields $\frac{W(1 – u_c)\cdot\text{SegSize}}{(1 – u_i)\cdot\text{SegSize}}$. Similarly, the alternative to the cost formula, $\sum_{i=1}^{f} \frac{1 – u_i}{(1 – u_i)\cdot\text{SegSize}}$, becomes $\frac{W_{u_c}}{1 – u_c} \cdot \text{SegSize}$. We can easily find that these two formulas have almost the same mathematical behavior regardless of the value of $k_u$ by computing their derivatives.
Appendix C: Proof of Lemma 2

The greedy cleaning method allows cold data to be uniformly distributed over the log area. Thus, the expected size of the cold data to be collected from segment $i$ (whose utilization is $u_i$) is $\frac{m_i \cdot u_i \cdot l_f \cdot \text{SegSize}}{N_{seg}} = k_s \cdot u_i \cdot l_f \cdot \text{SegSize}$. Then, for segment $i$, whose cold data is collected, its size of valid data is $u_i (1 - k_s \cdot l_f) \text{SegSize}$. Therefore, the utilization of segment $i$ is $u_i (1 - k_s \cdot l_f)$.

□

Appendix D: Proof of Lemma 3

By the Definition 4, the collection period is calculated as a ratio of the size of write requests per collection ($X$) to the size of write request per cleaning ($Y$). First, for computing $X$, we consider the fact that as locality is higher, the rate of fragmentation of the cold data gets slower. We assume that only $(1 - l_f)$ of all accesses go to the cold data. Hence, the write requests per collection equals $\frac{\text{ColSize}}{(1-l_f)}$. Secondly, as for $Y$, we expect that a cleaned segment whose utilization is $u_{col}$ has valid data with a ratio $(1-u_{col})$. Thus, $Y$ equals $(1-u_{col}) \text{SegSize}$.

□

Appendix E: Proof of Theorem 1

From Lemma 3 and Eq. (5), the value of $c$ is calculated as $(1 - u_{col,i}) (1 - l_f)$. Thus, the collection gain is rewritten by $\sum_{i=1}^{p} \frac{(u_0 - (1 - u_{col,i}) (1 - l_f))}{1 - u_0} = \frac{u_{col,i}}{1 - u_{col,i}}$, which is simply denoted as $\sum_{i=1}^{p} g_{u_0,l_f}(u)$ with $u_{col,i} = u$. Here, if $g_{u_0,l_f}(u) > 0$, which is a sufficient condition for positive collection gain, then $\sum_{i=1}^{p} g_{u_0,l_f}(u) > 0$. Thus, let us differentiate $g_{u_0,l_f}(u)$ with respect to $u$, then $\frac{\partial g_{u_0,l_f}(u)}{\partial u} = \frac{1 - l_f}{1 - u_0} - \frac{1}{1 - u_0}$. By using this derivative, we find that $g_{u_0,l_f}$ has the maximum value at $u_m = 1 - \sqrt{\frac{1 - u_0}{1 - l_f}}$. Here, there exist two cases for $g_{u_0,l_f}(u) > 0$. (i) If $u_m < 0$, then $g_{u_0,l_f}(0) > 0$. (ii) If $u_m \geq 0$, then $g_{u_0,l_f}(u_m) > 0$. Consequently, as a sufficient condition for $\sum_{i=1}^{p} g_{u_0,l_f}(u) > 0$, we have $(l_f > u_0$ and $l_f > 1 - u_0)$ or $(l_f \leq u_0$ and $l_f > \frac{1}{4(u_0-1)} + 1)$.

□

Han-joon Kim received the B.S. and M.S. degrees in Computer Science and Statistics from Seoul National University, Seoul, Korea in 1996 and 1998 and the Ph.D. degree in Computer Science and Engineering from Seoul National University, Seoul, Korea in 2002, respectively. His current research interests include file systems, databases, information systems, and data/text mining technologies.

Sang-goo Lee received the B.S. degree in Computer Science from Seoul National University, Seoul, Korea in 1985 and the M.S. and Ph.D. degrees in Computer Science from Northwestern University, Evanston, IL, in 1987 and 1990, respectively. He taught at the University of Minnesota from 1989 to 1990, and worked as a research engineer at Electronic Data Systems, Troy, MI, from 1990 through 1992. He is currently an associate professor at the School of Computer Science and Engineering, Seoul National University, Korea. His current research interests include databases, e-business solutions, information retrieval, and digital libraries.