Cleaning Policies in Mobile Computers Using Flash Memory[1]

Mei-Ling Chiang[†‡] Paul C. H. Lee[‡] Ruei-Chuan Chang[†‡]

Department of Computer and Information Science[†]
National Chiao Tung University,
Hsinchu, Taiwan, ROC

Institute of Information Science[‡]
Academia Sinica, Taipei, ROC

Abstract

Flash memory shows promise for use in storage devices for mobile computers. However, flash memory cannot be overwritten unless erased in advance. Erase operations are slow that usually decrease system performance, and consume power. For power conservation, better system performance, and longer flash memory lifetime, system support for erasure management is necessary. In this report, a non-update-in-place scheme is proposed to implement a flash memory server. A new cleaning policy is used to reduce the number of erase operations and to evenly wear out flash memory. The policy uses a fine-grained method to effectively cluster hot data and cold data in order to reduce cleaning overhead. Performance evaluations show that erase operations are significantly reduced and flash memory is evenly worn.

Keywords: Flash Memory, Cleaning Policy, Mobile Computer, and Embedded Systems

1. Introduction

Flash memory is nonvolatile that retains data even after power is turned off and consumes relatively little power. It provides low latency and high throughput for read accesses. Besides, flash memory is small, lightweight, and shock resistant. Because of these features, flash memory is promising for use in storage devices for mobile computers, handheld computers, and personal digital assistants (PDAs) [Ballard, 1994; Halfhill, 1993].

[1] This work is a part of ***Ramos*** project for improving flash-memory utilization and reducing cleaning costs. This report is also submitted to *Journal of Systems and Software* and is accepted for publications.

Read Cycle Time	150 ~ 250 *ns*
Write Cycle Time	6 ~ 9 *u*s/byte
Block Write Time	0.4 ~ 0.6 sec
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.

However, flash memory requires additional system support for erasure management because of the hardware characteristics [Caceres et al., 1993; Diper and Levy, 1993; Douglis et al., 1994; Intel, 1994; Intel, 1997; Kawaguchi et al., 1995; Wu and Zwaenepoel, 1994] shown in Table 1. Flash memory is partitioned into segments[2] defined by hardware manufacturers (e.g., 64 Kbytes or 128 Kbytes for Intel Series 2+ Flash Memory Cards [Intel, 1994; Intel, 1997] and 512 bytes for SanDisk flash memory cards [SanDisk, 1993]). Segments cannot be overwritten unless erased in advance. The erase operations can only be performed on full segments and are slow that usually decrease system performance and consume power. Power conservation is a critical issue for mobile computers. Segments also have limited endurance (e.g., 1,000,000 erase cycles for the Intel Series 2+ Flash Memory Cards). Therefore, erase operations must be avoided for power conservation, better system performance, and longer flash memory lifetimes. Besides, data must be written evenly to all segments to avoid wearing out specific segments to affect the usefulness of the entire flash memory, that is usually named as *even wearing* or *wear leveling*.

Since segments must be erased in advance before updating, updating data in place is not efficient.

```
Write()                               In-place-update()
{                                     {
    If new write {                        Read all data in the segment into a system buffer;
        Allocate a free block;            Update data in the system buffer;
        Write data into the free block;   Erase the segment;
    } else                                Write back all data from system buffer to segment;
        In-place-update()             }
}
```

Figure 1: Operations for updating data in place.

[2] "Segment" is used here to represent hardware-defined erase block and "block" to represent software-defined block.

In flash memory that has large segments [Intel, 1994; Intel, 1997], all data in the segment to be updated must first be copied to a system buffer and then updated. After the segment has been erased, all data must be written back from the system buffer to the segment. Figure 1 shows the detailed operations for *in-place update*. Therefore, if every update is performed in place, then performance is poor since updating even one byte requires one slow erase and several write operations, and flash memory blocks of hot spots would soon be worn out. However, storage systems cannot avoid data updating. Besides, some systems or applications exhibit locality of accesses. For example, the access behavior of a UNIX file system has such high locality that 67-78% of the writes are to metadata and most of the metadata updates are synchronous [Ruemmler and Wilkes, 1993].

To avoid having to erase during every update, updates are not performed in place in many systems [Kawaguchi et al., 1995; Torelli, 1995; Wu and Zwaenepoel, 1994]. Data are updated to empty spaces in flash memory and obsolete data are left at the same place as garbage, which a software cleaning process later reclaims. The operations of cleaning process involve three stages as shown in Figure 2. The cleaning process first selects a victim segment and then identifies *valid* data that are not obsolete in the victim segment. After valid data are migrated into another empty spaces in flash memory, the segment is erased and available for rewriting. Updating data is efficient when cleaning can be performed in the background. Figure 3 shows the detailed operations for *non-in-place update* and cleaning process.

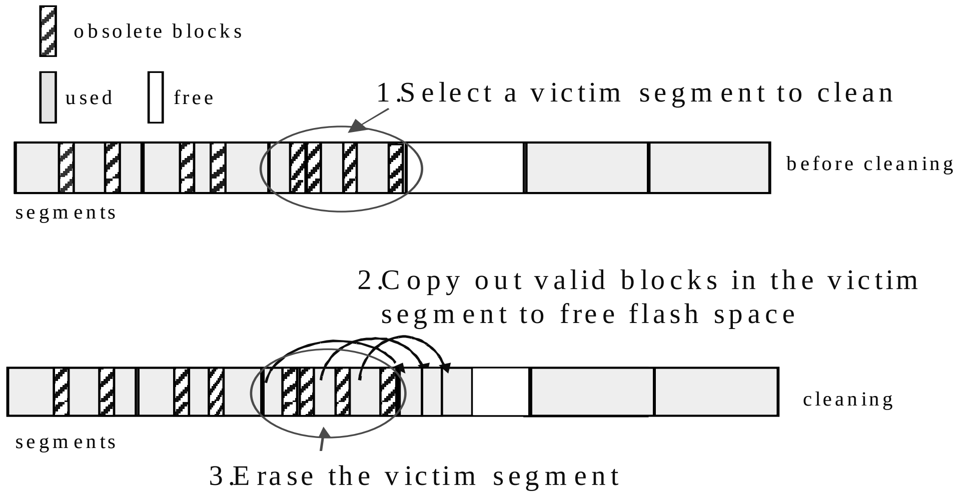

obsolete blocks
used
free
1.Select a victim segment to clean
before cleaning
segments
2.Copy out valid blocks in the victim segment to free flash space
cleaning
segments
3.Erase the victim segment

Figure 2: Three-stage operations of cleaning process.

```
Write()
{
    Allocate a free block;                                          (data placement)
    If new write
        Write data into the free block;
    else {
        /* perform the non-in-place-update */
        Mark the obsolete data as invalid;
        Write data into the free block;
    }
}

Cleaning()
{
    Select a victim segment for cleaning;                           (segment selection)
    Identify valid data in the victim segment;
    Copy out valid data to another clean flash memory spaces;       (data redistribution)
    Erase the victim segment;
    Enqueue the victim segment to free segment lists that are available for rewriting;
}
```

Figure 3: Non-in-place update and cleaning operations.

C*leaning policies* determine when to clean, which segments to clean, and where to write data. There are several cleaning policies in disk-based storage systems that use non-in-place update scheme [Blackwell et al., 1995; Matthews et al., 1997; Rosenblum, 1992; Rosenblum and Ousterhout, 1992; Seltzer et al., 1993; Wilkes et al., 1996]. They always write data sequentially as a log, or collect data and write several segments as a whole. However, flash memory is free from seek penalty and rotational latency, but has the hardware characteristics of limited endurance, bulk erase, and slow erase. Therefore, cleaning policies dedicated to flash memory were either newly proposed [Wu and Zwaenepoel, 1994] or modified from existing policies [Kawaguchi et al., 1995]. Their experimental results showed that their policies were sensitive to data access behaviors and able to reduce large number of erasures.

In this report, we propose a new cleaning policy, the **C**ost **A**ge **T**imes (CAT), to reduce the number of erase operations performed and to evenly wear flash memory. CAT differs from previous work in that CAT takes even wearing into account and selects segments for cleaning according to cleaning cost, ages of data in segments, and the number of times the segment has been erased. CAT also employs a fine-grained data clustering method to reduce cleaning overhead.

A Flash Memory Server (FMS) with various cleaning policies has been implemented to demonstrate the advantage of CAT policy. Experimental results show that CAT policy significantly reduced the number of erase operations and the overhead in cleaning, ensuring evenly wear flash memory. Under high locality of references, CAT policy outperformed the greedy policy [Kawaguchi et al., 1995; Rosenblum, 1992; Rosenblum and Ousterhout, 1992; Seltzer et al., 1993; Wu and Zwaenepoel, 1994] by 54.93%, and outperformed the cost-benefit policy [Kawaguchi et al., 1995] by 28.91% in reducing the number of erase operations performed. Trace-driven simulation was also performed to explore in detail the impact of data access patterns, utilization, flash memory size, segment size, segment selection algorithms, and data redistribution methods on cleaning. We find that data redistribution methods have the most significant impact on cleaning and have more impact than segment selection algorithms, which is less discussed in previous research. The proposed fine-grained data clustering outperformed the other methods by a large margin.

2. Issues of Flash Memory Cleaning Policies

We describe issues of cleaning policies in detail in Section 2.1 and describe the flash memory cleaning cost used to measure the effectiveness of cleaning policies in Section 2.2.

2.1 Issues of Cleaning Policies

There are many policies that control the cleaning operations:

When When is cleaning started and stopped?

Which Which segment is selected for cleaning? One may select a segment with the largest amount of garbage or select segments using information about segment data, such as age, update times, etc. This is referred to as *segment selection algorithm*.

What What size a segment should use? Segment size affects cleaning performance since the larger a segment is the more migration of live data in the segment to be cleaned.

How many How many segments should be cleaned at once? The more segments are cleaned at once, the more the valid data can be reorganized. However, cleaning several segments at once needs a large buffer to accommodate all valid data. This also delays availability of clean segments for a long time. Blocks in cleaning segments may be deleted or modified soon after cleaning; this results in useless migration.

How and where How should valid data in the cleaned segment be written out? Where is the data written out? This is referred to as *data redistribution*. There are various ways to reorganize valid data, such as enhancing the locality of future reads by grouping blocks of similar age together into new segments or grouping related files together into the same segment, etc.

Where Where are data allocated in flash memory? This is referred to as *data placement*. One may vary the allocation according to different types of data.

Therefore, we divide cleaning policies into three problem areas: segment selection, data redistribution, and data placement.

2.2 Cleaning Costs

The goal of cleaning policy is to minimize cleaning cost. The cleaning costs include erasure cost and the migration cost for copying valid data to free spaces in other segments according to the formula:

$$\textit{Cleaning Cost}_{\textit{Flash Memory}} = \textit{Number of Erase} * (\textit{Erase Cost} + \textit{Migrate Cost}_{\textit{valid data}}).$$

The cost of each erasure is constant regardless of the amount of valid data in the segment being cleaned, while migration cost is determined by the amount of valid data in the segment being cleaned. The larger the amount of valid data is, the higher the migration cost. However, the cost to erase a segment is much higher than to write a whole segment. The erasure cost dominates the migration cost in terms of operation time and power consumption. Therefore, the number of erase operations determines the cleaning costs. For better performance, longer flash memory lifetime, and power conservation, the primary goal is to minimize the number of erase operations. The second goal is to minimize the number of blocks copied during cleaning. This is different from the goal of cleaning policies in disk-based and RAM-based systems, which have no extra erase operations at all.

3. Flash Memory Cleaning Policies

The existing cleaning policies are introduced in Section 3.1 and the proposed CAT policy is presented in Section 3.2.

3.1 Existing Cleaning Policies

There are several segment selection algorithms. The *greedy* policy always selects segments with the largest amount of garbage for cleaning, hoping to reclaim as much space as possible with the least cleaning work. The *cost-benefit* policy [Kawaguchi et al., 1995] chooses to clean segments that maximize the formula: $\frac{age * (1-u)}{2u}$, where u is segment utilization and (1-u) is the amount of free space reclaimed. The *age* is the time since the most recent modification (i.e., the last block invalidation) and is used as an estimate of how long the space is likely to stay free. The cost of cleaning a segment is *2u* (one *u* to read valid blocks and the other *u* to write them back). In cost-benefit policy for disk, the cost is (*1+u*) as described in [Rosenblum, 1992; Rosenblum and Ousterhout, 1992; Seltzer et al., 1993]. Though greedy policy works well for uniform access, it was shown to perform poorly for high localities of reference [Kawaguchi et al., 1995; Rosenblum, 1992; Rosenblum and Ousterhout, 1992; Seltzer et al., 1993; Wu and Zwaenepoel, 1994]. Cost-benefit policy performs well for high localities of reference; it does not perform as well as greedy policy for uniform access [Rosenblum, 1992; Rosenblum and Ousterhout, 1992; Seltzer et al., 1993].

There are several methods to redistribute valid data in the cleaned segment. These methods assume hot data are recently referenced data that have high possibility to be accessed and then quickly become garbage. Therefore, they all try to gather hot data together to form the largest amount of garbage to reduce cleaning cost. The *age sort* used in Log-Structured File System (LFS) [Rosenblum, 1992; Rosenblum and Ousterhout, 1992; Seltzer et al., 1993] sorts valid data blocks by age before writing them out to enforce the gathering of hot data. For the better effect, several segments are cleaned at once. The *separate segment cleaning* proposed in Flash Based File System [Kawaguchi et al., 1995] uses separate segments in cleaning: one for cleaning not-cold segments and writing new data, the other for cleaning cold segments. The cold segment is defined as the cleaned segment in which utilization is less than the average utilization of file system. The separate segment cleaning was shown to perform better than when only one segment is used in cleaning, since hot data are less likely to mix with cold data.

3.2 The Proposed CAT Policy

The principle of CAT policy is to cluster data according to data type. Since a flash segment is relatively large, data blocks in a segment can be classified as three types according to their stability as shown in Table 2: read-only, cold, and hot. Read-only data once created are never modified. Cold data are modified infrequently, whereas hot data are modified frequently. Upon cleaning, before a segment is reclaimed, all valid data in the cleaned segment are migrated to empty spaces in flash memory. Those valid data may be read-only, hot, or cold. There are three possible situations for the valid data in the cleaned segment:

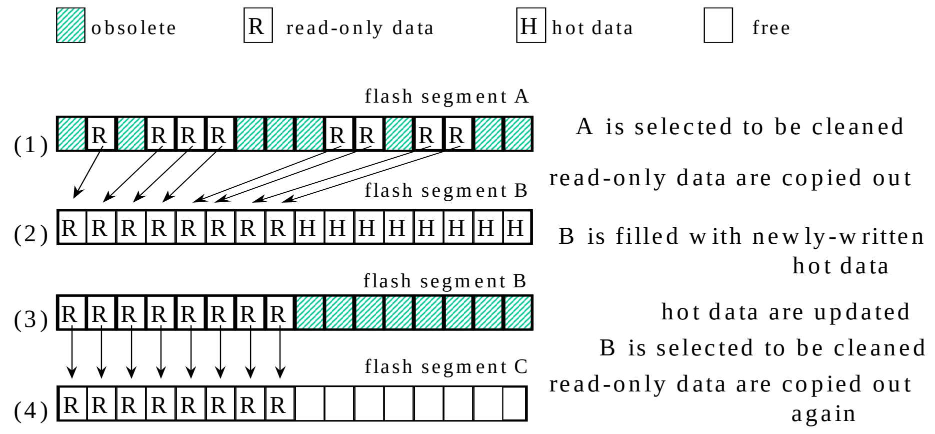

obsolete
R read-only data
H hot data
free
flash segment A
(1)
A is selected to be cleaned
read-only data are copied out
flash segment B
(2)
B is filled with newly-written hot data
flash segment B
(3)
hot data are updated
B is selected to be cleaned
flash segment C
read-only data are copied out again
(4)

Figure 4: Repeatedly migrating read-only data when they are mixed with dynamic data.

		Dynamic
Stable (slow-changing)	Read-only	Cold
Non-stable (fast-changing)	X	Hot

Table 2: Classification of data according to their stability.

- Read-only data mix with writable data

  If the cleaned segment contains read-only data, all read-only data are migrated to another new segment in flash memory. If the new segment is selected for cleaning, then those read-only data previously migrated will be migrated again. This situation is illustrated in Figure 4, in which read-only data in the cleaned segments are migrated again and again. If all read-only data are gathered and allocated in segments especially for read-only data, then segments gathered with all read-only data will never be selected for cleaning. The result is that no read-only data will be copied during cleaning process.

- Cold data mix with hot data

  If the cleaned segment contains cold data and hot data, since cold data are updated less frequently, cold data have high possibility to remain valid at the cleaning time and thus are migrated during cleaning process. Figure 5 illustrates this situation. If hot data and cold data are gathered separately so that segments are either full of all hot data or all cold data, then

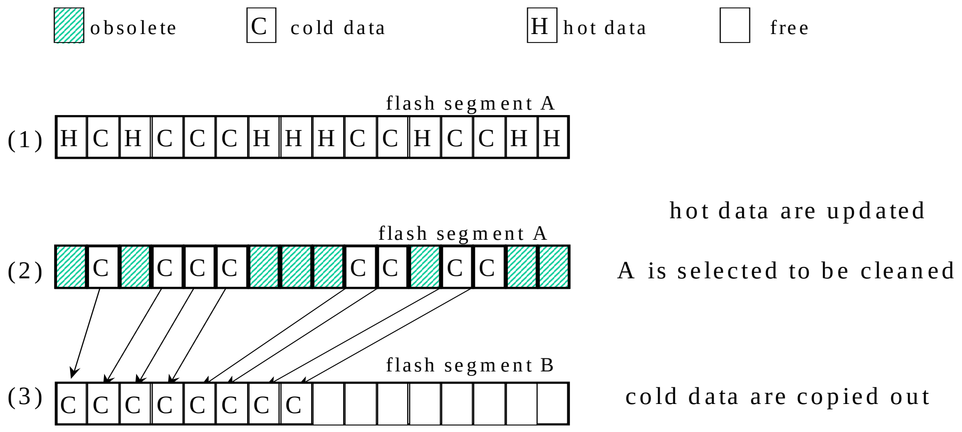


Figure 5: Migrating cold data when they are mixed with hot data.

segments containing all hot data will soon come to contain the largest amount of invalidated blocks because hot data are updated frequently and soon become invalidated. Cleaning these hot segments can minimize the migration cost since the least amount of valid data is copied and the largest amount of garbage is reclaimed.

- Data have high locality of reference

  If data in the cleaned segment exhibit high locality of reference, it is possible that these hot data are valid at the cleaning time, while soon after being migrated to another empty flash segment, these hot valid data are updated and become garbage. This situation results in *useless migration* as illustrated in Figure 6. If this segment is given more time before being cleaned, more garbage is accumulated.

Based on the above discussion, the principle of CAT policy is to cluster data according to their stability using a fine-grained way, so that segments are full of all hot data or all cold data. Especially, even wearing is considered. The basic schemes are as follows:

1. Read-only data and writable data are allocated in separate segments. No read-only data are mixed with permutable data.
2. Hot data are clustered separately from cold data. When cleaning, cold valid data in the cleaned segments are migrated to segments dedicated for cold data, while hot valid data are migrated to hot segments. New-written data are treated as hot. The *hot degree* of a block is defined as the

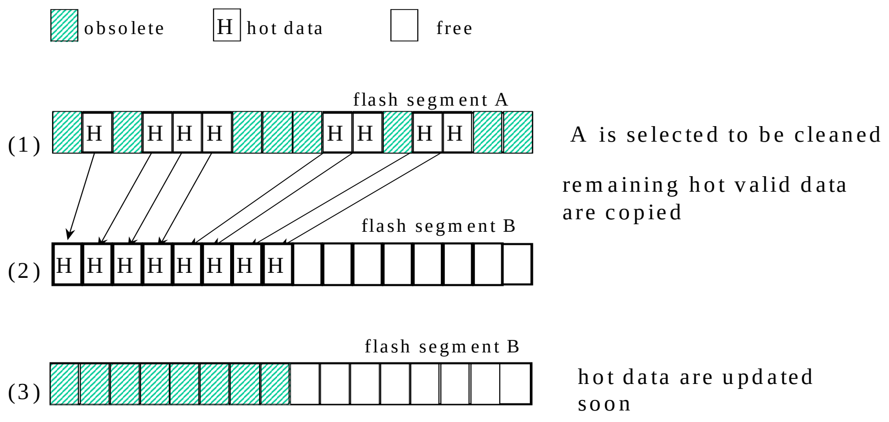

obsolete
H hot data
free
flash segment A
(1)
A is selected to be cleaned
remaining hot valid data are copied
flash segment B
(2)
flash segment B
(3)
hot data are updated soon

Figure 6: Useless migration when hot data are updated soon after being migrated.

number of times the block has been updated and decreases as the block's age grows. A block is defined as hot if its hot degree exceeds the average hot degree. Otherwise, the block is cold.

3. Evenly wearing out flash memory. When a segment is reaching its physical lifetime, we swap the segment with maximum erase times and the segment with minimum erase times to avoid wearing out specific segments.

4. The cleaner chooses to clean segments that minimize the formula:

$$\textit{Cleaning Cost}_{\textit{Flash Memory}} * \frac{1}{\textit{Age}} * \textit{Number of Cleaning},$$

called the ***C**ost **A**ge **T**imes* (CAT) formula. The *cleaning cost* is defined as the cleaning cost of every useful write to flash memory as $\frac{u}{1-u}$, 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 cleaning cost is similar to Wu and Zwaenepoel's definition of flash cleaning cost [Wu and Zwaenepoel, 1994], however, they did not consider erasure cost in evaluating alternate cleaning policies. The *age* is defined as the elapsed time since the segment was created. Here, age is normalized by a heuristic transformation function as shown in Figure 7 to avoid being too large to affect formula value. However, the effectiveness of a transformation function depends largely on workloads. 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 to avoid useless migration. In addition, to

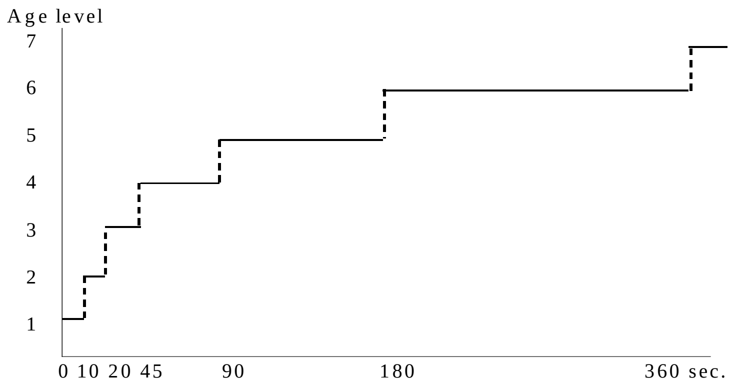


Figure 7: Age transformation function.

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.

In comparison, greedy policy considers only cleaning cost whereas cost-benefit policy considers both cleaning cost and age of the data. The CAT formula considers cleaning cost, age of the data, and number of cleaning.

4. Flash Memory Server

We have implemented a **F**lash **M**emory **S**erver (FMS) for flash memory [Chiang et al., 1997]. The FMS serves as the platform for us to build various cleaning policies on it in order to evaluate the cleaning effectiveness. The FMS manages flash memory as fixed-size blocks. The data layout on flash memory is shown in Figure 8 in which each segment has a *segment header* to describe segment information, such as the number of times a segment has been erased, *per-block information array,* etc. The per-block information describes information about every block in the segment, such as the number of times the block has been updated, flags to indicate whether a block is obsolete, etc. The *segment summary header* records information about flash memory. A *lookup table* as shown in Figure 9 is used to record segment information used by cleaner to speed up the selection of segments to be cleaned. When the number of free segments is below a certain threshold, cleaner begins to reclaim spaces occupied by obsolete data. One segment is cleaned at a time.

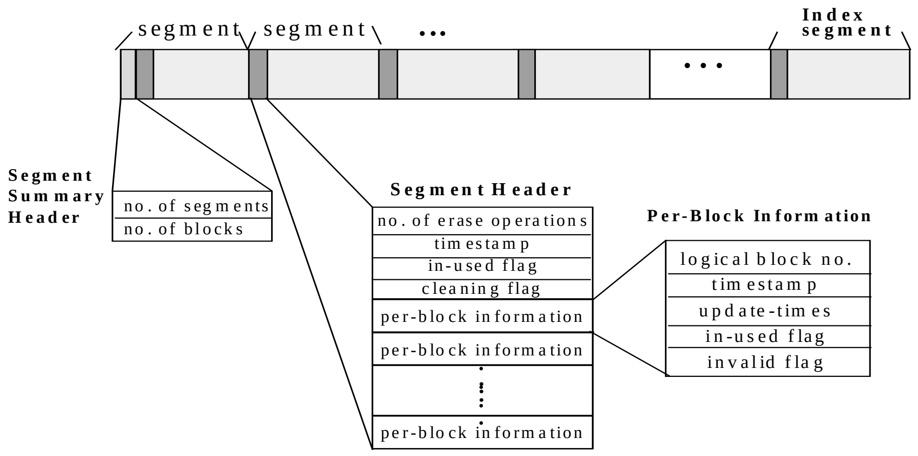


Figure 8: Data layout on flash memory.

	Erase count	Timestamp	Used flag	Cleaning flag	Valid blocks count	First free block no.
	.	.	.	.	.	.
Segment no. *i*						
	.	.	.	.	.	.

Figure 9: Lookup table to speed up cleaning.

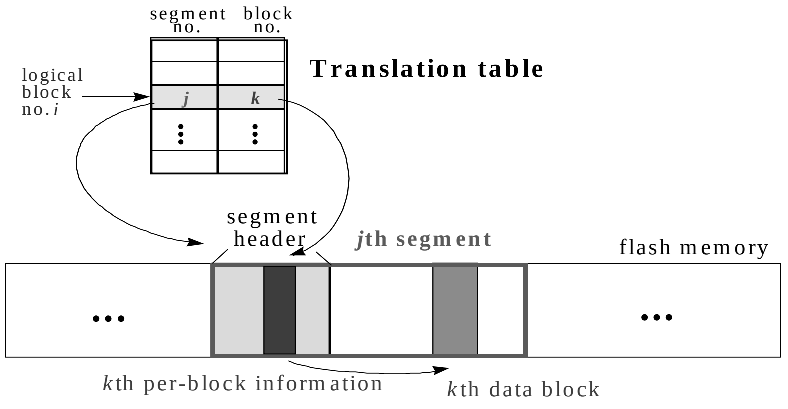


Figure 10: Translation table and address translation.

FMS uses the *non-in-place-update* scheme to manage data in flash memory to avoid having to erase during every update. Therefore, every data block is associated with a unique constant *logical block number*. As data blocks are updated, their physical locations in flash memory change. So a *translation table* as shown in Figure 10 is maintained to record blocks' physical locations in order to speed up address translation from logical block numbers to physical addresses in flash memory. Initially, the translation table and the lookup table are constructed in main memory from segment headers on flash memory during the startup time of the FMS.

Read-only data and writable data are allocated to separate segments. During cleaning, hot valid blocks in cleaned segments are distributed to hot segments while cold valid blocks are distributed to cold segments. The FMS records segments currently used for writing in *index segment* as a triple as (read-only, hot, cold). The index segment is laid out as an appended log.

5. Experimental Results and Analysis

We have implemented our Flash Memory Server (FMS) with various cleaning policies on Linux Slackware96 in GNU C++. The FMS manages data in 4 Kbyte fixed-sized blocks. We used a 24 Mbyte Intel Series 2+ Flash Memory Card. All measurements were performed both on Intel 486 DX33 and Pentium 133 to show that slow erase is the primary bottleneck as CPU gets faster. Table 3 summaries the experimental environment. Three cleaning policies were measured: *Greedy* represents the greedy policy with no separation of hot and cold blocks; *Cost-benefit* represents the cost-benefit policy with separate segment cleaning for hot and cold segments; and *CAT* represents the CAT policy with fine-grained separation of hot and cold blocks. These policies have different segment selection

	Pentium 133 MHz	**Intel 486 DX33**
Hardware	PC Card Interface Controller: Intel PCIC Vadem VG-468	Omega Micro 82C365G
	Flash memory: Intel Series 2+ 24Mbyte Flash Memory Card (segment size:128 Kbytes)	
	RAM: 32 Mbytes	
	HD: Seagate ST31230N 1.0 G	
Operating system	Linux Slackware 96 Kernel version: 2.0.0, PCMCIA package version: 2.9.5	

Table 3: Experimental environment.

algorithms and data redistribution methods.

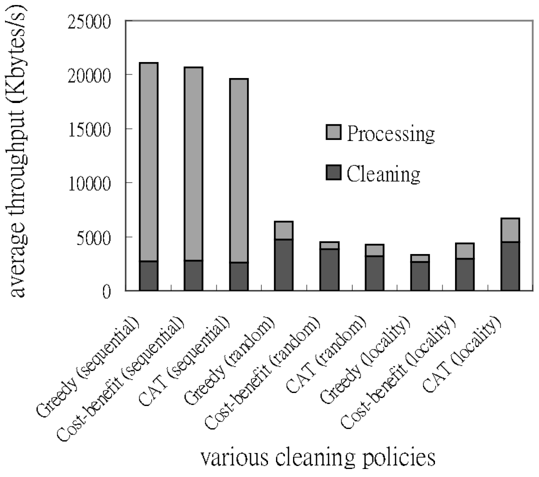
Since at low utilization cleaning overhead does not significantly affect performance [Kawaguchi et al., 1995], in order to evaluate cleaning effectiveness, we initialized the flash memory by writing blocks sequentially to fill it to 90% of flash memory spaces. The created benchmarks then updated the initial data according to the required access patterns, such as sequential, random, and locality accesses. The workload of locality of reference was created based on the *hot-and-cold* workload used in the evaluation of Sprite LFS cleaning policies [Rosenblum, 1992; Rosenblum and Ousterhout, 1992]. A total of 192 Mbyte data were written to the flash memory in 4 Kbyte units. All measurements were run on a freshly start of the system, averaging four runs.

We found that CAT policy significantly reduced the number of erase operations performed and blocks copied, as described in Section 5.1. Flash memory was also more evenly worn. As the locality of reference and flash memory utilization increased, CAT policy outperformed the other policies by a large margin, as shown in Section 5.2 and 5.3.

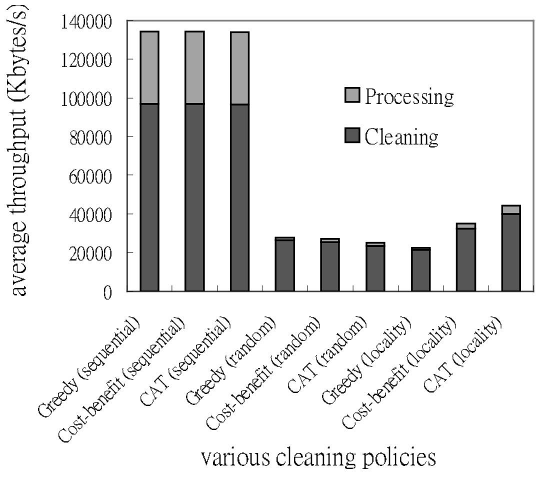
5.1 Performance of Various Cleaning Policies

Table 4 shows the performance results. For sequential access, each policy performed equally well and no blocks were copied, since sequential updates cause invalidation of each block in the cleaned segment. For random access, Greedy policy performed best and the policies for locality (Cost-benefit and CAT) performed similarly. CAT policy incurred only 1.94% more erase operations than Greedy policy. The average throughput of CAT policy was slightly less than Greedy policy and Cost-benefit policy because CAT policy incurred more processing overhead.

For high locality of reference in which 90% of the write accesses went to 10% of the initial data, CAT policy performed best, incurring 54.93% fewer erase operations than Greedy policy, and 28.91% fewer than Cost-benefit policy. This is because Greedy policy does not distinguish hot data from cold data and so it is possible that data get mixed. The performance advantage over Cost-benefit policy is because CAT policy uses a more fine-grained way to cluster hot data and cold data. CAT policy operates at block granularity whereas Cost-benefit policy operates at segment granularity. Therefore, CAT policy had the highest average throughput, 95.16% higher than Greedy policy and 26.54% higher than Cost-benefit policy. CAT policy incurred 64.59% fewer blocks copied than Greedy policy and 38.28% fewer than Cost-benefit policy. This result suggests that CAT policy can be applied to other storage systems, such as LFS [Rosenblum, 1992; Rosenblum and Ousterhout, 1992] that concerns only to reduce the number of blocks copied in cleaning.



(a) Intel 486 DX 33.



(b) Pentium 133.

Figure 11: Breakdown of elapsed time.

	Greedy				**Cost-benefit**				**CAT**			
	E	B	T	D	E	B	T	D	E	B	T	D
Sequential	1567	0	21077	2.63	1568	0	20659	2.64	1568	0	19593	2.64
Random	7103	171624	6419	4.03	7252	176230	4515	3.38	7290	177414	4275	3.01
Locality	8827	225068	3340	11.85	5596	124888	4372	8.3	3978	74726	6671	5.38

(a) Intel 486.

	Greedy				**Cost-benefit**				**CAT**			
	E	B	T	D	E	B	T	D	E	B	T	D
Sequential	1567	0	134441	2.63	1568	0	134334	2.64	1568	0	133950	2.64
Random	7103	171624	27989	4.03	7265	176634	27118	3.51	7241	175891	25055	2.96
Locality	8827	225068	22680	11.08	5733	129142	34980	9.20	4138	79705	44263	5.97

(b) Pentium 133.

Table 4: Performance of various cleaning policies. 'E' is the number of erasures. 'B' is the number of blocks copied. 'T' is the average throughput (Kbytes/s). 'D' is the degree of even wearing.

To explore each policy's degree of wear leveling, a utility was created to read the number of erase operations performed on each flash segment. We then used the standard deviation of these numbers as degree of wear leveling. The smaller the standard deviation, the more evenly the flash memory was worn. As shown in Table 4, all policies had the same degree of even wearing for sequential access. CAT policy performed best for random access and locality of access, though it incurred slightly more erase operations than the other policies for random access. This is because only CAT policy considers even wearing when selecting segments to clean. The segments seldom erased are given more chances to be selected.

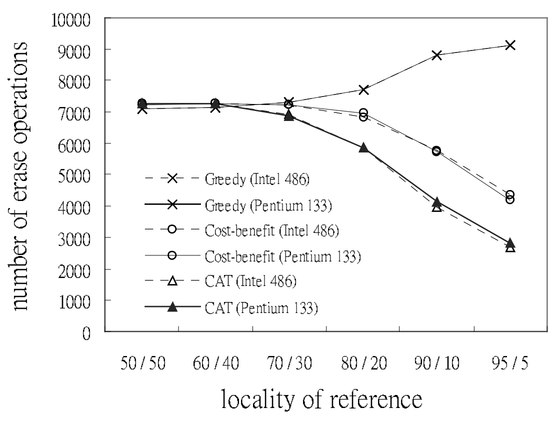
Figure 11 shows the breakdown of elapsed time: as CPU gets faster, a much greater percentage of time is spent on cleaning. On Intel 486, the FMS spent averaging only 13.34% of time cleaning for sequential access, 78.43% of time for random access, and 71.91% of time for high locality of reference, as shown in Figure 11(a). On Pentium 133, the FMS spent averaging 72.06% of time cleaning for sequential access, 94.42% of time for random access, and 92.90% of time for high locality of reference, as shown in Figure 11(b). The results show that the slow erase is the primary bottleneck as CPU gets faster. Though fine-grained data clustering needs more processing time, CAT policy significantly increases throughput by eliminating a large number of erase operations and blocks copied.

5.2 Impact of Locality of Reference

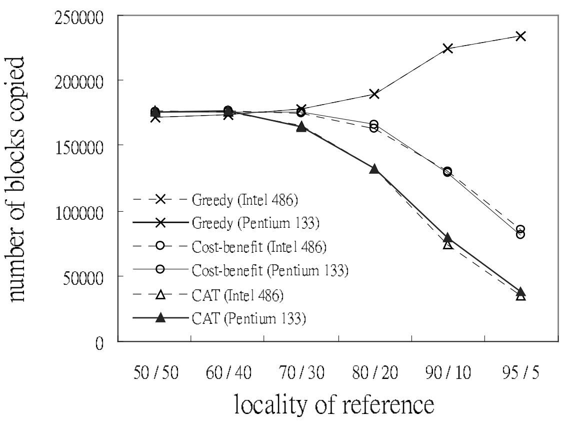
Figure 12 shows the results of varying localities of reference. Throughout this report, we used the notation for locality of reference as "x/y" that x% of all accesses go to y% of the data while (1-x)% go to the remaining (1-y)% of data. CAT policy performed best when locality was above 60/40. As the locality was increased, the performance of CAT policy increased rapidly whereas the Greedy policy deteriorated severely. The performance gap widened dramatically as well. This is because CAT policy uses fine-grained methods to separate data, so cold data are less likely to mix with hot data when contrasted with the other policies. This effect is more prominent under high locality of reference. When locality was 95/5, the number of erase operations performed by CAT policy was 69.16% less than Greedy policy and 33.22% less than Cost-benefit policy, as shown in Figure 12(a). The number of blocks copied by CAT policy was 83.55% less than the Greedy policy and 52.97% less than Cost-benefit policy, as shown in Figure 12(b). The throughput was 201.11% higher than Greedy policy and 36.98% higher than Cost-benefit policy, as shown in Figure 12(c). CAT also performed best in the degree of wear leveling for various localities, as shown in Figure 12(d).

5.3 Impact of Flash Memory Utilization

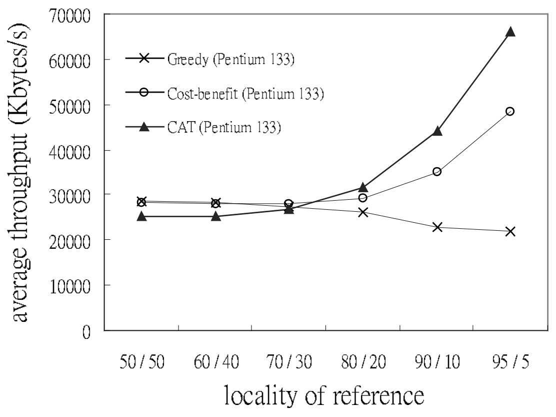
This experiment measured the cleaning effectiveness of various policies for varying utilization. In each test, the flash memory was filled with the desired percentage of data first, and then 192 Mbytes of data were overwritten to the initial data in 4 Kbyte units. Figure 13 shows that performance decreased as utilization increased since less free space was left and more cleaning had to be done. For sequential access and random access, all policies performed similarly. However, for high locality of reference, Greedy policy degraded dramatically as utilization increased while CAT policy degraded



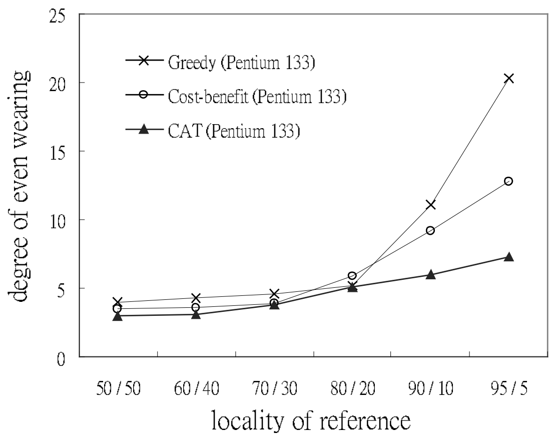
(a) Number of erasing operations performed.



(b) Number of blocks copied.



(c) Average throughput.



(d) Degree of even wearing.

Figure 12: Varying locality of reference.

much more gradually. CAT policy performed best for various degree of utilization, especially in high locality of reference.

6. Simulation Studies

In order not to be restricted to specify flash memory, simulation was performed to get more general results in this section. Furthermore, in order to examine cleaning issues in a controlled way and to explore in detail the impact of data access patterns, utilization, flash memory size, segment size, segment selection algorithms, and data redistribution methods on cleaning, we used trace-driven simulation to identify the most critical factors. The simulator and workloads are introduced in Section 6.1. Section 6.2 describes the validation of simulator. Section 6.3 presents the results.

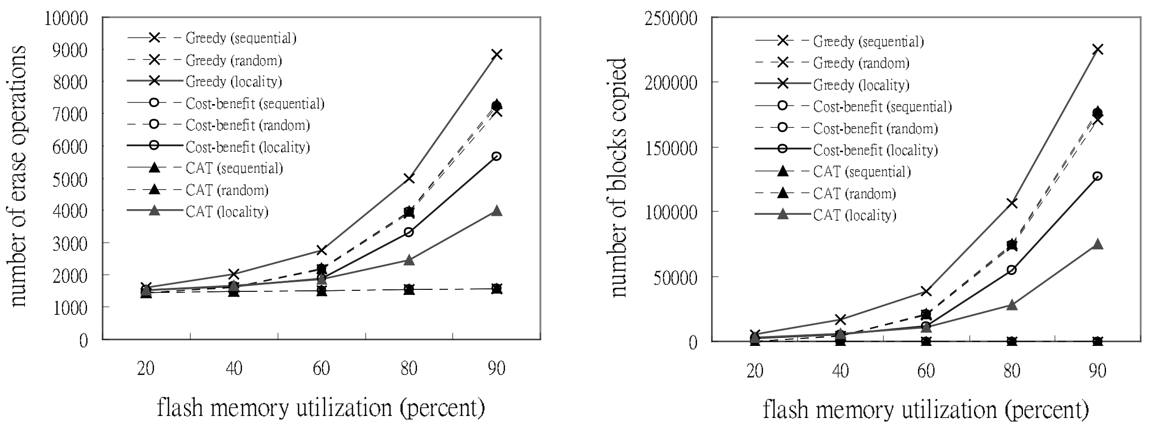
6.1 Simulator and Workloads

The simulator and workloads are described in Section 6.1.1 and 6.1.2, respectively.

6.1.1 Simulator

The simulator comprises about 20 K lines of C++ codes to completely simulate the FMS server except that the simulator stores data in a large memory array instead of in flash memory. The simulator is sufficiently flexible to accept the following parameters:

Flash size	Flash memory size.
Flash segment size	The size of an erase unit.
Flash block size	The block size that the server maintains.
Flash utilization	The amount of initial data preallocated in flash memory at the start of simulation.
Segment selection algorithm	Algorithms to select segments for cleaning.
Data placement method	A flag to control whether read-only data are allocated separately from writable data.
Data redistribution method	Parameters to control how data in the segment to be cleaned are distributed.



(a) Number of erasing operations performed. (b) Number of blocks copied.

Figure 13: Varying flash memory utilization.

Data redistribution methods		**Segment Selection Algorithms**		
		Greedy	Cost-benefit	CAT
One Segment Cleaning	Sequential writing	M1	M1	M1
	Age sorting	M2	M2	M2
	Times sorting	M3	M3	M3
Separate Segment Cleaning	Segment based	M4	M4	M4
	Block based	M5	M5	M5
	Fine-grained	M6	M6	M6

Table 5: Various cleaning policies used in simulation.

This simulator provides three segment selection algorithms and six data redistribution methods, as shown in Table 5. The total is 18 combinations. The three segment selection algorithms are greedy [Kawaguchi et al., 1995; Rosenblum, 1992; Rosenblum and Ousterhout, 1992; Seltzer et al., 1993; Wu and Zwaenepoel, 1994], cost-benefit [Kawaguchi et al., 1995], and CAT. The data redistribution methods are divided into two classes: *one segment cleaning* and *separate segment cleaning*. The one segment cleaning uses one segment for both data writing and cleaning. The separate segment cleaning treats hot data and cold data differently. During cleaning, hot valid data in the cleaned segment are distributed into hot segments, while cold valid data are distributed into cold segments. The following six data redistribution methods are examined:

M1. One segment cleaning with sequential writing
Valid data in the cleaned segment are copied out to empty flash spaces in the same order as they appear in the cleaned segment.

M2. One segment cleaning with age sorting
Valid blocks in the cleaned segment are sorted by their *age* before being copied out to empty flash spaces. The age is the elapsed time since the block was last updated. The youngest data are thought of as the hottest. Age sorting is used in LFS [Rosenblum, 1992; Rosenblum and Ousterhout, 1992; Seltzer et al., 1993].

M3. One segment cleaning with times sorting
Valid blocks in the cleaned segment are sorted by their *hot degree* before being copied out to empty flash spaces. The hot degree of a block is determined by the number of times the block has been updated but decreases as the block's age grows.

M4. Separate segment cleaning with segment-based separation for hot and cold segments

Two segments are used in which one is for cleaning cold segments and one is for data writing and cleaning not cold segments. If the utilization of a cleaned segment is less than the average utilization, then the valid data blocks in the cleaned segment are defined as cold. This method is used in [Kawaguchi et al., 1995].

M5. Separate segment cleaning with block-based separation for hot and cold blocks
Two segments are used in which one is for cleaning cold blocks and one is for data writing and cleaning hot blocks. A block is defined as hot if the number of times it has been updated exceeds the average.

M6. Separate segment cleaning with fine-grained separation for hot and cold blocks
This method is similar to M5 but a block is defined as hot if its hot degree is larger than the average hot degree. The hot degree of a block is determined by the number of times the block has been updated but decreases as the block's age grows.

6.1.2 Workload

The HP I/O traces [Ruemmler and Wilkes, 1993] and generated workloads were used to drive the simulator. The HP traces are disk-level traces of HP-UX workstation collected by Ruemmler and Wilkes [Ruemmler and Wilkes, 1993] at Hewlett-Packard. We used only the traces from personal workstation (**hplajw**), which was used primarily for electronic mail and document editing. Because the usage behavior of personal computers is likely to be similar to mobile computers, hplajw traces were often used in simulations of mobile computers [Douglis et al., 1994; Li, 1994; Li et al., 1994, Marsh et al., 1994].

However, the disks (334 Mbytes) used in hplajw are much larger than flash memory. Therefore, the traces were preprocessed to map flash memory spaces before simulation. The original traces exhibit such high locality of reference that 71.2% of writes were to metadata [Ruemmler and Wilkes, 1993]. Locality is possibly affected by this mapping. Traces from root file system in hplajw were used and contain 1364 Mbytes of references.

Because hplajw traces exhibit high locality of reference, we wanted to know whether CAT policy performs well for other access patterns. A workload generator generated the workloads for sequential, random, and locality accesses. The workload of locality of reference was created based on the *hot-and-cold* workload used in the evaluation of Sprite LFS cleaning policies [Rosenblum, 1992; Rosenblum and Ousterhout, 1992]. A total of 192 Mbytes of data were written in 4 Kbyte units. The interarrival rate of requests was Poisson distribution. For simplicity, this simulator assumed each request could be finished before the arrival of next request.

6.2 Simulator Validation

To validate the simulator, we performed the same experiments as in Section 5.1 both on the simulator and on the FMS. The generated workloads were used. Table 6 shows that all simulated performance data were within only a few percentage of actual performance. This suggests that our simulator is quite accurate in examining the cleaning effectiveness.

6.3 Performance Results

In Section 6.3.1, we show that data redistribution is the most important factor affecting cleaning effectiveness. The fine-grained separate segment cleaning performed best in reducing the cleaning overhead. The CAT policy had the best degree of even wearing. Much impact, such as the flash memory utilization, flash memory size, flash segment size, and locality of reference, was examined in detail as described in Section 6.3.2, 6.3.3, 6.3.4, and 6.3.5, respectively.

6.3.1 Performance results for HP traces

The simulated flash memory was a 24 Mbyte Intel Series 2+ Flash Memory Card with 128 Kbyte erase segment size. The server maintained data in 1 Kbyte blocks. We first wrote enough blocks in sequence to fill the flash memory to 85% of flash memory spaces, and then hplajw traces were used in the simulation.

Figure 14 shows that there were large performance gaps between one segment cleaning class (M1, M2, and M3) and separate the segment cleaning class (M4, M5, and M6). When one segment cleaning was used, there was no much performance difference among segment selection algorithms. However, when separate segment cleaning was used, performance was greatly improved: 60.04% of erase operations were reduced for Greedy policy, 64.99% for Cost-benefit policy, and 63.56% for

		Sequential		**Random**		**Locality**	
		Number of erasures	Number of copied blocks	Number of erasures	Number of copied blocks	Number of erasures	Number of copied blocks
Greedy	actual	1567	0	7103	171624	8827	225068
	simulated	1567	0	7075	170735	8834	225283
Cost-benefit	actual	1568	0	7265	176634	5733	129142
	simulated	1568	0	7237	175769	5666	127049
CAT	actual	1568	0	7241	175891	4138	79705
	simulated	1568	0	7295	177562	3987	75005

Table 6: Validation for simulated performance.

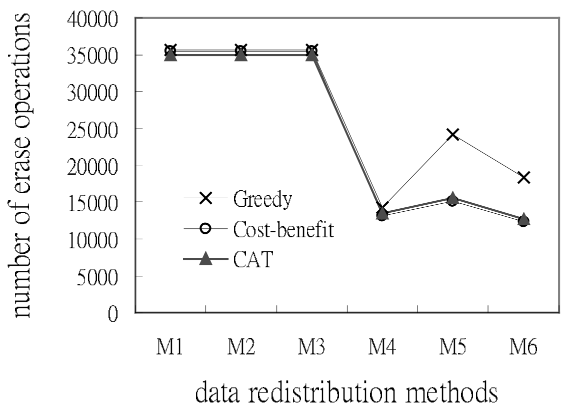
CAT policy, as shown in Figure 14(a). 86.04% of blocks copied were reduced for Greedy policy, 93.44% for Cost-benefit policy, and 91.88% for CAT policy, as shown in Figure 14(b). For Cost-benefit policy, M6 outperformed M4 by 5.55% in reducing the number of erase operations and 31.03% in reducing the number of blocks copied. For CAT policy, M6 outperformed M4 by 5.64% in reducing the number of erase operations and 27.93% in reducing the number of blocks copied. This is because fine-grained method is more effective than segment-based method in separating hot and cold data. Furthermore, flash memory was more evenly worn for separate segment cleaning. As shown in Figure 14(c), CAT policy had the best degree of even wearing. Greedy policy performed worst. M5 did not perform as well as M4 and M6. This result suggests that it is not appropriate to use only the number of update times to represent the hot degree of a block. The age of data should be taken into account.

We conclude that data redistribution methods have more impact on the cleaning effectiveness than segment selection algorithms. Separate segment cleaning can largely reduce the number of erase operations performed and the number of blocks copied. M6 is more effective than M4 in separating hot and cold data. To achieve the best performance, an effective data redistribution method must be used with an effective segment selection algorithm.

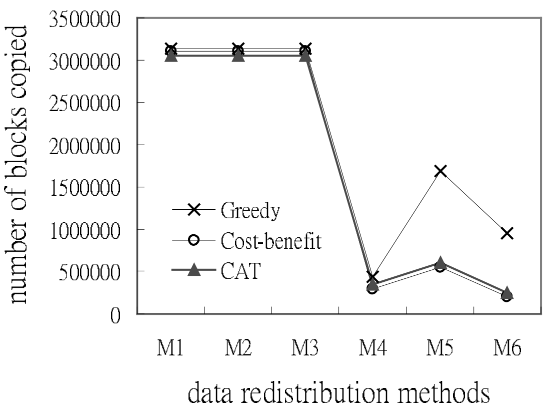
6.3.2 Impact of Flash Memory Utilization

To find out how performance varied for varying utilization, we wrote blocks in sequence to fill the flash memory to various level of utilization before simulation, then hplajw traces were used. The flash memory was 24 Mbytes with 128 Kbyte erase segments. Block size was 1 Kbytes. When only one segment was cleaned at a time, M1, M2, and M3 had the same performances; therefore, only the performance of M1 was displayed.

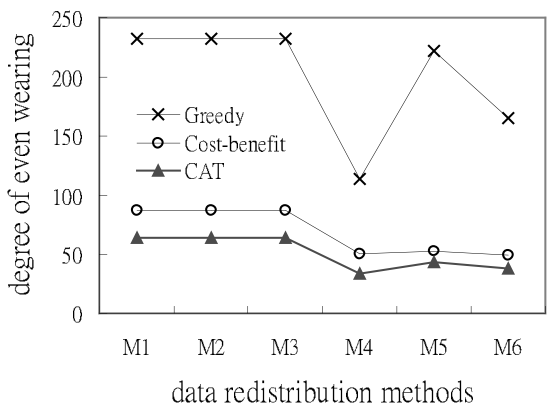
Figure 15 shows the results of varying utilization. For each policy, performance decreased as the increase of utilization since less free space was left and more cleaning was needed. However, no matter which segment selection algorithm was used, as utilization increased, the performance of one segment cleaning decreased dramatically while separate segment cleaning decreased slightly. This is because hot data and cold data were less likely to be mixed for M4 and M6. Greedy policy performed worst. The amount of blocks copied was significantly reduced as well, as shown in Figure 15(b). The results show that M6 is the most effective data redistribution method among different flash-memory utilization.



(a) Number of erasing operations performed.



(b) Number of blocks copied.

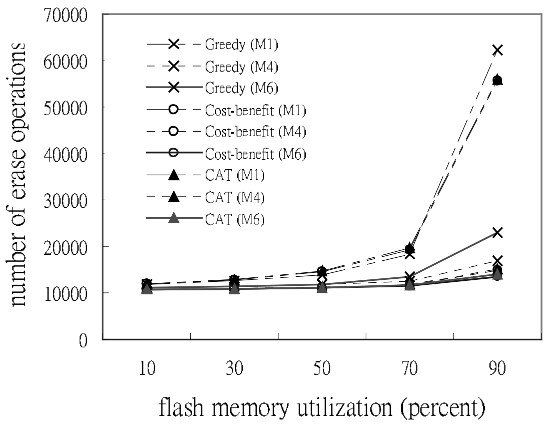


(c) Degree of even wearing.

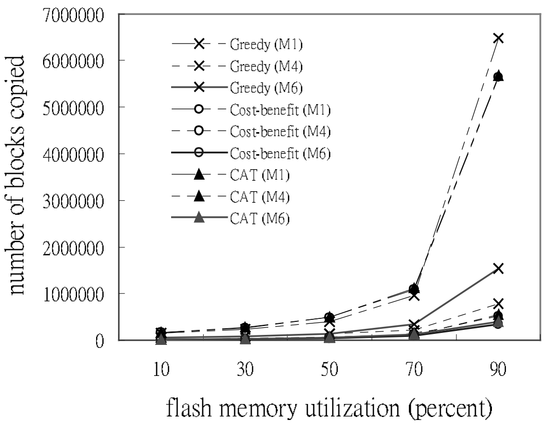
Figure 14: Performance results for HP traces.

6.3.3 Impact of Flash Memory Size

Though flash memory capacity is still small, we investigated the impact of flash memory size on cleaning. The flash memory utilization was set to 85% and segment size was 128 Kbytes. The hplajw traces were used. Figure 16 shows that as flash memory size increased, each policy performed better since much more free space was left and less cleaning was needed. When separate segment cleaning was used, each policy performed well for various flash memory sizes. M6 performed best for each segment selection algorithm. However, the performance of M1 depended largely on flash memory size no matter which segment selection algorithm was used. Therefore, large flash memory size is required for policies to perform well when one segment cleaning is used.

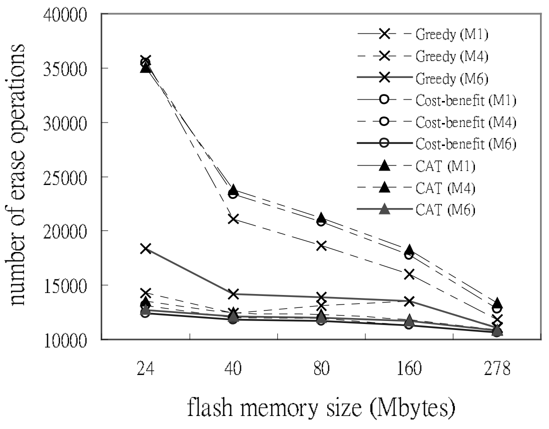


(a) Number of erasing operations performed.

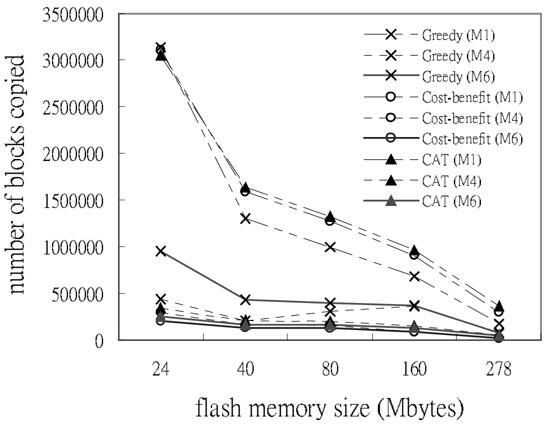


(b) Number of blocks copied.

Figure 15: Varying flash memory utilization.



(a) Number of erasing operations performed.

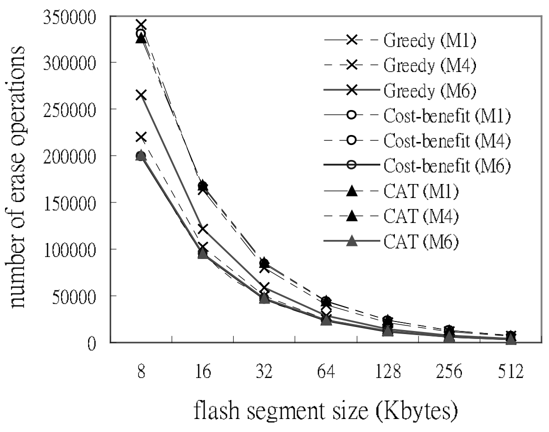


(b) Number of blocks copied.

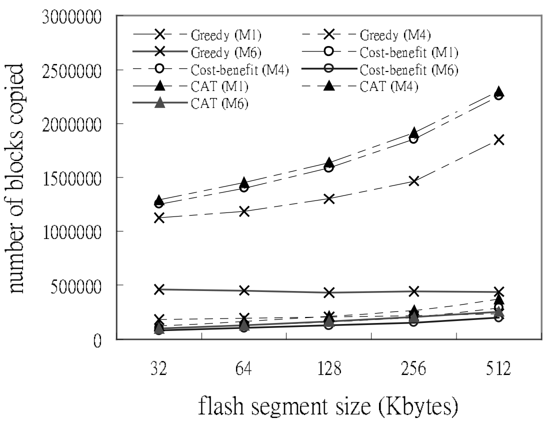
Figure 16: Varying flash memory size.

6.3.4 Impact of Erase Segment Size

Though the segment size in erasing is fixed by the hardware manufacturer, we measured the impact of erase segment size. The flash memory was 40 Mbytes and utilization was 85%. The hplajw traces were used. Figure 17 shows the results. The number of erase operations decreased at the same rate as the increase of segment size, as shown in Figure 17(a). This is because more space was reclaimed at once for rewriting as segment size was enlarged. Among all data redistribution methods, M6 incurred the least amount of erase operations. Among all segment selection algorithms, CAT policy performed best. However, as segment size became larger, more valid blocks in the cleaned segment must be copied as shown in Figure 17(b). The performance of M1 degraded dramatically while M4 and M6

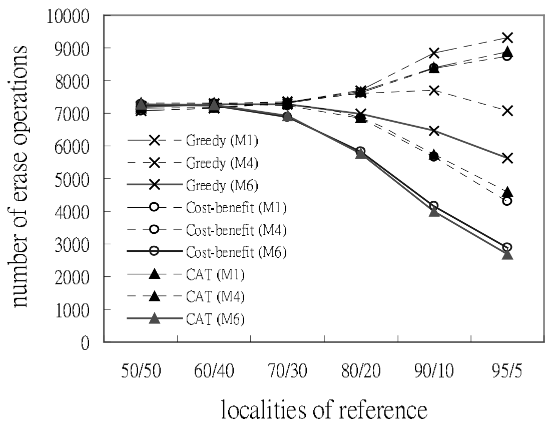


(a) Number of erasing operations performed.

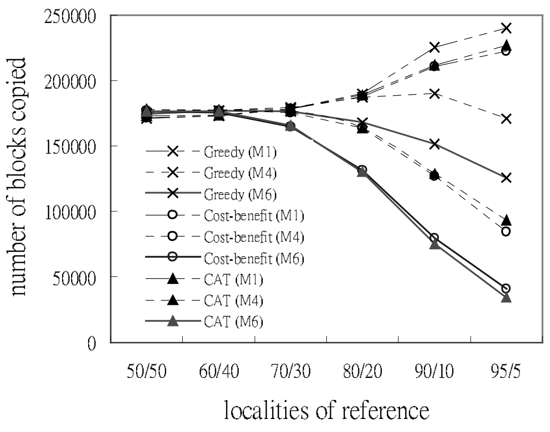


(b) Number of blocks copied.

Figure 17: Varying flash segment size.



(a) Number of erasing operations performed.



(b) Number of blocks copied.

Figure 18: Varying localities of reference.

degraded gradually. The results show that M6 is the most effective data redistribution method for various erase segment sizes.

6.3.5 Impact of Locality of Reference

We investigated the impact of locality of reference for various data redistribution methods. Therefore, generated workloads were used in this simulation. The flash memory was 24 Mbytes with 128 Kbyte segment size. The utilization was 90%. Figure 18 shows that as the increase of locality, performance gaps among data redistribution methods widened. For each segment selection algorithm, the performance of M1 decreased dramatically, while the performances of M4 and M6 increased rapidly. M4 and M6 performed better than M1 when locality was above 60/40. M6 outperformed M1 by

0.98% to 39.52% in reducing the number of erase operations performed for Greedy policy, 6.02% to 66.93% for Cost-benefit policy, and 5.63% to 69.85% for CAT policy. M6 outperformed M4 by 1.37% to 20.48% in reducing the number of erase operations performed for Greedy policy, 4.96% to 32.70% for Cost-benefit policy, and 5.04% to 41.62% for CAT policy. The number of blocks copied was significantly reduced as well. The results show that when the locality is above 60/40, M6 is the most effective method to separate hot and cold data, so that cleaning cost is the lowest. Among all, CAT policy with M6 performed best.

7. Related Work

Cleaning policies have long been discussed in log-based disk storage systems [Blackwell et al., 1995; Jonge et al., 1993; Matthews et al., 1997, Rosenblum, 1992; Rosenblum and Ousterhout, 1992; Seltzer et al., 1993; Wilkes et al., 1996]. Rosenblum [Rosenblum, 1992] suggested that the Log-Structured File System (LFS) [Rosenblum, 1992; Rosenblum and Ousterhout, 1992] can be applied to flash memory, which writes data as appended log instead of updating data in place. The *greedy* policy was shown to perform poorly under high localities of reference, so the *cost-benefit* policy was proposed in LFS. However, writing several large segments as a whole to reduce seek time and rotational latency is not necessary for a flash memory-based storage system. LFS also needs a large buffer to accommodate data blocks from several segments. This large buffer may not be affordable for a low-end resource-limited mobile computer. Their *age sort* scheme used to separate hot and cold data has limited effect when only one segment is cleaned at a time. Besides, even wearing is not an issue in LFS.

In HP AutoRAID [Wilkes et al., 1996], a two-level disk array structure, the *hole plugging* method is used in garbage collection. In hole plugging, valid data in the cleaned segment are overwritten to the other segments' holes (invalidated space which obsolete data occupy). J. N. Matthews et al. [Matthews et al., 1997] proposed an *adaptive cleaning* to combine the hole plugging into traditional LFS cleaning to adapt to changes in disk utilization. However, the holes in flash memory cannot be overwritten unless erased first.

Microsoft's Flash File System [Torelli, 1995], which uses a linked-list structure and supports the DOS FAT system, uses the greedy policy in garbage collection. Linux PCMCIA [Anderson, 1995] flash memory driver [Hinds, 1997a; Hinds, 1997b] also uses the greedy policy, but it sometimes chooses to clean the segments that are erased the fewest number of times for even wearing. However, greedy policy was shown to incur large number of erasures for high localities of reference

Issues	Flash Memory Server (FMS)	Flash-Memory Based File System	ENVy	Linux PCMCIA Package	Microsoft FFS
When	Starts when # free segments < low-water mark Stops when #free segments > high-water mark	Gradually falling threshold which decreases as # of free segments decreases	No free flash space	No free flash space	No free flash space
Which	Greedy **CAT**: **u/(1-u) * 1/age * cleaning times**	Greedy Cost-benefit: age * (1-u) / 2u	Greedy	Revised Greedy (uses Greedy most of time, sometimes selects segments erased the fewest times)	Greedy
What size (segment)	As manufacturer defines	As manufacturer defines	A segment contains several erase blocks	As manufacturer defines	As manufacturer define
How many (segments cleaned at once)	1	1	1	1	1
Where & how	* Separate segment data allocation for read-only and writable data. * Separate segment cleaning with fine-grained separation for hot and cold blocks. (the hot degree is based on update-times/age)	* Separate segment cleaning with separation for hot and cold segments.	* Locality gathering - Migrating data between neighbor segments (hot data are moved towards the bottom and cold data are moved up to the top) (lower utilization of hot segments, increase utilization of cold segments) -Locality preservation Data are flushed back to the same segment where they come from. * Hybrid cleaning - Flash memory is divided into partitions, locality gathering is used between partitions, and FIFO is used within partition.	* One segment cleaning with sequential writing.	* One segment cleaning with sequential writing.

Table 7: Comparison of various cleaning policies.

[Kawaguchi et al., 1995; Wu and Zwaenepoel, 1994].

eNVy [Wu and Zwaenepoel, 1994], a storage system for flash memory, provides transparent update-in-place scheme by using copy-on-write and page remapping techniques to avoid updating data in place. Their *hybrid cleaning* combines *FIFO* and *locality gathering* in cleaning segments. However, the locality gathering requires additional movement for valid data blocks in segments, which results in additional data blocks being copied and erasure. So its cleaning overhead is large. Additionally, eNVy considers only flash memory write cost in evaluating the effectiveness of cleaning policies. In fact, it is the erasure cost that dominates the total cleaning cost. In contrast, our work focuses on reducing the number of erase operations while evenly wearing out flash memory.

Kawaguchi et al. [Kawaguchi et al., 1995] adopts a log approach similar to LFS to design a flash memory based file system for UNIX. They used the cost-benefit policy modified from LFS with different cost. However, their results showed that cost-benefit policy incurred more erasures than greedy policy for high localities of reference. They found cold blocks and hot blocks were mixed in segments when only one segment was used in cleaning. The *separate segment cleaning* which separates cold segments and hot segments was thus proposed to clean segments. Their work did not implement wear leveling.

Kawaguchi's work motivates us that using an effective data redistribution method is more important. To obtain better cleaning effectiveness, good segment selection algorithms should be used with effective data redistribution methods. We design a new data redistribution method that uses a fine-grained method to separate cold and hot data. The method is similar to Kawaguchi's work but operates at the granularity of blocks. To further contribute to the separation of different types of blocks, read-only data and writable data are separately allocated. Furthermore, our policy takes wear-leveling into account, which selects segment based on cleaning cost, age of the data, and the number of times the segment has been erased. An even-wearing method is also proposed. As contrasted with the above, our work considers all the cleaning issues including segment selection, data redistribution, data placement, and even wearing. Table 7 summarizes the comparison.

8. Conclusions

Flash memory shows promise for use as storage for mobile computers, embedded systems, and consumer electronics. However, system support for erasure management is required to overcome the hardware limitations. In this report a new cleaning policy, the CAT policy, is proposed to reduce erasure cost and to evenly wear flash memory. The CAT policy employs a fine-grained method to cluster hot, cold, and read-only data into separate segments for reducing cleaning overhead. It provides even wearing by selecting segments for cleaning according to utilization, age of the data, and the number of erase operations performed on segments.

We have implemented a Flash Memory Server (FMS) with various cleaning policies to demonstrate the advantages of CAT policy. Performance evaluations show that CAT policy significantly reduces a large number of erase operations and evenly wears flash memory. Under high locality of reference, CAT policy outperformed greedy policy by 54.93% and outperformed cost-benefit policy by 28.91% in reducing the number of erase operations performed. The result is extended flash memory lifetime and reduced cleaning overhead.

CAT policy also outperformed greedy policy by 64.59% in reducing the number of blocks copied and outperformed cost-benefit by 38.28%. This result suggests that CAT policy can also be applied to other media storage systems such as RAM or disks that concerns only to reduce the number of blocks copied to improve cleaning performance. For example, the separate segment cleaning with fine-grained separation for hot and cold blocks scheme can be applied to LFS before segment data in buffers are written out to disk.

Trace-driven simulation was assisted to examine cleaning issues in detail. We found that data redistribution methods are the most important factor affecting cleaning effectiveness. Improving data redistribution methods is more effective than improving segment selection algorithms. Separate segment cleaning with fine-grained separation for hot and cold data shows its strength in cleaning effectiveness.

Future work can be summarized as follows. We will use real applications to extensively examine the effectiveness of the proposed cleaning policy on our FMS server. We will then do performance tuning of the FMS server and integrate it into ROSS [Chiang et al., 1997], a RAM-based Object Storage Server designed for PDAs, to enable ROSS to store data in flash storage.

Acknowledgments

We thank John Wilkes at Hewlett Packard Laboratories for his efforts in making their HP I/O traces available to us. We would also like to thank David Hinds for his valuable comments and help on our work.

References

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

Ballard, N., State of PDAs and Other Pen-Based Systems, *Pen Computing Magazine*, pp. 14-19, Aug. (1994).

Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M., 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).

Blackwell, T., Harris, J., and Seltzer, M., Heuristic Cleaning Algorithms in Log-Structured File Systems, *Proceedings of the 1995 USENIX Technical Conference*, Jan. (1995).

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

Chiang, M. L., Lee, Paul C. H., Lo, S. Y. and Chang, R. C., Design and Implementation of a Memory-Based Object Server for Hand-held Computers, *Journal of Information Science and Engineering*, vol. 13 (1997).

Chiang, M. L., Lee, Paul C. H., and Chang, R. C., Managing Flash Memory in Personal Communication Devices, *Proceedings of the 1997 International Symposium on Consumer Electronics (ISCE'97)*, Dec. 2-4 (1997).

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

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

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

Hinds, D., Linux PCMCIA HOWTO, *http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-HOWTO.html* (1997).

Hinds, D., Linux PCMCIA Programmer's Guide, *http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-PROG.html* (1997).

Intel, *Flash Memory*, 1994.

Intel Corp., Series 2+ Flash Memory Card Family Datasheet, *http://www.intel.com/design/flcard/datashts* (1997).

Jonge, W. D., Kaashoek, M. F., and Hsieh, W. C., Logical Disk: A Simple New Approach to Improving File System Performance, Technical Report MIT/LCS/TR-566, Massachusetts Institute of Technology, 1993.

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

Li, K., Towards a low power file system, Technical Report UCB/CSD 94/814, Masters Thesis, University of California, Berkeley, CA, May 1994.

Li, K., Kumpf, R., Horton, P., and Anderson, T., A Quantitative Analysis of Disk Drive Power Management in Portable Computers, *Proceedings of the 1994 Winter USENIX* (1994).

Marsh, B., Douglis, F., and Krishnan, P., Flash Memory File Caching for Mobile Computers, *Proceedings of the 27 Hawaii International Conference on System Sciences* (1994).

Matthews, J. N., Roselli, D., Costello, A. M., Wang, R. Y. and Anderson, T. E., Improving the Performance of Log-Structured File Systems with Adaptive Methods, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, Oct. 5-8 (1997).

Rosenblum, M., The Design and Implementation of a Log-Structured File System, Ph.D. Thesis, University of California, Berkeley, Jun. 1992.

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

Ruemmler, C. and Wilkes, J., UNIX Disk Access Patterns, *Proceedings of the 1993 Winter USENIX* (1993).

SanDisk Corporation, *SanDisk SDP Series OEM Manual*, 1993.

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

Torelli, P., The Microsoft Flash File System, *Dr. Dobb's Journal*, Feb. (1995).

Wilkes, J., Golding, R., Staelin, C., and Sullivan, T., The HP AutoRAID Hierarchical Storage System, *ACM Transactions on Computer Systems*, 14(1), Feb. (1996).

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