Marina Gavrilova et al. (Eds.)

Computational Science and Its Applications – ICCSA 2006

LNCS 3980

International Conference
Glasgow, UK, May 2006
Proceedings, Part I

1 Part I

Springer

An Intelligent Garbage Collection Algorithm for Flash Memory Storages

Long-zhe Han[1], Yeonseung Ryu[1,⋆], Tae-sun Chung[2] , Myungho Lee[1], and Sukwon Hong[1]

[1] Department of Computer Software, Myongji University, Korea
[2] College of Information Technology, Ajou University, Korea
ysryu@mju.ac.kr, longzhehan@gmail.com

Abstract. Flash memory cannot be overwritten unless erased in advance. In order to avoid having to erase during every update, non-in-place-update schemes have been used. Since updates are not performed in place, obsolete data are later reclaimed by garbage collection. In this paper, we study a new garbage collection algorithm to reduce the cleaning cost such as the number of erase operations and the number of data copies. The proposed scheme automatically predicts the future I/O workload and intelligently selects the victims according to the predicted I/O workload. Experimental results show that the proposed scheme performs well especially when the degree of locality is high.

1 Introduction

Flash memory is becoming important as nonvolatile storages because of its superiority in fast access speeds, low power consumption, shock resistance, high reliability, small size, and light weight [7, 11, 8, 6, 13]. Because of these attractive features, and the decreasing of price and the increasing of capacity, flash memory will be widely used in consumer electronics, embedded systems, and mobile computers.

Though flash memory has many advantages, its special hardware characteristics impose design challenges on storage systems. First, flash memory is organized in terms of *blocks*, where each block is of a fixed number of *pages* [8]. A block is the smallest unit of erase operation, while reads and writes are handled by pages. The size of page is fixed from 512B to 2KB and the size of block is somewhere between 4KB and 128KB depending on the product. Second, flash memory cannot be written over existing data unless erased in advance. Besides erase operation can be performed in a larger unit than the write operation and it takes an order of magnitude longer than a write operation (See Table 1). Third, the number of times an erasure unit can be erased is limited (e.g., 10,000 to 1,000,000 times). Therefore, data must be written evenly to all blocks to avoid wearing out specific blocks to affect the usefulness of the entire flash memory device, that is usually named as *wear leveling*.

⋆ This work was supported by the Korea Research Foundation Grant funded by the Korean Government(MOEHRD)(R08-2004-000-10391-0).

M. Gavrilova et al. (Eds.): ICCSA 2006, LNCS 3980, pp. 1019–1027, 2006.

Table 1. Characteristics of different storage media. (NOR Flash:Intel 28F128J3A-150, NAND Flash: Samsung K9F5608U0M).

Media	Access time		
	Read (512B)	Write (512B)	Erase
DRAM	2.56μs	2.56μs	
NOR Flash	14.4μs	3.53ms	1.2s (128KB)
NAND Flash	135.9μs	226μs	2-3ms (16KB)
Disk	12.4ms	12.4ms	

Since blocks should be erased in advance before updating, updates in place is not efficient. All data in the block to be updated must first be copied to the system buffer and then updated. After the block is erased, all data must be written back from the system buffer to the block. This results in poor update performance. Moreover, the blocks of hot spots would soon be worn out. To solve these problems, data are updated to empty spaces and obsolete data are left at the same place as garbage, which a garbage collector later reclaims.

There have been many researches on garbage collection algorithms (or cleaning policies) for log-structured file systems used on disk-based storages. Recently, some garbage collection algorithms for flash memory file systems are proposed. Garbage collection algorithms should deal with some issues such as how many blocks to erase, which blocks to erase, and where to migrate valid data from erased block. The primary concern of garbage collection algorithms has been to reduce the cleaning cost. But, the number of victim blocks is also a problem for garbage collection policy of flash memory file system. This is because the cost of erase operation is much higher than read/write operations and thus garbage collection could disturb normal I/O operations.

In this paper, we study an intelligent garbage collection algorithm, which predicts I/O workload of the near future and determines the number of victim blocks to be erased according to the predicted I/O workload. If we can predict the I/O workload such as the number of I/O request arrivals during the next garbage collection execution, we can control the number of victim blocks to be erased according to the estimated I/O workload. When the number I/O request arrivals during the next garbage collection is estimated as high, garbage collector selects one or no victim block. Otherwise, garbage collector can select several victim blocks. When garbage collector gathers many valid data from several victim blocks, it can perform data migration efficiently by grouping the data according to their characteristics (i.e., cold data and non-cold data).

The rest of this paper is organized as follows. In Section 2, we review previous works that are relevant for this paper. In Section 3, we present the architecture of proposed garbage collection and deal with the problem of predicting the I/O workload. We also propose a new garbage collection algorithm. Section 4 presents the experimental results to show the performance of proposed scheme. The conclusions of this paper are given in Section 5.

2 Background

2.1 The Cost of Garbage Collection

Flash memory cannot be written over existing data unless erased in advance. Besides erase operation can be performed in a larger unit than the write operation and it takes an order of magnitude longer than a write operation. The erase operation can only be performed on a full block and is slow that usually decreases system performance and consumes power. Therefore if every update is performed in place, then performance is poor since updating even one byte requires one erase and several write operations. In order to avoid having to erase during every update, a *logging approach* has been recommended since it is quite effective in several ways (see Fig. 1). First, logging 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 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.

When data are updated to empty spaces at the end of the log, obsolete data are left at the same place as *garbage*, which a *garbage collector* process later reclaims. Since garbage collection can be performed in the backgound, update operation can be performed efficiently. The operation of garbage collection usually involves three stages as Fig. 2. It first selects victim blocks and then identifies valid data that are not obsolete in the victim blocks. And it copies valid data from the victim blocks to the end of log. After that, the victim blocks are erased and available for rewriting.

The cleaning cost and the degree of wear-leveling are two primary concerns of garbage collector. The garbage collector tries to minimize cleaning cost and wear down all blocks as evenly as possible. Sometimes the objective of minimizing cleaning cost conflicts with that of wear-leveling. For example, excessive wear-leveling generates a large number of invalidated blocks, which degrades cleaning performance.

First, we define the cleaning cost. Let MC_i be migration cost and EC_i be erasure cost for i-th garbage collection, respectively. The migration cost denotes the number of copies of valid data from the victim blocks to free space in other

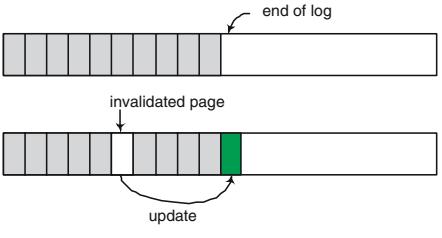


Fig. 1. Log-structured management

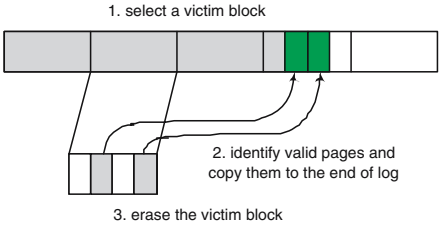


Fig. 2. Three steps of garbage collection

blocks. The erasure cost denotes the number of erasure. Then the cleaning cost of garbage collection can be described as follows:

$$\sum MC_i + EC_i \tag{1}$$

The cost to erase a block is much higher than to write a whole block. The erasure cost dominates the migration cost in terms of operation time and power consumption. Therefore, the number of erase operations determines the garbage collection costs. For better performance and power conservation, the primary goal is to minimize the number of erase operations.

Next, we define the degree of wear-leveling as follows:

$$\epsilon = E_{max} - E_{min} \tag{2}$$

where E_{max} is the maximum erase count and E_{min} is the minimum erase count, respectively. The smaller ϵ is, the longer the lifetime of system is. Since excessive wear-leveling does cleaning performance more bad than good, it is sufficient that ϵ should be below a predefined threshold.

2.2 Garbage Collection Algorithms

There are some issues of garbage collection algorithms:

When. When is garbage collection started and stopped? It usually executes periodically or is triggered when the number of free blocks gets below some threshold.

How many. How many blocks are cleaned at once? The more blocks are cleaned at once, the more valid data can be reorganized. However, cleaning several blocks needs much time, which can disturb normal I/O execution. Thus, most garbage collection algorithms select only one block.

Which. Which block is selected for erasing? One may select a block with the largest amount of garbage or select blocks using information such as age, update time, etc. This is referred to as *victim selection algorithm*.

Where. Where is the valid data written out? This is referred to as *data migration algorithm*. There are various ways to reorganize valid data, such as enhancing the cleaning performance by grouping pages of similar age together or grouping related files together into the same block, etc.

A number of victim selection algorithms based on the *block utilization* have been studied [9, 15, 10, 4, 12]. The *greedy* algorithm selects blocks with the largest amount of garbage for erasure, hoping to reclain as much as possible with the least cleaning work [12]. The greedy policy tends to select a block in a FIFO order irrespective of data access patterns. It is known that the greedy policy works well for uniform access, but does not perform well for high localities of access [15]. The *cost-benefit* algorithm chooses blocks that maximize the formula [12, 9]: $\frac{age \cdot (1-u)}{u+1}$, where u is the utilization of a block (the fraction of space occupied by valid data) and *age* is the time since the most recent modification,

respectively. The cost is derived from migration and erasure, which are reflected by the denominator $(u+1)$. The benefit is given as the space-time product form. The term $(1-u)$ reflects how much free space it acquires. Because of term *age*, cold blocks can be cleaned at a much higher utilization than hot blocks.

There are several methods to migrate valid data to the cleaned blocks. Most schemes try to gather hot data together to form the largest amount of garbage to reduce garbage collection cost. The *age-sort* algorithm used in Log-Structued File system(LSF) sorts valid data by age before writing them out to enforce the gathering of hot data [12]. For better effect, several blocks are cleaned at once. The *separate block cleaning* algorithm uses separate blocks in cleaning: one for cleaning not-cold blocks and writing new data, the other for cleaning cold segments [9]. The separate segement cleaning was show to perform better than when only one segment is used in cleaning, since hot data are less likely to mix with cold data. The *dynamic data clustering* algorithm clusters data according to their update frequencies by active data migration [5].

As for wear-leveling, simple swapping approaches have also been proposed. However, swapping data between two blocks requires buffer memory, erasing two blocks and rewriting swapped data. Thus the swapping methods consume a lot of available system resources and time, which could disturb normal I/O execution.

3 Intelligent Garbage Collection

3.1 Motivation

The performance of garbage collection depends on the combination of victim selection policy and data migration policy. The cost-benefit policy, a representative victim selection algorithm, generally performs better than the greedy policy. But it does not perform well for high localities of access without combining efficient data migration policy. We will show it in next section. Under high localities of access, after a large number of logging and cleaning operations, cold data becomes mixed with non-cold data within each block. After that time, cold data moves around uselessly together with non-cold data. For this reason, the utilization of cleaned blocks remains stable at a high value and the amount of free space collected becomes small. In other words, migration cost and erasure cost could be increased. In order to overcome this problem, the cost-benefit policy has to combine data migration policy that separates cold data and hot data when migrating valid data.

Many flash memory file systems are based on Log-Structued File system [14, 1]. But, the separate block cleaning policy and the dynamic data clustering policy cannot be used for log-structured file system. The age-sort policy was used in Log-Structued File system. It sorts the valid pages of victim blocks by the time they were last modified and migrates them at the end of log. For example, it migrates the oldest pages first at the end of log. We use the *cost-benefit with age-sort* algorithm like Log-Structued File system [12].

The problem of this policy is that garbage collector should select several victim blocks for better separation of cold and non-cold data. When it collects valid data

from many victim blocks, there may be high probability of isolating cold data and non-cold data and of migrating them to separated blocks. But, previous garbage collection algorithms choose one victim block since erase operation takes much time and thus erasure of several victims at once can disturb normal I/O operation.

If we can predict the I/O workload such as the number of I/O request arrivals during the next garbage collection execution, we can control the number of victim blocks to be erased according to the estimated I/O workload. When it is predicted that the number I/O request arrivals for the next garbage collection is high, garbage collector selects one or no victim block. Otherwise, garbage collector can select several victim blocks and thus improve its performance.

3.2 Architecture

The proposed garbage collection module consists of three components: (i) a *monitor* that measures the request arrival rate, (ii) a *predictor* that uses the measurements from the monitor module to estimate the workload characteristics in the near future, and (iii) a *garbage collector* that performs garbage collection task.

The monitor is responsible for measuring the request arrivel rate. The monitor tracks the number of request arrivals a_i in each measurement interval (I) and records this value. The monitor maintains a finite history consisting of the most recent H values of the number of arrivals. Let A_i be the sequence $a_i^1, ..., a_i^H$ of values from the measurement history. Let W be the time units to execute garbage collector. The predictor uses the past measurements to predict the number of arrivals $\hat{n}_i$ and the arrival rate λ_i for the W time units.

The garbage collector uses the cost-benefit with age-sort algorithm. It uses the predicted workload to determine the number of victim blocks.

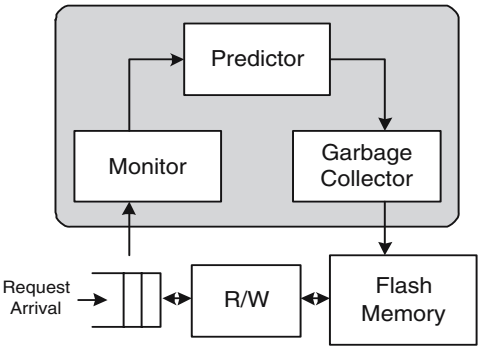


Fig. 3. Proposed garbage collector architecture

3.3 Predicting the Arrival Rate

This section presents a method to predict the I/O arrival rate. In order to predict the number of arrivals $\hat{n}_i$, we use the model of AR(1) process [3, 2](autogressive of order 1). Using the AR(1) model, a sample value of A_i is estimated as

$$\hat{a}_i^{j+1} = \bar{a}_i + \rho_i(1) \cdot (a_i^j - \bar{a}_i) + e_i^j, \tag{3}$$

where, ρ_i and $\bar{a}_i$ are the autocorrelation function and mean of A_i respectively, and e_i^j is a white noise component. We assume e_i^j to be 0, and a_i^j to be estimated values $\hat{a}_i^j$ for $j \geq H+1$. The autocorrelation function ρ_i is defined as

$$\rho_i(l) = \frac{E[(a_i^j - \bar{(a)}_i) \cdot (a_i^{j+l} - \bar{a}_i)]}{\sigma_{a_i}^2}, 0 \leq l \leq H-1, \tag{4}$$

where, σ_{a_i} is the standard deviation of A_i and l is the lag between sample values for which the autocorrelation is computed.

Let $M = W/I$. Then we estimate $\hat{a}_i^{H+1}, ..., \hat{a}_i^{H+M}$ using Equ. 3. Then, the estimated number of arrivals in W time units is given be $\hat{n}_i = \sum_{j=H+1}^{H+M} \hat{a}_i^j$ and finally, the estimated arrival rate, $\hat{\lambda}_i = \frac{\hat{n}_i}{W}$.

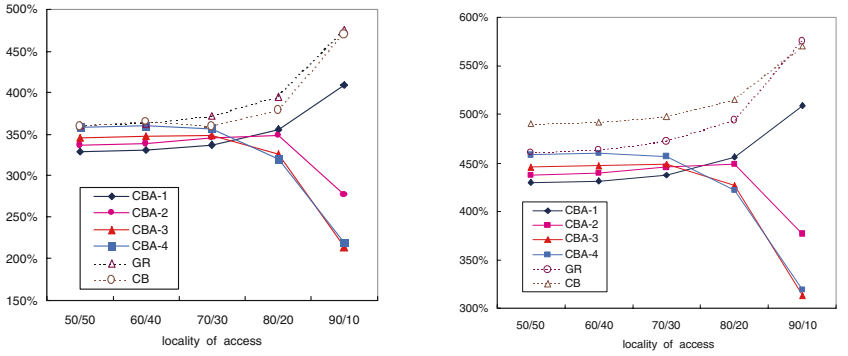
4 Experiment

We have performed simulations in order to investigate the cost-benefit with age-sort policy by varying the number of victim blocks. There have been no previous works about it. We have also implemented three algorithms for comparison: GR represents the greedy policy with no separation of hot and cold data; CB represents the cost-benefit policy with no separation of hot and cold data; and CBA-x represents the cost-benefit with age-sort policy, where x is the number of victim blocks.

Since at low utilization garbage collection overhead does not significantly affect performance, in order to evaluate the effectiveness, we initialized the flash memory by writing data sequentially to fill it to 90% of flash memory spaces. The created workloads then updated the initial data according to the required access patterns. We used the notation for locality of reference as 'x/y' that x% of all accesses go to y% of the data while $(1-x)$% go to the remaining $(1-y)$% of data.

We define the number of extra erase operations as the number of erase operations minus the number of erase operations from an ideal scheme. The ideal scheme is defined as a scheme that performs one erase operation for every n-page write requests, where n is the number of pages per block. Similarly, the number of extra write operations is defined as the number of write operations minus the number of writes requested. Performance metrics are the ratio of the number of extra erase operations to the number of erase operations from ideal scheme, the ratio of the number of extra write operations to the number of write requests and the degree of wear-leveling.

Fig. 4 (a) shows the ratio of the number of extra erase operations to the number of erase operations from ideal scheme and (b) shows the ratio of the number of extra write operations to the number of write requests. Both figures illustrate that CBA performed best. As the locality is increased, the performance of CBA-2, CBA-3, and CBA-4 increased rapidly whereas GR, CB, and CBA-1



(a) Erase performance (b) Write performance

Fig. 4. Comparison of erase and write performance

Table 2. Comparison of wear-leveling degree

	CBA-1	CBA-2	CBA-3	CBA-4	GR	CB
50/50	4	6	11	21	4323	5
60/40	5	11	15	239	4651	6
70/30	6	841	747	11083	6054	6
80/20	7	2387	4215	39513	9515	12
90/10	26	10619	28177	80165	18569	40

deteriorated severely. This is becasue CBA with several victims can separate data, such that cold data are less likely to mix with hot data as compared with the other policies. This effect is more prominent under higher locality of access. Fig. 4 also shows that the performance of CBA-3 and CBA-4 are nearly the same. Thus, it is sufficient to select at most three victim blocks.

Table 2 shows the simulation results about the degree of wear-leveling. It illustrates that CB and CBA-1 provide stable wear-leveling effects, while the leveling effect of GR and CBA-{2|3|4} is not satisfactory especially when the locality is high. In case of CBA with several blocks, hot data and cold data are stored separately on different blocks and hot blocks are much likely to erase frequently. Thus, the wear-leveling degree becomes higher.

5 Concluding Remarks

In this paper, we propose an intelligent garbage collection algorithm, which predicts I/O workload of the near future and determines the number of victim blocks according to the predicted I/O workload. If we can predict the number of I/O request arrivals during the next garbage collection execution, we use this information to control the number of victim blocks so that garbage collector can gather valid data from several victim blocks as much as possible. Proposed

garbage collection scheme can reduce the cleaning cost by performing data migration efficiently. Experimental results show that the proposed scheme performs well especially when the degree of locality is high.

References

1. Yaffs (yet another flash filing system). http://www.aleph1.co.uk/yaffs/.
2. G. Box and G. Jenkins. *Time Series Analysis: Forecasting and Control*. Holden-Day, 1976.
3. A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements, 2002.
4. M. Chiang and R. Chang. Cleaning policies in mobile computers using flash memory. *Journal of Systems and Software*, 48(3):213–231, 1999.
5. M. Chiang, P. Lee, and R. Chang. Using data clustering to improve cleaning performance for flash memory. *Software Practice and Experience*, 29(3):267–290, 1999.
6. T. Chung, D. Park, Y. Ryu, and S. Hong. Lstaff: System software for large block flash memory. *Lecture Notes in Computer Science*, 3398:704–710, 2005.
7. F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. Tauber. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, 1994.
8. Samsung Electronics. 256m x 8bit / 128m x 16bit nand flash memory. http://www.samsungelectronics.com.
9. A. Kawaguchi, S. Nishioka, and H. Motoda. Flash memory based file system. In *Proceedings of USENIX95*, pages 155–164, 1995.
10. H. Kim and S. Lee. An effective flash memory manager for reliable flash memory space management. *IEICE Trans. Information and Systems*, E85-D(6):950–964, 2002.
11. B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii International Conference on Systems Sciences*, 1994.
12. M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Computer Systems*, 10(1):26–52, 1992.
13. Y. Ryu and K. Lee. Improvement of space utilization in nand flash memory storages. *Lecture Notes in Computer Science*, 3820:766–775, 2005.
14. David Woodhouse. Jffs: The journalling flash file system. In *Proceedings of the Ottawa Linux Symposium*, 2001.
15. M. Wu and W. Zwanepoel. envy: A non-volatile, main memory storage system. In *Proceedings of the 6th Internation Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.