ELSEVIER

The Journal of Systems and Software 48 (1999) 213–231

The Journal of Systems and Software

www.elsevier.com/locate/jss

Cleaning policies in mobile computers using flash memory

M.-L. Chiang [a], R.-C. Chang [a,b,*]

[a] *Department of Computer and Information Science, National Chiao Tung University, Hsinchu 30050, Taiwan, ROC*
[b] *Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC*

Received 20 January 1998; received in revised form 9 May 1998; accepted 18 June 1998

Abstract

Flash memory is small, lightweight, shock-resistant, nonvolatile, and consumes little power. Flash memory therefore shows promise for use in storage devices for consumer electronics, mobile computers and embedded systems. However, flash memory cannot be overwritten unless erased in advance. Erase operations are slow that usually decrease system performance, and consume power. The number of erase cycles is also limited. For power conservation, better system performance, and longer flash memory lifetime, system support for erasure management is necessary. In this paper, we use the non-update-in-place scheme to implement a flash memory server and propose a new cleaning policy to reduce the number of erase operations needed 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. A wear-leveling algorithm is also proposed. Performance evaluations show that erase operations are significantly reduced and flash memory is evenly worn. Though the proposed fine-grained separation scheme is targeted at flash memory-based systems, it can be applied to other storage systems as well. © 1999 Elsevier Science Inc. All rights reserved.

Keywords: Flash memory; Cleaning policy; Mobile computer; Consumer electronics; Embedded system

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 consumer electronics, embedded systems, and mobile computers, such as digital cameras, audio recorders, set-top boxes, cellular phones, notebooks, handheld computers, and personal digital assistants (PDAs) (Ballard, 1994; Halfhill, 1993).

However, flash memory requires additional system support for erasure management because of the hardware characteristics (Baker et al., 1992; Caceres et al., 1993; Dipert and Levy, 1993; Douglis et al., 1994; Intel, 1994, 1997; Kawaguchi et al., 1995; Wu and Zwaenepoel, 1994) shown in Table 1. Flash memory is partitioned into segments [1] defined by the 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. This is called *even wearing* or *wear-leveling*.

Since segments must be erased in advance before updating, updating data in place is not efficient. In flash memory that has large segments (Intel, 1994, 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. Fig. 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

[*] Corresponding author. E-mail: rc@cc.nctu.edu.tw
[1] We use "segment" to represent hardware-defined erase block and "block" to represent software-defined block.

Table 1
Flash memory characteristics

Read cycle time	150 ∼ 250 ns
Write cycle time	6 ∼ 9 μs/byte
Block write time	0.4 ∼ 0.6 sec
Block erase time	0.6 ∼ 0.8 sec
Erase block size	64 or 128 Kbytes
Erase cycles per block	100,000 ∼ 1,000,000

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 Fig. 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. Fig. 3 shows the detailed operations for *non-in-place update* and cleaning process.

Cleaning 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 paper, 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.

```
Write()
{
   If new write {
      Allocate a free block;
      Write data into the free block;
   } else
      In-place-update()
}

In-place-update()
{
   Read all data in the segment into a system buffer;
   Update data in the system buffer;
   Erase the segment;
   Write back all data from system buffer to segment;
}
```

Fig. 1. Operations for updating data in place.

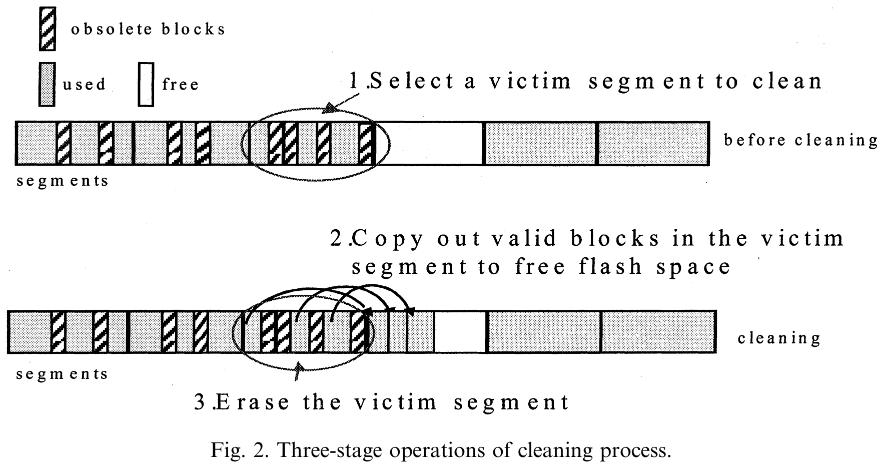

obsolete blocks
used
free
1.Select a victim segment to clean
before cleaning
segments
2.Copy out valid blocks in the victim segment to free flash space
cleaning
segments
3.Erase the victim segment

Fig. 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;
}
```

Fig. 3. Non-in-place update and cleaning operations.

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 cleaning overheads, 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.

The rest of this paper is organized as follows. Section 2 describes issues of flash memory cleaning policies and defines cleaning cost used to measure the effectiveness of cleaning policies. Section 3 presents various cleaning policies. Section 4 describes the implementation of FMS and we investigate the effectiveness of alternate cleaning policies on it. Section 5 presents the experimental results. Section 6 shows the simulation that explores in detail the effect of data redistribution methods and segment selection algorithms on cleaning. Section 7 describes related work, and Section 8 concludes this paper.

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 cost

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

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

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 cleaning cost is determined by the number of erase operations. 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: $\text{age} * (1 - u)/2u$, 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) and 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 a time. 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,

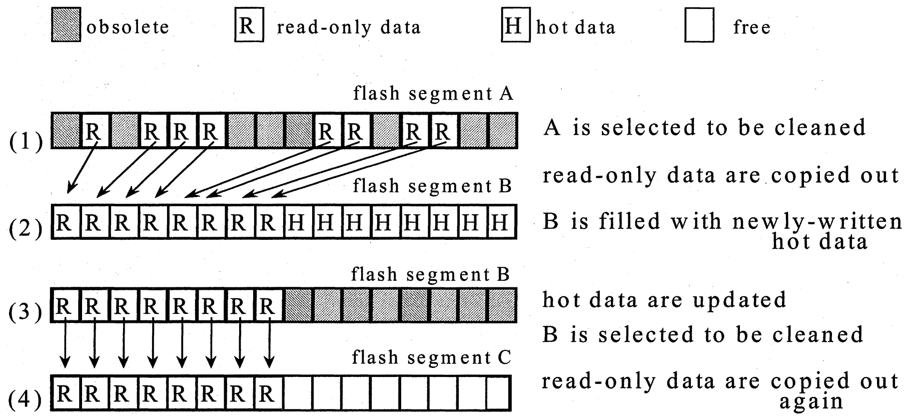
Table 2
Classification of data according to their stability

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

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:

- *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 Fig. 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. Fig. 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 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 Fig. 6. If this segment is given more time before being cleaned, more garbage can be accumulated.

Based on the above discussion, the principle of CAT policy is to cluster data according to their stability using



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
(4)
read-only data are copied out again

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

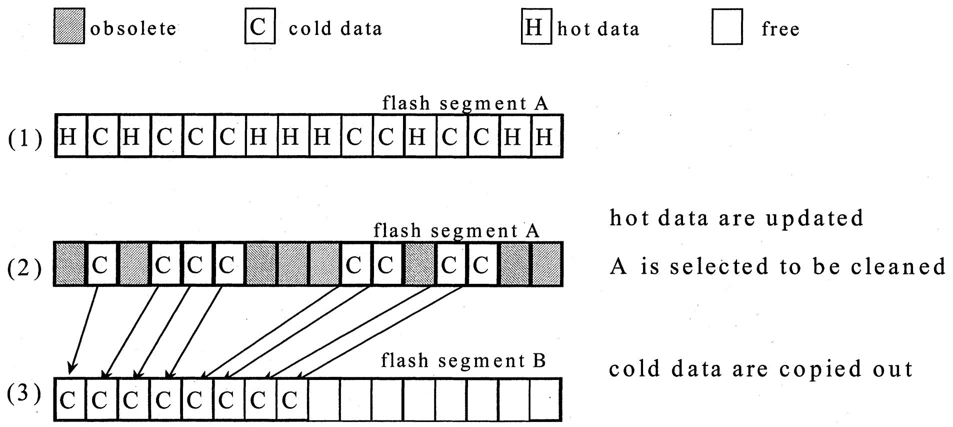


Fig. 5. Migrating cold data when they are mixed with hot data.

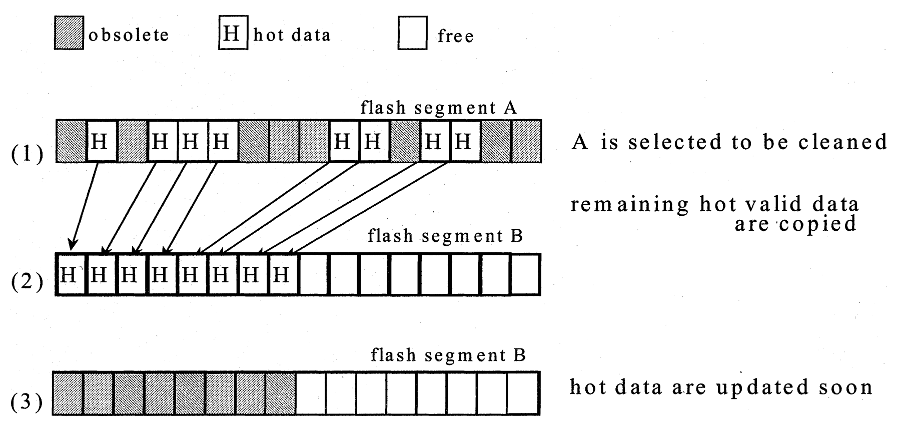

obsolete
H hot data
free
flash segment A
(1)
A is selected to be cleaned
remaining hot valid data are copied
flash segment B
(2)
flash segment B
(3)
hot data are updated soon

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

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. Data newly written are treated as hot. The *hot degree* of a block is defined as the 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:

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

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 $u/(1-u)$, 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 Fig. 7, aiming to avoid age being too large to overemphasize age and affect segment choosing. 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 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 Fig. 8 in which each segment has a *segment header* to describe segment

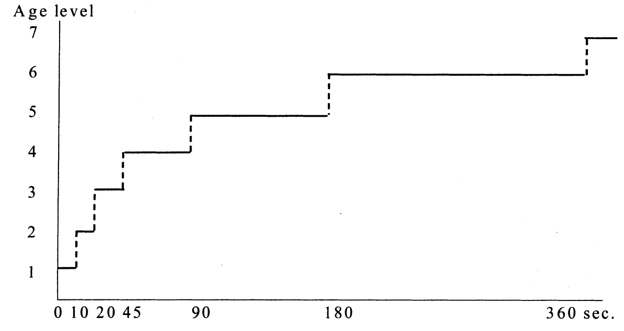


Fig. 7. Age transformation function.

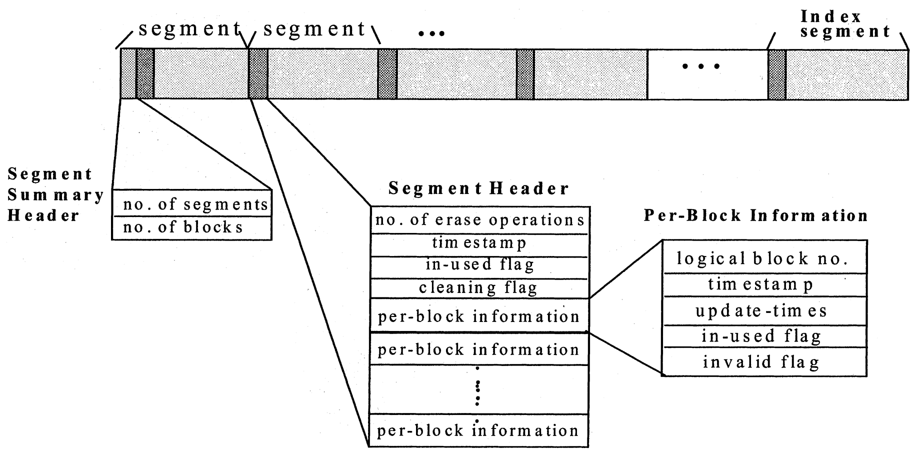


Fig. 8. Data layout on flash memory.

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 Fig. 9 is maintained to record segment information that is used by the cleaner to speed up the selection of segments for cleaning. When the number of free segments is below a certain threshold, the cleaner begins to reclaim spaces occupied by obsolete data. One segment is cleaned at a time.

The 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 Fig. 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 the cleaned segments are distributed to hot segments while cold valid blocks are distributed to cold segments. The FMS records segments that are 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-MB Intel Series 2+ Flash Memory Card. All measurements were performed both on Intel 486 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 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 space. The created

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

Fig. 9. Lookup table to speed up cleaning.

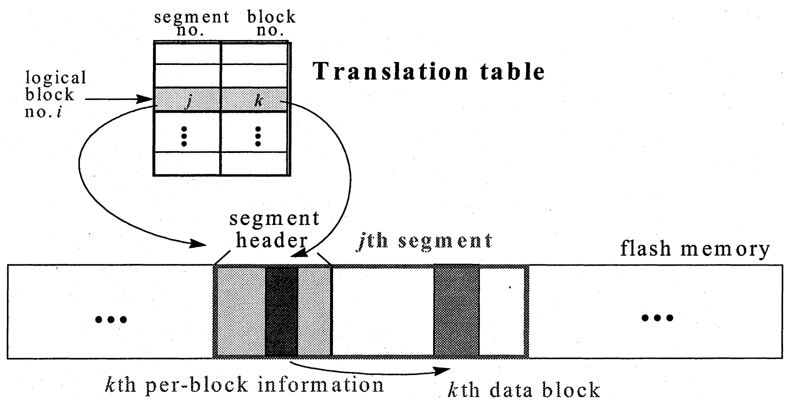


Fig. 10. Translation table and address translation.

benchmarks then updated the initial data according to the required access patterns, such as sequential, random, and locality accesses. The workload for 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 show that CAT policy significantly reduced the number of erase operations performed and blocks copied. So throughput was greatly improved 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 Sections 5.2 and 5.3.

5.1. Performance of various cleaning policies

Table 4 shows the results. We noted the number of erasures and the amount of blocks copied were slightly different between 486 and Pentium, due to the different processing speeds such that the age of data is different. The age of data affects segment selection for Cost-benefit policy and CAT policy. 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, Cost-benefit policy and CAT policy performed similarly and slightly worse than Greedy policy.

For high locality of reference in which 90% of the write accesses went to 10% of the initial data, CAT policy incurred least number of erasures, 54.93% fewer than Greedy policy and 28.91% fewer than Cost-benefit policy. CAT policy also incurred least number of blocks copied, 64.59% less than Greedy policy and 38.28% less than Cost-benefit policy. Therefore, CAT policy had the highest average throughput, 95.16% higher than Greedy policy and 26.54% higher than Cost-benefit policy. Greedy performed worst since it does not distinguish hot data from cold data, so data are possible to get mixed. CAT policy outperformed Cost-benefit policy because CAT policy is more fine-grained in separating data: CAT policy operates at block granularity whereas Cost-benefit policy operates at segment granularity.

To explore each policy's degree of wear-leveling, a utility was created to read out the number of times each segment has been erased from flash memory. We then used the standard deviation of these numbers as the degree of uneven wearing. The smaller the standard deviation, the more evenly the flash memory was worn. As shown in Table 4, CAT policy performed best since only CAT policy considers even wearing when selecting segments to clean: segments seldom erased are given more chances to be selected.

Fig. 11 shows the breakdown of elapsed time. The FMS spent much more time in cleaning when running on Intel 486 than on Pentium 133. For example, for sequential access, the FMS spent averaging only 13.34% of time cleaning on Intel 486, while averaging 72.06% of

Table 3
Experimental environment

	Pentium 133 MHz	Intel 486 DX33
Hardware	PC card interface controller: Intel PCIC Vadem VG-468	Omega micro 82C365G
	Flash memory: Intel Series 2+ 24Mbytes 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 4
Performance of various cleaning policies

	Number of erasures			Number of blocks copied			Average throughput (bytes/s)			Degree of uneven wearing		
	Greedy	Cost-benefit	CAT	Greedy	Cost-benefit	CAT	Greedy	Cost-benefit	CAT	Greedy	Cost-benefit	CAT
Intel 486												
Sequential	1567	1568	1568	0	0	0	21 077	20 659	19 593	2.63	2.64	2.64
Random	7103	7252	7290	171 624	176 230	177 414	6419	4515	4275	4.03	3.38	3.01
Locality	8827	5596	3978	225 068	124 888	74 726	3340	4372	6671	11.85	8.3	5.38
Pentium 133												
Sequential	1567	1568	1568	0	0	0	134 441	134 334	133 950	2.63	2.64	2.64
Random	7103	7265	7241	171 624	176 634	175 891	27 989	27 118	25 055	4.03	3.51	2.96
Locality	8827	5733	4138	225 068	129 142	79 705	22 680	34 980	44 263	11.85	9.20	5.97

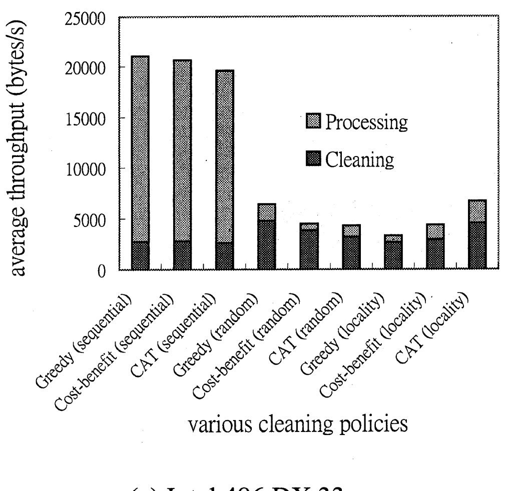
time on Pentium 133. The results show that the slow erase is the primary bottleneck as CPU gets faster. So, though fine-grained data clustering needs more CPU processing time, CAT policy still significantly increases throughput because large amount of erase operations and blocks copied are eliminated.

5.2. Impact of locality of reference

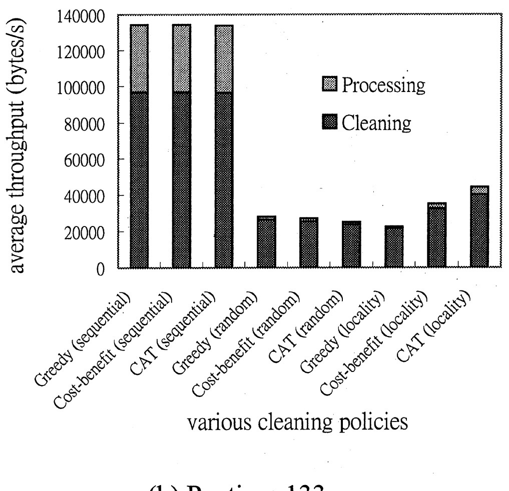
Throughout the paper, we used the notation for locality of reference as "x/y" that x% of all accesses go to y% of the data while (1−x)% goes to the remaining (1−y)% of data. Fig. 12 shows that CAT policy performed best when locality was above 60/40. As the locality was increased, the performance of CAT policy increased rapidly whereas Greedy policy deteriorated severely. This is because CAT policy uses fine-grained methods to separate data, such that cold data are less likely to mix with hot data as compared with the other policies. This effect is more prominent under higher locality of 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. The reduction of the number of blocks copied and the improvement of throughput were much more promising. The throughput was 201.11% higher than Greedy policy and 36.98% higher than Cost-benefit policy. Besides, CAT policy also performed best in the wear-leveling for various localities.

5.3. Impact of flash memory utilization

This experiment measured the impacts of flash memory utilization for various policies. In each measurement, the flash memory was initially filled with the desired percentage of data. Fig. 13 shows that performance decreased as utilization increased since less free space was left and more cleaning had to be done in order to come out enough free space. For sequential and random accesses, all policies performed similarly. However, for high locality of reference, Greedy policy degraded dramatically as utilization increased while CAT policy degraded much more gradually. CAT policy performed best for various degrees of utilization.

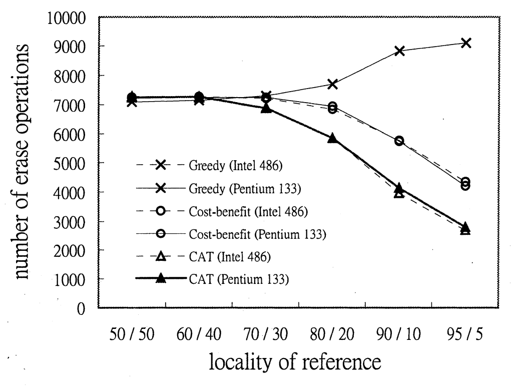


(a) Intel 486 DX 33.

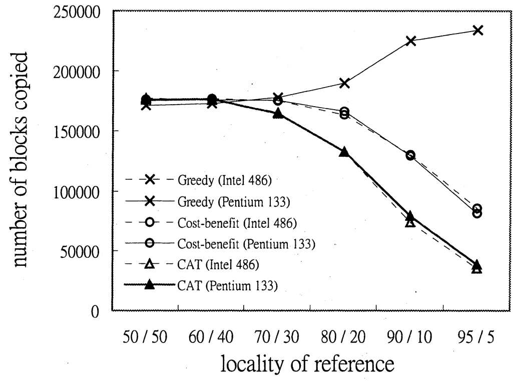


(b) Pentium 133.

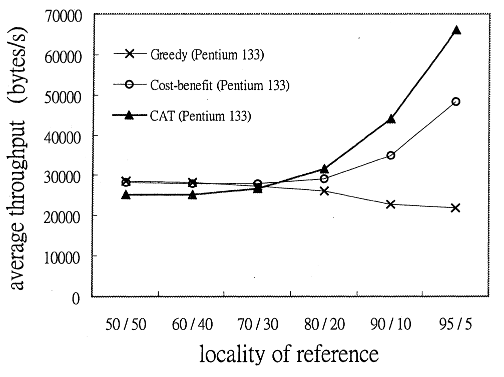
Fig. 11. Breakdown of elapsed time. (a) Intel 486 DX 33; (b) Pentium 133.



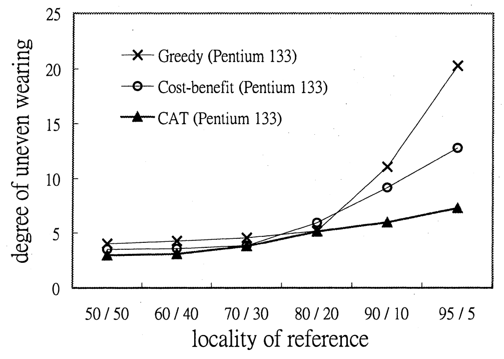
(a) Number of erase operations performed.



(b) Number of blocks copied.

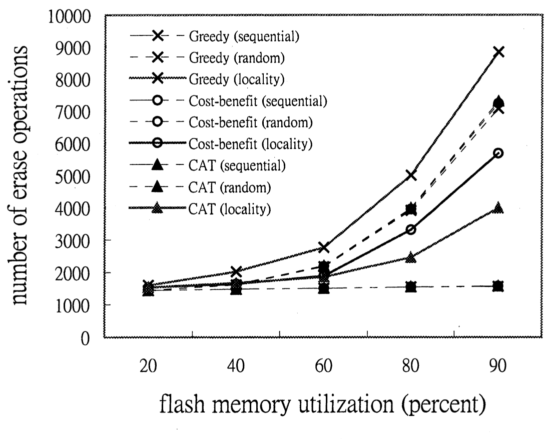


(c) Average throughput.

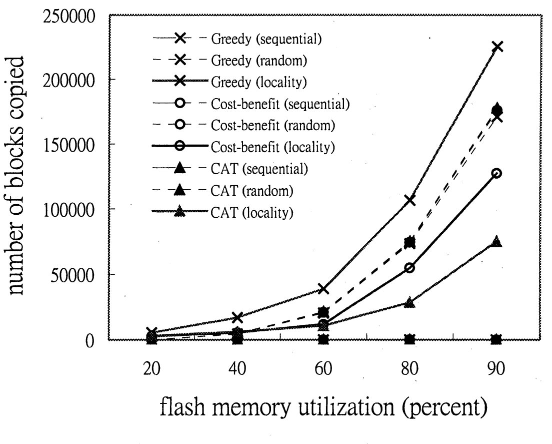


(d) Degree of uneven wearing.

Fig. 12. Varying localities of reference. (a) Number of erase operations performed; (b) Number of blocks copied; (c) Average throughput; (d) Degree of uneven wearing.



(a) Number of erase operations performed.



(b) Number of blocks copied.

Fig. 13. Varying flash memory utilization. (a) Number of erase operations performed; (b) Number of blocks copied.

6. Simulation studies

In order not to be restricted to specific flash memory, simulation was performed to get more general results. 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 the simulator. Section 6.3 presents the results.

6.1. Simulator and workloads

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

6.1.1. Simulator

The simulator completely simulates the FMS server except that it stores data in a large memory array instead of in flash memory. The simulator is 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.

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 written to empty flash space 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).

Table 5
Various cleaning policies used in simulation

Data redistribution methods		Segment selection algorithms		
		Greedy	Cost-benefit	CAT
One segment cleaning	Sequential writing		M1	
	Age sorting		M2	
	Times sorting		M3	
Separate segment cleaning	Segment based		M4	
	Block based		M5	
	Fine-grained		M6	

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.	

For simplicity, this simulator assumed each request could be finished before the arrival of next request.

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 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. Since currently flash memory capacity is still small, we do not expect flash memory contain swap space. So traces from swap space in hplajw were not included, in which write references are totally 1331 Mbytes.

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 for locality of reference was created based on the *hot-and-cold* workload (Rosenblum, 1992; Rosenblum and Ousterhout, 1992). The generated workloads cover two weeks' write activities, in which a total of 192 Mbytes of data were written in 4-Kbyte units. The interarrival rate of requests was Poisson distribution.

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 and 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 flash memory utilization, flash memory size, flash segment size, and locality of reference, was examined in detail in the remaining sections.

6.3.1. Performance results for HP traces

The simulated flash memory was a 24 Mbyte Flash Memory Card with 128 Kbytes erase segments. 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 space, and then hplajw traces were used in the simulation.

Fig. 14 shows that there were large performance gaps between one segment cleaning class (M1, M2, and M3) and separate 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 of each segment selection algorithm was greatly improved: 60% $\sim$ 65% of erase operations were reduced, 86% $\sim$ 93.4% of blocks copied were reduced, and flash memory was more evenly worn. 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. CAT policy performed best in the wear leveling.

We conclude that data redistribution methods have more impact on the cleaning effectiveness than segment selection algorithms. Separate segment cleaning can

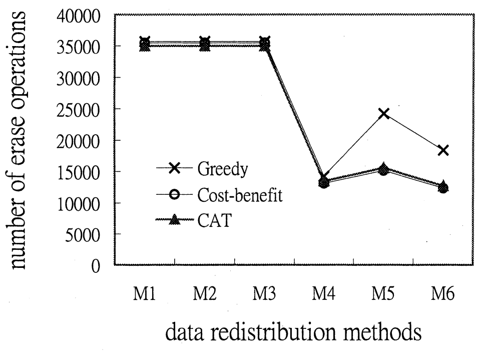
Table 6
Validation for simulated performance

		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	17035	8834	225283
	difference	0%	0%	0.39%	0.52%	0.08%	0.1%
Cost-benefit	actual	1568	0	7265	176634	5733	129142
	simulated	1568	0	7237	175769	5666	127049
	difference	0%	0%	0.39%	0.49%	1.17%	1.62%
CAT	actual	1568	0	7241	175891	4138	79705
	simulated	1568	0	7295	177562	3987	75005
	difference	0%	0%	0.74%	0.94%	3.65%	5.9%

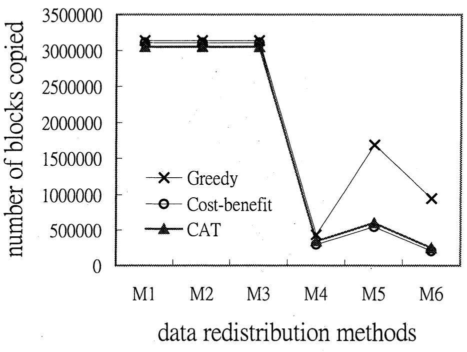
largely reduce the number of erase operations performed and the number of blocks copied. M6 is more effective than M4 in separating hot and cold data. To achieve the best performance, an effective data redistribution method must be used with an effective segment selection algorithm.

6.3.2. Impact of flash memory utilization

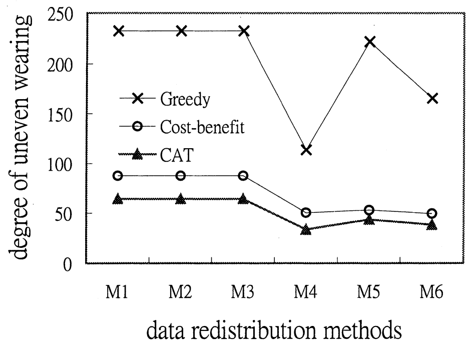
To find out how performance varied for varying utilization, we wrote blocks in sequence to fill the flash memory to various levels of utilization before each run of simulation, then hplajw traces were used. Fig. 15 shows that as utilization increased, the performance of one segment cleaning decreased dramatically while separate segment cleaning decreased gradually. The decrease is because less free space was left and more cleaning work was needed. This difference is because hot data and cold data were less likely to be mixed for M4 and M6. Greedy policy performed worst. For Cost-benefit policy, M6 outperformed M1 by 10.2% ~ 75.6% and outperformed M4 by 0.2% ~ 8.9% in reducing the number of erase operations. For CAT policy, M6



(a) Number of erase operations performed.

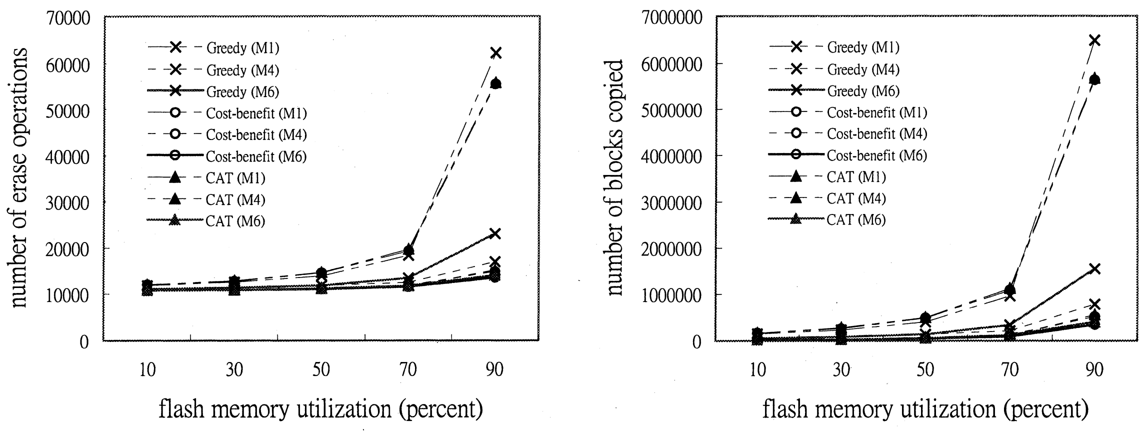


(b) Number of blocks copied.



(c) Degree of uneven wearing.

Fig. 14. Performance results for HP traces. (a) Number of erase operations performed; (b) Number of blocks copied; (c) Degree of uneven wearing.



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

Fig. 15. Varying flash memory utilization. The flash memory was 24 Mbytes with 128-Kbyte erase segments. Block size was 1 Kbytes and hplajw traces were used. Because M1, M2, and M3 had the same performances when one segment was cleaned at a time, only M1 was displayed; (a) Number of erase operations performed; (b) Number of blocks copied.

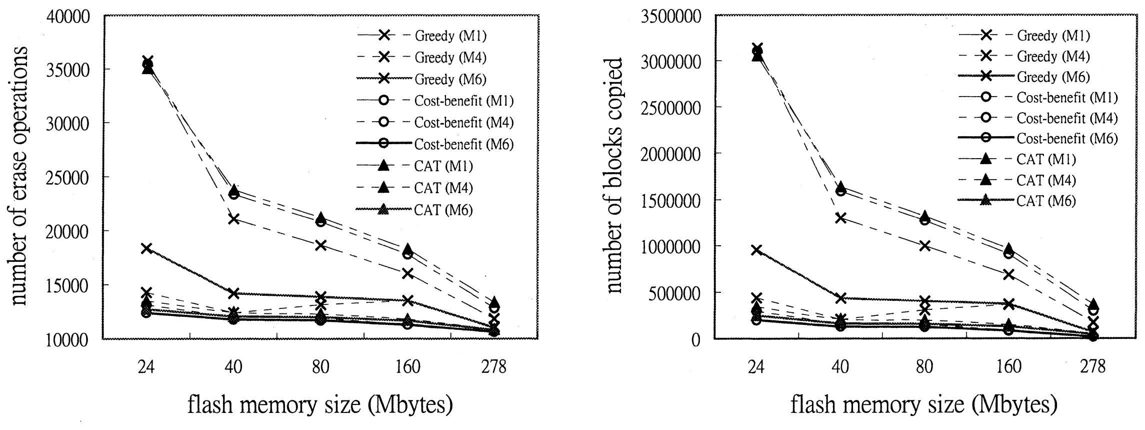
outperformed M1 by 10.2% ~ 74.9% and outperformed M4 by 0.2% ~ 7.7%. The results show that M6 is the most effective data redistribution method for various degrees of utilization.

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. Fig. 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 sizes. M6 performed best while 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.

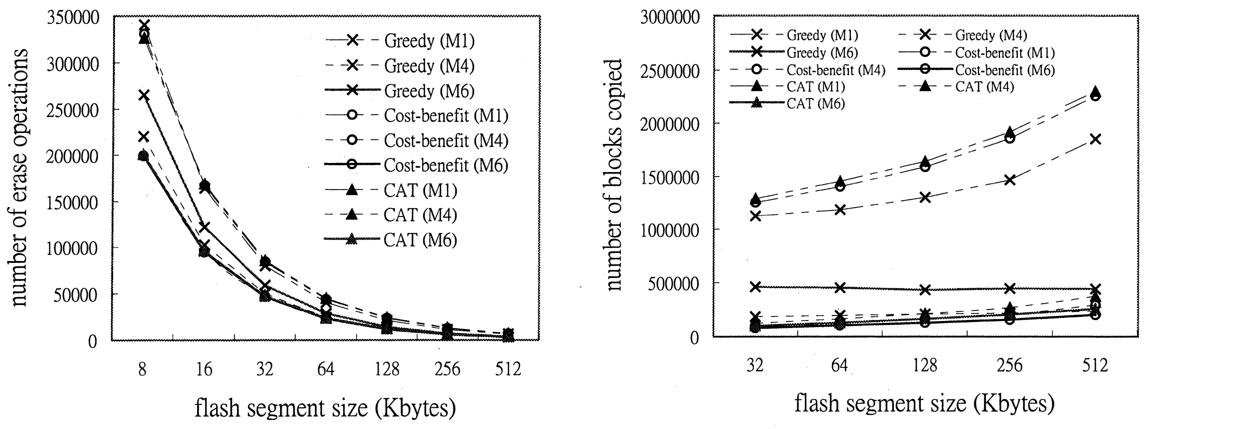
6.3.4. *Impact of erase segment size*

Though erase segment size is fixed by hardware manufacturers, we measured the impact of erase segment size. Fig. 17 shows that the number of erase operations decreased at the same rate as the increase of segment size. This decrease is because more space was reclaimed at once as segment size become larger. CAT policy performed best among all segment selection algorithms. M6 performed best among all data redistribution methods. However, more valid blocks in the



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

Fig. 16. Varying flash memory size. The segment size was 128 Kbytes and block size was 1 Kbytes. The flash memory utilization was set to 85% initially and hplajw traces were used. (a) Number of erase operations performed; (b) Number of blocks copied.



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

Fig. 17. Varying flash segment size. The flash memory was 40 Mbytes and block size was 1 Kbytes. The utilization was set to 85% initially and hplajw traces were used. (a) Number of erase operations performed; (b) Number of blocks copied.

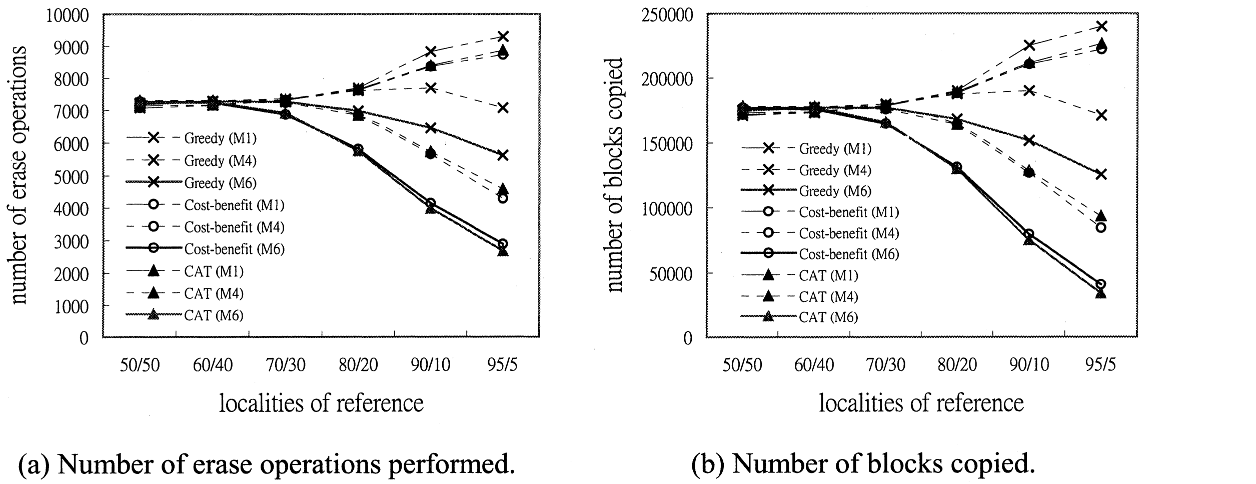
cleaned segment must be copied, as segment size became larger. This effect is especially severe for M1.

6.3.5. *Impact of locality of reference*

We investigated the impact of locality of reference for various data redistribution methods. Therefore, generated workloads for various degrees of localities were used in this simulation. Fig. 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 performance of M6 increased rapidly. M6 significantly outperformed M1 and M4. The result shows that M6 is the most effective method to separate hot data from cold data, such that cleaning overhead is the lowest when locality is above 60/40. 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; deJonge 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 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 flash memory-based storage systems. Besides, a large buffer to accommodate



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

Fig. 18. Varying localities of reference. The flash memory was 24 Mbytes with 128-Kbyte segments and the utilization was set to 90%. The workload for locality of reference was used. (a) Number of erase operations performed; (b) Number of blocks copied.

data blocks from several segments may not be affordable for a low-end resource-limited mobile computer. Their *age sort* scheme used to separate hot data from cold data has limited effect when only one segment is cleaned at a time. Furthermore, 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). 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.

Douglis et al. (Douglis et al., 1994) provides a detailed discussion of storage alternatives for mobile computers. They found that the erasure unit of flash memory could significantly affect performance. They also found that the flash memory utilization has substantial impacts: at 95% utilization as compared to 40% utilization, energy consumption is increased by 70%~190%, write response time is degraded by 30%, and lifetime is decreased by up to a third.

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, b) 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 (Kawaguchi et al., 1995; Wu and Zwaenepoel, 1994).

eNVy (Wu and Zwaenepoel, 1994), a storage system for flash memory, employs hardware support of copy-on-write and page remapping techniques to provide transparent update-in-place semantics. Their *hybrid cleaning* combines *FIFO* and *locality gathering* in cleaning segments, aiming to perform well for both uniform access and high localities of reference. eNVy considers only flash memory write cost in evaluating the effectiveness of cleaning policies. However, erasure cost dominates the total cleaning cost. Our work focuses on reducing the number of erase operations while evenly wearing 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 from 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 data from 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, CAT 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 paper 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

Table 7
Comparison of various cleaning policies

Issues	Flash memory server (FMS)	Flash-memory based files 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 (u: utilization)	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 and how	* Separate segment data allocation for read-only data and writable data * Separate segment cleaning with fine-grained separation for hot blocks and cold blocks (the hot degree is based on update-times/age)	* Separate segment cleaning with separation for hot and cold segments 1	* Locality gathering – Locality preservation Data are flushed back to the same segment where they come from – Active data redistribution Lower utilization of hot segments and increase utilization of cold segments, by migrating data between neighbor segments (hot data are moved towards the bottom and cold data are moved up to the top) * 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

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.

For the concerns about whether flash memory wears out in the life of the devices. It is expected that flash memory will be largely used in life. More and more applications will use flash memory as their storage systems and large write and erase operations will be created. Wear-leveling will be very important especially when the flash memory size is increased. A good cleaning policy can maximize flash memory usage and use flash memory in a cost effectively way.

Acknowledgements

We are grateful to John Wilkes at Hewlett Packard Laboratories for providing valuable comments and 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. And finally, the reviewers' comments helped to improve this paper. This research was supported in part by the National Science Council of the Republic of China under grant No. NSC866-2221-E009-021.

References

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

Ballard, N., 1994. State of PDAs and Other Pen-Based Systems, Pen Computing Magazine, 14–19.

Baker, M., Asami, S., Deprit, E., Ousterhout, J., Seltzer, M., 1992. Non-Volatile Memory for Fast, Reliable File Systems, Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, 10–22.

Blackwell, T., Harris, J., Seltzer, M., 1995. Heuristic cleaning algorithms in log-structured file systems, Proceedings of the 1995 USENIX Technical Conference, New Orleans, LA, 277–288.

Caceres, R., Douglis, F., Li, K., Marsh, B., 1993. Operating system implications of solid-state mobile computers, Fourth Workshop on Workstation Operating Systems, Napa, CA, 21–27.

Chiang, M.L., Lee, Paul C.H., Chang, R.C., 1997. Managing flash memory in personal communication devices, Proceedings of the 1997 International Symposium on Consumer Electronics (ISCE'97), Singapore, 177–182.

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

Douglis, F., Caceres, R., Kaashoek, F., Li, K., Marsh, B., Tauber, J.A., 1994. Storage alternatives for mobile computers, Proceedings of the 1st Symposium on Operating Systems Design and Implementation, 25–37.

Halfhill, T.R., 1993. PDAs Arrive But Aren't Quite Here Yet. BYTE 18 (11), 66–86.

Hinds, D., 1997a. Linux PCMCIA HOWTO, http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-HOWTO.html, vol. 2.5.

Hinds, D., 1997b. Linux PCMCIA Programmer's Guide, http://hyper.stanford.edu/~dhinds/pcmcia/doc/PCMCIA-PROG.html, vol. 1.38.

Intel, 1994. Flash Memory.

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

deJonge, W., Kaashoek, M.F., Hsieh, W.C., 1993. Logical disk: A simple new approach to improving file system performance, Technical Report MIT/LCS/TR-566, Massachusetts Institute of Technology.

Kawaguchi, A., Nishioka, S., Motoda, H., 1995. A flash-memory based file system, Proceedings of the 1995 USENIX Technical Conference, New Orleans, LA, 155–164.

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

Li, K., Kumpf, R., Horton, P., Anderson, T., 1994. A quantitative analysis of disk drive power management in portable computers, Proceedings of the 1994 Winter USENIX, San Francisco, CA, 279–291.

Marsh, B., Douglis, F., Krishnan, P., 1994. Flash memory file caching for mobile computers, Proceedings of the 27 Hawaii International Conference on System Sciences, Maui, HI, 451–460.

Matthews, J. N., Roselli, D., Costello, A. M., Wang, R. Y., Anderson, T. E., 1997. Improving the performance of log-structured file systems with adaptive methods, Proceedings of the Sixteenth ACM Symposium on Operating System Principles, Saint Malo, France.

Rosenblum, M., 1992. The design and implementation of a log-structured file system, Ph.D. Thesis, University of California, Berkeley.

Rosenblum, M., Ousterhout, J.K., 1992. The design and implementation of a log-structured file system. ACM Transactions on Computer Systems 10 (1), 26–52.

Ruemmler, C., Wilkes, J., 1993. UNIX disk access patterns, Proceedings of the 1993 Winter USENIX, San Diego, CA, 405–420.

SanDisk Corporation, 1993. SanDisk SDP Series OEM Manual.

Seltzer, M., Bostic, K., McKusick, M. K., Staelin, C., 1993. An implementation of a log-structured file system for UNIX, Proceedings of the 1993 Winter USENIX, 307–326.

Torelli, P., 1995. The microsoft flash file system, Dr. Dobb's Journal, 62–72.

Wilkes, J., Golding, R., Staelin, C., Sullivan, T., 1996. The HP AutoRAID hierarchical storage system. ACM Transactions on Computer Systems 14 (1), 108–136.

Wu, M., Zwaenepoel, W., 1994. eNVy: A non-volatile, main memory storage system, Proceedings of the sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, 86–97.

Mei-Ling Chiang received the B.S. degree in Management Information Science from National Chengchi University, Taipei, Taiwan, in 1989, and the M.S. degree in Computer and Information Science from National Chiao Tung University, Hsinchu, Taiwan, in 1993. She is currently working toward the Ph.D. degree in computer and Information Science at National Chiao Tung University. Her research interests include operating systems and mobile computing.

Rui-Chuan Chang received the B.S. degree in 1979, the M.S. degree in 1981, and his Ph.D. degree in 1984, all in computer engineering from National Chio Tung University, Taiwan. In August 1983, he joined the Department of Computer and Information Science at

National Chiao Tung University as a Lecturer. Now he is a Professor in the Department of Computer and Information Science. He is also an Associate Research Fellow in the Institute of Information Science, Academia Sinica, Taipei, Taiwan. He is currently the director of Computer Center and University Library of National Chiao Tung Univeristy. His current research interests include operating systems, wireless communication, multimedia systems, and computer graphics.