1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

DATA MANAGEMENT IN A FLASH MEMORY BASED STORAGE SERVER*

Mei-Ling Chiang[+], *Paul C. H. Lee*[£], *and Ruei-Chuan Chang*[+£]

Department of Computer and Information Science[+]
National Chiao-Tung University, Hsinchu, Taiwan, R.O.C.
Email: joanna@os.nctu.edu.tw, rc@cc.nctu.edu.tw

Institute of Information Science[£]
Academia Sinica, Nankang, Taiwan, R.O.C.
Email: paul@iis.sinica.edu.tw

ABSTRACT

Flash memory has many attractive features, such as non-volatility, light weight, and low power consumption. These features show promise for using flash memory as storage in consumer electronics, embedded systems, and mobile computers. However, flash memory has specific hardware characteristics that impose challenges on the design of storage systems. It cannot be overwritten unless erased in advance. The erase operations are slow and power-wasted, which usually decrease system performance and consume lots of power. In addition, the number of times that flash memory can be erased is also limited. This paper descries the design and implementation of a storage server for flash memory. To overcome hardware limitations, the server employs an effective dynamic data clustering method and an efficient cleaning algorithm. Performance evaluation shows that, with these mechanisms, throughput is significantly improved, flash memory lifetime is prolonged, and even wearing is ensured.

1. INTRODUCTION

Flash memory is non-volatile, which can retain data even after system is powered off. Besides, it has many attractive features, such as fast access speed, low power consumption, shock resistance, small size, and light weight [2,3,15]. As its price decreases and capacity increases, flash memory is expected to be largely used in consumer electronics, embedded systems, and mobile computers. Applications are digital cameras, voice recorders, set-top boxes, pagers, cellular phones, notebooks, hand-held computing devices, Personal Digital Assistants (PDAs) [9], etc.

Flash memory has special hardware characteristics [7,12,13,18] that impose challenges on the design of storage systems. Flash memory is partitioned into *segments*[1] and the segment sizes are defined by hardware manufactures. Segments cannot be written over existing data unless erased in advance. The erase operations must be performed only on whole segments, which waste relatively lots of power. Besides, the erase operations are slow compared to read operation and write operation. Write operations are much slower than read operations. Furthermore, the number of times a segment can be erased is limited. Table 1 lists the typical flash memory characteristics [12,13].

Read Cycle Time	150 ~ 250 *ns*
Write Cycle Time	6 ~ 9 *us*/byte
Block Erase Time	0.6 ~ 0.8 sec
Erase Block Size	64 Kbytes or 128 Kbytes
Erase Cycles Per Block	100,000 ~ 1,000,000

Table 1: Flash memory characteristics.

Because of these hardware limitations, the design of flash memory based storage systems should avoid having to erase as possible for the longer flash memory lifetime, better system performance, and power conservation. To maximize the lifetime of flash memory and avoid wearing out some segments to affect the usefulness of entire flash memory, write and erase operations must be balanced over the whole flash memory. The operation is called *wear-leveling* or *even wearing* [7].

Since flash segment sizes are large, storage systems generally divide flash segments into smaller read/write blocks. Updating data in place is not efficient since all data in the segment to be updated must first be copied out and then updated. After the segment has been erased, all data with updates must be written back to the segment. Thus, updating even one byte data requires one slow erase and several write operations. If every update is performed in place, then performance will be poor and flash memory blocks of hot spots will soon be worn out.

Our goal is to design and implement a flash memory storage server to overcome limitations from hardware characteristics. To avoid having to erase in every update,

* This work was partially supported by the National Science Council of the Republic of China under grant no. NSC88-2213-E-001-016.

[1] We use "segment" to represent hardware-defined erase block, and "block" to represent software-defined block.

we use the *non-update-in-place* scheme for data updating. Instead of updating data at the same address, data updates are written to any empty space in flash memory and obsolete data are left as garbage. A software cleaner later reclaims these garbage by migrating valid data from the segment to be cleaned to another segment. Then the original segment is erased and available for rewriting.

With this *non-in-place-update* mechanism, the cleaner has significant effect as the utilization (the percentage of flash memory space occupied by valid data) gets higher because more segments need to be reclaimed in order to have one free segment. As a result, more data must be migrated and more erasures have to be done. Performance is thus severely degraded, lifetime is greatly decreased, and energy consumption is greatly increased. Cleaning policies determining which segments to clean, when to clean them, and how to clean them control the behavior of the cleaner. Thus, cleaning policies severely affect cleaning performance [5,14,21] and are key to flash memory management.

Two major concerns regarding cleaning policies are the *segment selection algorithm* that determines the segments to be cleaned and the *data reorganization method* that determines how to migrate valid data in the selected segments. The data reorganization method has the most important impact on cleaning performance [5]. Clustering hot data together in the same segments can reduce cleaning overhead has been shown in previous literature [5,14].

In this paper, the data reorganization method, Dynamic Data Clustering (DAC), is used for data clustering. The DAC method dynamically clusters data according to data update frequencies by active data migration. The data clustering is performed during segment cleaning and data updating time, which is fine-grained and low-overhead. When selecting segments to clean, the Cost Age Times policy (CAT) [4,5] is used, which selects segments according to cleaning cost, ages of data in segments, and the number of times the segment has been erased. Because the number of times segments has been erased is concerned, even wearing is ensured.

This paper describes the design and implementation of a storage server utilizing DAC method and CAT policy. Performance evaluation shows that the number of erase operations performed and the cleaning overhead are significantly reduced. Flash memory is evenly worn.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 presents the flash memory management scheme using DAC method and CAT policy. Section 4 describes the design and implementation of the flash memory server. Section 5 shows performance evaluation results, and Section 6 concludes this paper.

2. RELATE WORK

Several storage systems and file systems have been developed for flash memory. They are either designed from the scratch for flash memory or use the device driver approach. The driver approach is used with existent file systems and translates the file system requests from disk sectors to flash memory addresses.

Wu and Zwaenepoel [21] proposed a large flash memory based storage system, eNVy, to provide flash memory as linear memory array rather than emulated disks. eNVy uses hardware support of copy-on-write and page-remapping techniques to avoid updating data in place. Its *hybrid cleaning* method combining FIFO and locality-gathering in cleaning segments aims to minimize the cleaning costs for uniform access and high localities of reference.

Microsoft's Flash File System (MFFS) [20] uses a linked-list data structure to manage data and supports the DOS FAT system. MFFS maintains data blocks of variable size instead of fixed length. The *greedy policy* that always selects segments with the largest amount of garbage for cleaning is used to clean segments.

M-Systems's TrueFFS [6] allows flash memory to emulate hard disks for DOS and Windows. TrueFFS is a software block device driver to be used with an existent file system. The data recording format, patented by M-Systems, is called Flash Translation Layer (FTL) standard. Flash memory is divided into fixed-sized read/write blocks. TrueFSS uses a statistical approach to wear-leveling. The garbage collection selects segments with the large amount of garbage, the least number of erasures, and the most static data. It then decides which segments to clean. A random selection process is also used to ensure the evenly reclamation among all segments.

Rosenblum et al. [16] suggested that the Log-Structured File System [16,17,19] that writes data as append-only log instead in-place update can be applied to flash memory. Kawaguchi et al. [14] used a log approach similar to LFS to design a flash-memory-based file system for UNIX. The device driver approach is used. They also modified the cost-benefit policy [16,17] of LFS to use different cost measure for flash memory. The proposed *separate segment cleaning*, which separates hot segments from cold segments, was used when segments are to be cleaned. Wear leveling is not implemented.

David Hinds [10,11] implemented flash memory drivers in the Linux PCMCIA [1] package. It uses the greedy method at most of time, but sometimes chooses to clean the segment that has been erased the fewest number of times for even wearing.

In our early study, the *CAT* policy [4,5] is proposed to take into account utilization, segment age, and the number of times segments have been erased in selecting segments to clean. Because the number of erasures performed on individual segments is concerned, flash memory is more evenly worn than greedy policy and cost-benefit policy. Valid blocks in the segments to be cleaned are migrated into separate segments depending on whether the blocks

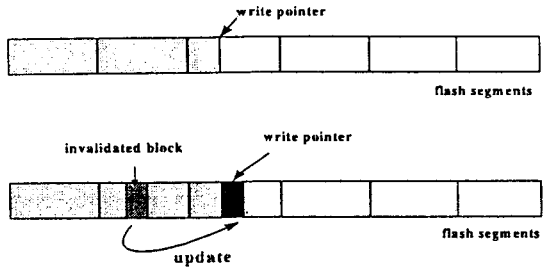


Figure 1: Non-in-place update that updates data to empty flash memory spaces.

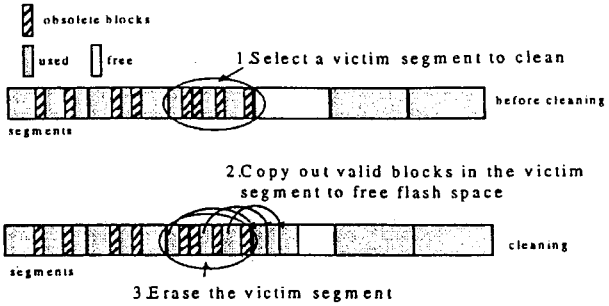


Figure 2: Three-stage operations of cleaning process.

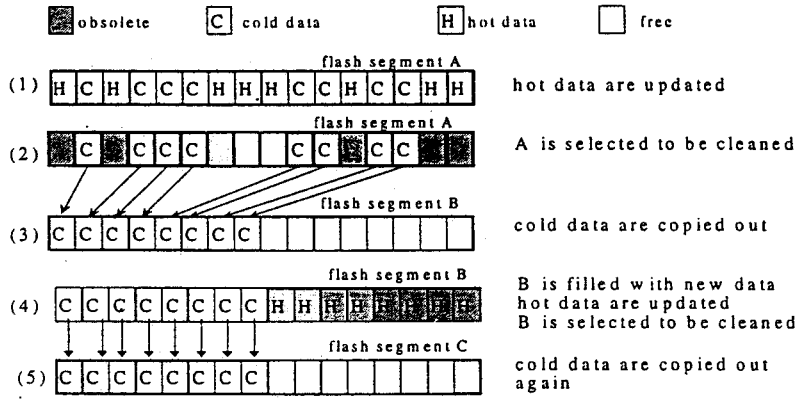


Figure 3: Cold data are migrated again and again when they are mixed with hot data.

are hot or cold. Data reorganization was shown to be the most important factor affecting cleaning performance [5].

Douglis et al. [8] provided a detailed discussion of storage alternatives for mobile computers: hard disks, flash memory disk emulators, and flash memory cards. It is concluded that a flash memory file system has the most attractive qualities with respect to energy and performance. They showed that the key to flash memory file system is erasure management and found that the utilization of a flash memory card has substantial impact on energy consumption, performance, and endurance.

3. FLASH MEMORY MANAGEMENT

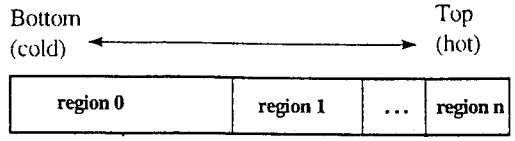
We use the *non-in-place-update* scheme in our server to manage data in flash memory to avoid having to erase during every update. As shown in Figure 1, data updates are written to any empty space, and the obsolete data are marked invalidated and left at the original place as garbage. When the number of free segments is below a certain threshold, a software cleaning process, *cleaner*, begins to reclaim garbage.

The cleaning process involves three-stage operations, as shown in Figure 2. The cleaner first selects a victim segment for cleaning and identifies *valid* data (not obsolete) in this victim segment. Then it migrates valid data out by copying them to free space in other segments. Finally, the victim segment is erased and available for new data.

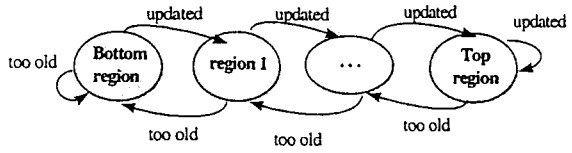
3.1 Data Reorganization Method

The way valid data in victim segments are migrated during segment cleaning can severely affect future cleaning costs [5,14]. The simplest way is to copy valid data to another segment in the same order as they appear in the victim segment. However, hot data and cold data are possible to be mixed. Figure 3 illustrates the following situation. If the victim segment contains both cold data and hot data, cold data have high possibility to remain valid at the cleaning time since cold data are updated less frequently. Cold data thus are migrated during cleaning process. The new segment is possible filled with hot data and soon is selected for cleaning. The cold data previously migrated are possible to remain valid and are migrated again and again.

If data are migrated in the way that *hot data* (most frequently updated data) are clustered in the same segments, then flash segments will be either full of all hot data or all non-hot data. Then segments containing most of the hot data will soon contain the largest amount of



(a) Partitioning flash memory into regions.



(b) State transition diagram.

Figure 4: State machine for data clustering.

invalidated blocks because hot data have high possibility to be updated soon and become invalidated. Cleaning these hot segments can reclaim the largest amount of garbage and the least amount of valid data must be migrated during cleaning. Cleaning costs thus can be significantly reduced.

Since separately clustering hot data and cold data can reduce cleaning overhead, the major problem of cleaning becomes how to effectively cluster hot data. Previous researches [4,5,14,21] reorganize data only at the cleaning time when valid data in the segment to be cleaned are migrated. We noted that data reorganization can not only be done during cleaning, but also can be done during data updating time without extra cost. By taking advantage of the chance that when data are updated, they are updated to another free flash space, hot data and cold data can be separately clustered.

We propose a new data reorganization method that uses a state machine to dynamically cluster data according to their access frequencies during run time and cleaning time: **DAC** (**D**ynamic d**A**ta **C**lustering) approach. Instead of classifying data into hot and cold as in [4,5,14], the DAC approach clusters data according to their write access frequencies. Only write access frequencies are concerned is because only write operations incur cleaning.

The DAC method logically partitions flash memory into several *regions* that contain data with different localities of reference, as shown in Figure 4(a). Data blocks in the same region have similar write access frequencies. Each data block is associated with a state indicating which region the block resides in. As their access frequencies change over time, data blocks are actively migrated between neighboring regions according to the state transition criteria. So regions can be dynamically shrunk or enlarged.

The state machine shown in Figure 4(b) contains several states and the starting state is "*Bottom region*". The state machine operates as follows. When a data block is newly created, it is allocated in the *Bottom* region. Once a data block is updated and is young to the current region (i.e., the

```
Write()
{
    If new write {
        Allocate a free block in Bottom region;
        Write data into the free block;
    } else
        Update ();
}

Update()
{
    Mark the obsolete data as invalid;
    If the data block is young to the current region
        Allocate a free block in the upper one region;
    else
        Allocate a free block in the current region;
    Write data into the free block;
}

Cleaning()
{
    Select a victim segment for cleaning;
    For all valid data blocks in the victim segment
    {
        Mark the block as invalid;
        If the data block is old to the current region
            Allocate a free block in the lower one region;
        else
            Allocate a free block in the current region;
        Copy this valid data block into the free block;
    }
    Erase the victim segment;
    Enqueue the victim segment to free segment list;
}
```

Figure 5: Operations for update and cleaning process.

resident time in the current region is smaller than a certain threshold), then it is promoted to the upper one region and its state changes accordingly. That is, the obsolete data block residing in the original region is invalidated as garbage and the update data are written to any free space in the segments belonging to the upper one region. Otherwise, the update data are written to the free space in the segments belonging to the current region.

When garbage collection is needed, the cleaner reclaims the invalidated spaces which obsolete data occupy. The segment to be cleaned is selected based on a certain segment selection algorithm. If the valid blocks in the cleaned segment are old to the current region (i.e., their resident times exceed a certain threshold), then they are demoted to the lower one region and their states change accordingly. That is, these valid blocks are migrated back to the lower one region by coping data into free space in the segments belonging to the lower one region. Otherwise, the blocks are migrated to the free space in the segments belonging to the current region. After all valid blocks in the segment to be cleaned are migrated out, the segment is erased and available for new data.

By this active data migration between regions during data updating and segment cleaning, data blocks in the *Top* region are the hottest during the recent accesses. The closer to the Top region, the hotter the block is. Otherwise, the

colder the block is. Therefore, data blocks of similar access frequencies can be effectively clustered. Figure 5 shows the detailed operations.

In comparison, DAC approach dynamically clusters data according to their write access frequencies during cleaning time and during data updating, while other policies [4,5,14] cluster data at the cleaning time by using certain criteria to determine whether data blocks are hot or not. The criteria may need more effort to be performed. Besides, only two states of data are allowed (i.e., hot and cold). The DAC approach is more fine-grained and more effective in data clustering since more states of data are allowed depending on the configuration of state machine.

3.2 Segment Selection Algorithms

The *Cost Age Times* (CAT) formula [4,5] is used. The cleaner chooses to clean segments that minimize the formula:

$$Cleaning\ Cost_{Flash\ Memory} * \frac{1}{Age} * Number\ of\ Cleaning.$$

The *cleaning cost* is defined as the cleaning cost of every useful write to flash memory as $u/(1-u)$, where u (utilization) is the percentage of valid data in a segment. Every $(1-u)$ write incurs the cleaning cost of writing out u valid data. The *age* is defined as the elapsed time since the segment was created. The *number of cleaning* is defined as the number of times a segment has been erased. The basic idea of CAT formula is to minimize cleaning costs, but gives segments just cleaned more time to accumulate garbage for reclamation. In addition, to avoid concentrating cleaning activities on a few segments, the segments erased the fewest number of times are given more chances to be selected for cleaning. Besides, to avoid wearing specific segments out and thus limiting the usefulness of whole flash memory, we swap the segment erased most times and the segment erased fewest times when a segment is reaching its projected lifecycle limit.

4. SYSTEM DESIGN AND IMPLEMENTATION

The server manages flash memory as fixed-size blocks and uses the *non-in-place-update* scheme. Every data block is associated with a unique constant logical block number, while its physical location in flash memory changes when updated. The server uses table-mapping method to map logical lock numbers to physical locations in flash memory.

We first describe the data layout on flash memory in Section 4.1. We then describe three tables, the region table, the translation table, and the lookup table, in Section 4.2, 4.3, and 4.4, respectively. Those tables are constructed in main memory and maintained during runtime, which are used mainly to speed up processing. The information stored in tables is only copies of information stored in flash memory. Therefore, even if power failures occur, these

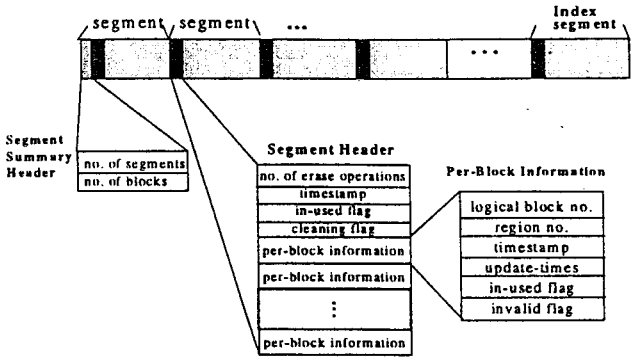


Figure 6: Data layout on flash memory.

tables can be reconstructed from flash memory. Storing these tables requires a substantial amount of main memory: 12 bytes per region, 13 bytes per block, and 17 bytes per segment. However, it is a trade-off between space consumption and performance. Because currently flash memory capacity is small, the space overhead is limited. For a 24-Mbyte flash memory, 128-Kbyte segments, 4K-byte blocks, and 4 regions, these tables take up 78 Kbytes of main memory.

4.1 Data Layout on Flash Memory

The data layout on flash memory is shown in Figure 6. Each segment has a *segment header* to record segment information, such as the number of times the segment has been erased, a timestamp, control flags for cleaning, and the *per-block information array*. The per-block information array contains information about each block in the segment, such as the corresponding logical block number, timestamp, the number of times the block has been updated, and invalid flag. The invalid flag indicates whether a block is obsolete or not. The *index segment* keeps track of currently active segments for data writing in each region. The *segment summary header*, located in the first segment, records global information about flash memory, such as the total number of segments in flash memory and the total number of blocks per segment.

4.2 Region Management

A *region table*, shown in Figure 7, keeps track of information for each region, including the total number of segments in the region, the currently active segment for data writing, and a *region segment list*. The region segment list keeps track of each segment in the region. This table is used by the cleaner in selecting segments to clean.

The *active segments* record those segments currently used for data writing in each region. They are stored in the index segment on flash memory. A *free segment list* records the available free segments. Initially, the server reads segment headers from flash memory to identify free segments to construct the free segment list at server startup time. When

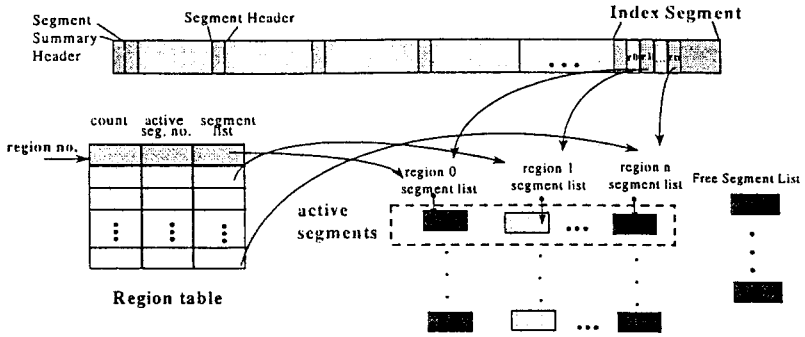


Figure 7: Region table and region segment lists.

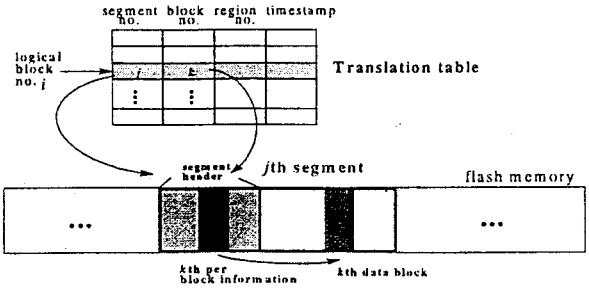


Figure 8: Translation table and address translation.

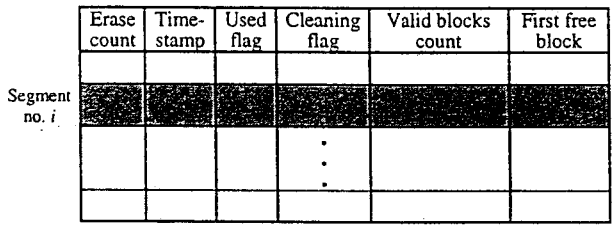


Figure 9: Lookup table to speed up cleaning.

an active segment is out of free space, a segment from the free segment list is used as the active segment and the change of active segment is written to the index segment as an appended log. When running out of free space, the index segment is erased first before wrapping around the log.

4.3 Block-based Translation Table and Address Translation

Since data blocks are not updated in place, their physical locations in flash memory change when updated. A *translation table*, shown in Figure 8, is constructed in main memory to record the physical location for each block to speed up the address translation from logical block numbers to physical addresses in flash memory. This table also records a region number for each block to indicate which region the block belongs to and a timestamp to indicate when the block is allocated in the region. These information together are used by the DAC state machine to decide whether a block's state should be switched and whether data should be migrated between neighboring regions.

During the startup time, the server reads all segment headers from flash memory to construct this translation table in main memory. When a data block is updated to another new empty block, the old block's per-block information is marked *invalid* and the new block's per-block information records the logical block number. The corresponding translation table entry is also updated to record the current physical location.

4.4 Segment-based Lookup Table

The *lookup table*, shown in Figure 9, records information about each segment, such as the number of times the segment has been erased, segment creation time, and control flags for cleaning. Initially, information is obtained by reading all segment headers from flash memory during server startup time. During run time, the server does bookkeeping of *valid blocks count* to count the number of valid blocks in a segment. The cleaner then uses these segment information to speed up the process of selecting segments for cleaning. The *first free block* indicates the first free block available for writing in a segment, which is used to speed up block allocation.

5. EXPERIMENTAL RESULTS

We have implemented the server on Linux Slackware96 in GNU C++. Table 2 lists the experimental environment. To measure the effectiveness of alternate cleaning policies, three policies were implemented in the server:

- greedy policy (**Greedy**)

  The cleaner always selects the segment with the largest amount of invalid data for cleaning.

- cost-benefit policy [14] (**Cost-benefit**)

  The cleaner chooses to clean segments that maximize the formula: $\frac{a*(1-u)}{2u}$, where u is flash memory utilization and a (age) is the time since the most recent modification.

- CAT policy [4,5] (**CAT**)

  The cleaner chooses to clean segments that minimize the

Hardware
Pentium 133 MHz with 32-Mbyte RAM
PC Card Interface Controller: Omega Micro 82C365G
Flash memory:
Intel Series 2+ 24Mbyte Flash Memory Card (segment size:128 Kbytes)
HD: Seagate ST31230N 1.0 G
Operating system:
Linux Slackware 96
(Kernel version: 2.0.0,PCMCIA package version: 2.9.5)

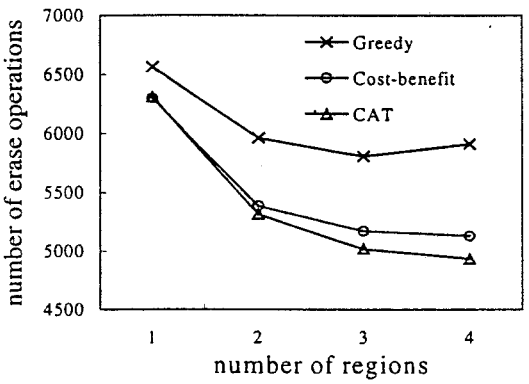
Table 2: Experimental environment.

formula: $\frac{u}{(1-u)*a} * t$, where u is utilization, a is segment age, and t is the number of times the segment has been erased.

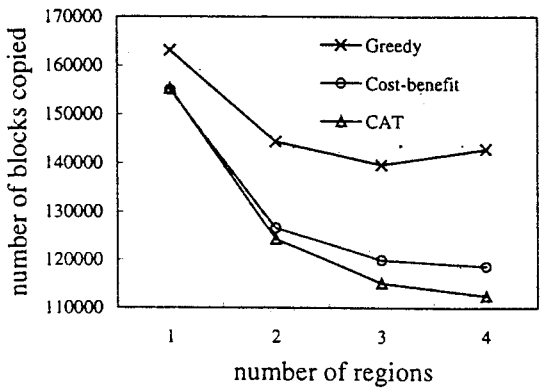
A synthetic workload combining random access and locality access was created. The workload contained 4-phase data accesses: the first and third phases were locality accesses in which 90% of accesses were to 10% of data; the other phases were random accesses. Since read operations do not incur cleaning, the workload focused on data updates that incurred invalidation of old blocks, writing of new blocks, and cleaning. Each phase contained 40-Mbyte write references and totally 160-Mbyte data were written to flash memory in 4-Kbyte units.

The block size the server managed is 4 Kbytes. The number of states that DAC state machine was configured ranged from 1 to 4. The time threshold for state switching was set to 30 minutes. Since at low utilization cleaning overhead does not significantly affect performance [14], in order to evaluate cleaning effectiveness, we initialized the flash memory by writing blocks sequentially to fill it to 90% of flash memory space for each measurement. Benchmarks were created to overwrite the initial data according to the synthetic workload.

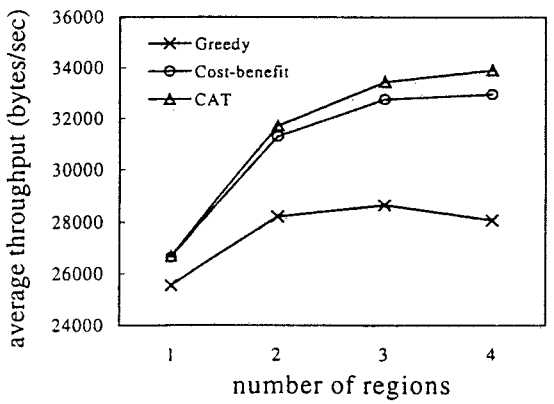
Figure 10 shows that applying DAC data clustering (i.e., the number of regions is more than 1) is beneficial for each policy. The numbers of erase operations were reduced by 15.8-21.83% for CAT, 14.54-18.57% for Cost-benefit, and 9.19-11.5% for Greedy, as shown in Figure 10(a). The numbers of blocks copied were reduced by 11.46-14.36% for Greedy, 18.33-23.43% for Cost-benefit, and 19.96-27.55% for CAT, as shown in Figure 10(b). Throughput improvement was 18.89-27.05% for CAT, 17.55-23.73% for Cost-benefit, and 9.9-12.16% for Greedy, as shown in



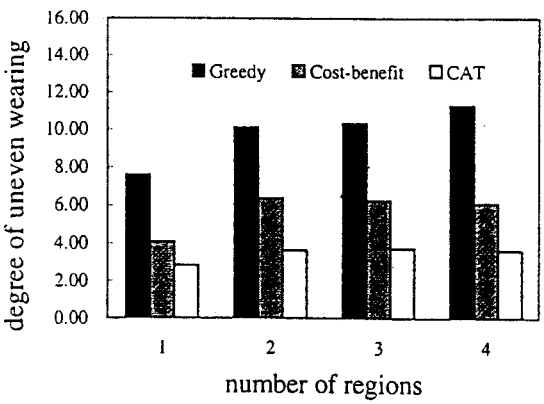
(a) Number of erase operations.



(b) Number of blocks copied during cleaning.



(c) Average throughput.



(d) Degree of uneven wearing.

Figure 10: Performance results of using DAC data clustering for various segment selection algorithms.

Figure 10(c). Among various segment selection algorithms, CAT performed best.

Since another important goal for flash memory storage systems is wear leveling, the degree of uneven wearing [5] which indicates the variance of wearing for flash segments is also used as the other metric. A utility was created to read the number of erase operations performed on individual segments from flash memory. The standard deviation of these numbers is computed as the degree of uneven wearing. The smaller the standard deviation, the more evenly the flash memory is worn. As shown in Figure 10(d), flash memory was more evenly worn for CAT. This is because only CAT considers even wearing when selecting segments to clean.

6. CONCLUSIONS

In this paper we describe the design and implementation of a storage server utilizing flash memory. The server uses the non-in-place-update approach to avoid having to erase during every update, and employs the DAC data reorganization technique for clustering frequently accessed data to reduce cleaning overhead. Data are clustered dynamically according to their write access frequencies. The CAT policy is used to reduce the number of erase operations performed and to evenly wear flash memory.

Performance evaluations show that with the CAT policy and the fine-grained DAC data clustering, the proposed storage server not only significantly reduces large amount of erase operations performed, but also evenly wear flash memory. The number of blocks copied during cleaning are significantly reduced as well. The result is extended flash memory lifetime and reduced cleaning overhead.

Several factors are important in determining how well the DAC data clustering will work in a given environment, such as the configuration for the number of states in the DAC state machine and the setting of the time threshold for state switching. In our experience, these factors are highly dependent on workloads. These factors will be examined in detailed with simulations in the future.

7.REFERENCES

[1] D. Anderson, *PCMCIA System Architecture*, MindShare, Inc. Addison-Wesley Publishing Company, 1995.

[2] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-Volatile Memory for Fast, Reliable File Systems," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992.

[3] R. Caceres, F. Douglis, K. Li, and B. Marsh, "Operating System Implications of Solid-State Mobile Computers," *Fourth Workshop on Workstation Operating Systems*, Oct. 1993.

[4] M. L. Chiang, Paul C. H. Lee, and R. C. Chang, "Managing Flash Memory in Personal Communication Devices," *Proceedings of the 1997 International Symposium on Consumer Electronics (ISCE'97)*, pp. 177-182, Singapore, Dec. 1997.

[5] M. L. Chiang and R. C. Chang, "Cleaning Policies in Mobile Computers Using Flash Memory," accepted by *Journal of Systems and Software*.

[6] R. Dan and J. Williams, 'A TrueFFS and Flite Technical Overview of M-Systems Flash File Systems', 80-SR-002-00-6L Rev. 1.30. http://www.m-sys.com/tech1.htm, Mar. 1997.

[7] B. Dipert and M. Levy, *Designing with Flash Memory*, Annabooks, 1993.

[8] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage Alternatives for Mobile Computers," *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, 1994.

[9] T. R. Halfhill, "PDAs Arrive But Aren't Quite Here Yet," *BYTE*, Vol. 18, No. 11, 1993, pp. 66-86.

[10] D. Hinds, "Linux PCMCIA HOWTO," http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-HOWTO.html, v2.5, Feb. 1998.

[11] D. Hinds, 'Linux PCMCIA Programmer's Guide', http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-PROG.html, v1.38, Feb. 1998.

[12] Intel, *Flash Memory, 1994.*

[13] Intel Corp., 'Series 2+ Flash Memory Card Family Datasheet', http://www.intel.com/design/flcard/datashts, 1997

[14] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proceedings of the 1995 USENIX Technical Conference*, Jan. 1995.

[15] B. Marsh, F. Douglis, and P. Krishnan, "Flash Memory File Caching for Mobile Computers," Proceedings of the 27 Hawaii International Conference on System Sciences, 1994.

[16] M. Rosenblum, "The Design and Implementation of a Log-Structured File System," *PhD Thesis*, University of California, Berkeley, Jun. 1992.

[17] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, Vol. 10, No. 1, 1992.

[18] SanDisk Corporation. *SanDisk SDP Series OEM Manual*, 1993.

[19] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the 1993 Winter USENIX*, 1993.

[20] P. Torelli, "The Microsoft Flash File System," *Dr. Dobb's Journal*, Feb. 1995.

[21] M. Wu and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.